

**TRAINING & REFERENCE**

**murach's**  
**Java**  
**SE 6**

**Joel Murach**  
**Andrea Steelman**



**MIKE MURACH & ASSOCIATES, INC.**

*4340 N. Knoll Ave. • Fresno, CA 93722*

[www.murach.com](http://www.murach.com) • [murachbooks@murach.com](mailto:murachbooks@murach.com)

**Authors:** Joel Murach  
Andrea Steelman  
**Editor:** Anne Boehm  
**Cover Design:** Zylka Design  
**Production:** Judy Taylor

## **Books for Java programmers**

*Murach's Java SE 6*  
*Murach's Oracle SQL and PL/SQL*  
*Murach's Java Servlets and JSP (Second Edition)*

## **Books for .NET programmers**

*Murach's C# 2008*  
*Murach's ADO.NET 3.5, LINQ and the Entity Framework with C# 2008*  
*Murach's ASP.NET 3.5 Web Programming with C# 2008*  
*Murach's Visual Basic 2008*  
*Murach's ASP.NET 3.5 Web Programming with VB 2008*  
*Murach's ADO.NET 3.5, LINQ, and the Entity Framework with VB 2008*  
*Murach's SQL Server 2008 for Developers*

## **Books for IBM mainframe programmers**

*Murach's OS/390 and z/OS JCL*  
*Murach's Mainframe COBOL*  
*Murach's CICS for the COBOL Programmer*

## **Books for Web programmers**

*Murach's JavaScript and DOM Scripting*

© 2007, Mike Murach & Associates, Inc.  
All rights reserved.  
Printed in the United States of America

10 9 8 7 6 5 4 3  
ISBN-13: 978-1-890774-42-4  
ISBN-10: 1-890774-42-1

# Contents

Introduction

xv

## **Section 1 Essential Java skills**

---

Chapter 1	How to get started with Java	3
Chapter 2	Introduction to Java programming	45
Chapter 3	How to work with data	91
Chapter 4	How to code control statements	121
Chapter 5	How to validate input data	153

## **Section 2 Object-oriented programming with Java**

---

Chapter 6	How to define and use classes	177
Chapter 7	How to work with inheritance	223
Chapter 8	How to work with interfaces	261
Chapter 9	Other object-oriented programming skills	295

## **Section 3 More Java essentials**

---

Chapter 10	How to work with arrays	321
Chapter 11	How to work with collections and generics	345
Chapter 12	How to work with dates and strings	387
Chapter 13	How to handle exceptions	413
Chapter 14	How to work with threads	437

## **Section 4 GUI programming with Java**

---

Chapter 15	How to get started with Swing	475
Chapter 16	How to work with controls and layout managers	509
Chapter 17	How to handle events and validate data	547
Chapter 18	How to develop applets	595

## **Section 5 Data access programming with Java**

---

Chapter 19	How to work with text and binary files	623
Chapter 20	How to work with XML	679
Chapter 21	How to use JDBC to work with databases	713
Chapter 22	How to work with a Derby database	759

## **Reference aids**

---

Appendix A	How to use the downloadable files for this book	791
Index		794

# Expanded contents

## Section 1 Essential Java skills

---

### Chapter 1 How to get started with Java

<b>Introduction to Java .....</b>	<b>4</b>
Toolkits and platforms .....	4
Java compared to C++ .....	4
Java compared to C# .....	4
Applications, applets, and servlets .....	6
How Java compiles and interprets code .....	8
<b>How to prepare your system for using Java .....</b>	<b>10</b>
How to install the JDK .....	10
A summary of the directories and files of the JDK .....	12
How to set the command path .....	14
How to set the class path .....	16
<b>How to use TextPad to work with Java .....</b>	<b>18</b>
How to install TextPad .....	18
How to use TextPad to save and edit source code .....	20
How to use TextPad to compile source code .....	22
How to use TextPad to run an application .....	22
Common error messages and solutions .....	24
<b>How to use the command prompt to work with Java .....</b>	<b>26</b>
How to compile source code .....	26
How to run an application .....	26
How to compile source code with a switch .....	28
Essential DOS skills for working with Java .....	30
<b>How to use the documentation for the J2SE API .....</b>	<b>32</b>
How to install the API documentation .....	32
How to navigate the API documentation .....	34
<b>Introduction to Java IDEs .....</b>	<b>36</b>
The Eclipse IDE for Java .....	36
The NetBeans IDE .....	38
The BlueJ IDE .....	38

### Chapter 2 Introduction to Java programming

<b>Basic coding skills .....</b>	<b>46</b>
How to code statements .....	46
How to code comments .....	46
How to create identifiers .....	48
How to declare a class .....	50
How to declare a main method .....	52
<b>How to work with numeric variables .....</b>	<b>54</b>
How to declare and initialize variables .....	54
How to code assignment statements .....	56
How to code arithmetic expressions .....	56
<b>How to work with string variables .....</b>	<b>58</b>
How to create a String object .....	58
How to join and append strings .....	58
How to include special characters in strings .....	60

	<b>How to use Java classes, objects, and methods .....</b>	<b>62</b>
	How to import Java classes .....	62
	How to create objects and call methods .....	64
	How to use the API documentation to research Java classes .....	66
	<b>How to use the console for input and output .....</b>	<b>68</b>
	How to use the System.out object to print output to the console .....	68
	How to use the Scanner class to read input from the console .....	70
	Examples that get input from the console .....	72
	<b>How to code simple control statements .....</b>	<b>74</b>
	How to compare numeric variables .....	74
	How to compare string variables .....	74
	How to code if/else statements .....	76
	How to code while statements .....	78
	<b>Two illustrative applications .....</b>	<b>80</b>
	<b>How to test and debug an application .....</b>	<b>84</b>
<b>Chapter 3</b>	<b>How to work with data</b>	
	<b>Basic skills for working with data .....</b>	<b>92</b>
	The eight primitive data types .....	92
	How to initialize variables .....	94
	How to initialize constants .....	94
	How to code assignment statements and arithmetic expressions .....	96
	How to use the shortcut assignment operators .....	98
	How to work with the order of precedence .....	100
	How to work with casting .....	102
	<b>How to use Java classes for working with data types .....</b>	<b>104</b>
	How to use the NumberFormat class .....	104
	How to use the Math class .....	106
	How to use the Integer and Double classes .....	108
	<b>The formatted Invoice application .....</b>	<b>110</b>
	The code for the application .....	110
	How to analyze the data problems in the Invoice application .....	112
	<b>How to use the BigDecimal class for working with decimal data .....</b>	<b>114</b>
	The constructors and methods of the BigDecimal class .....	114
	How to use BigDecimal arithmetic in the Invoice application .....	116
<b>Chapter 4</b>	<b>How to code control statements</b>	
	<b>How to code Boolean expressions .....</b>	<b>122</b>
	How to compare primitive data types .....	122
	How to compare strings .....	124
	How to use the logical operators .....	126
	<b>How to code if/else and switch statements .....</b>	<b>128</b>
	How to code if/else statements .....	128
	How to code switch statements .....	130
	An enhanced version of the Invoice application .....	132
	<b>How to code loops .....</b>	<b>134</b>
	How to code while and do-while loops .....	134
	How to code for loops .....	136
	The Future Value application .....	138
	How to code nested for loops .....	140

	<b>How to code break and continue statements .....</b>	<b>142</b>
	How to code break statements .....	142
	How to code continue statements .....	144
	<b>How to code and call static methods .....</b>	<b>146</b>
	How to code static methods .....	146
	How to call static methods .....	146
	The Future Value application with a static method .....	148
<b>Chapter 5</b>	<b>How to validate input data</b>	
	<b>How to handle exceptions .....</b>	<b>154</b>
	How exceptions work .....	154
	How to catch exceptions .....	156
	The Future Value application with exception handling .....	158
	<b>How to validate data .....</b>	<b>160</b>
	How to prevent exceptions from being thrown .....	160
	How to validate a single entry .....	162
	How to use generic methods to validate an entry .....	164
	<b>The Future Value application with data validation .....</b>	<b>166</b>
<b>Section 2</b>	<b>Object-oriented programming with Java</b>	
<b>Chapter 6</b>	<b>How to define and use classes</b>	
	<b>An introduction to classes .....</b>	<b>178</b>
	How classes can be used to structure an application .....	178
	How encapsulation works .....	180
	The relationship between a class and its objects .....	182
	<b>How to code a class that defines an object .....</b>	<b>184</b>
	The code for the Product class .....	184
	How to code instance variables .....	186
	How to code constructors .....	188
	How to code methods .....	190
	How to overload methods .....	192
	How to use the this keyword .....	194
	<b>How to create and use an object .....</b>	<b>196</b>
	How to create an object .....	196
	How to call the methods of an object .....	198
	How primitive types and reference types are passed to a method .....	200
	A ProductDB class that creates a Product object .....	202
	A ProductApp class that uses a Product object .....	204
	<b>How to code and use static fields and methods .....</b>	<b>206</b>
	How to code static fields and methods .....	206
	How to call static fields and methods .....	208
	How to code a static initialization block .....	210
	When to use static fields and methods .....	210
	<b>The Line Item application .....</b>	<b>212</b>
	The console .....	212
	The classes used by the Line Item application .....	212
	The LineItemApp class .....	214
	The Validator class .....	214
	The LineItem class .....	218

<b>Chapter 7</b>	<b>How to work with inheritance</b>	
	<b>An introduction to inheritance .....</b>	<b>224</b>
	How inheritance works .....	224
	How the Java API uses inheritance .....	226
	How the Object class works .....	228
	How to use inheritance in your applications .....	230
	<b>Basic skills for working with inheritance .....</b>	<b>232</b>
	How to create a superclass .....	232
	How to create a subclass .....	234
	How polymorphism works .....	236
	<b>The Product application .....</b>	<b>238</b>
	The console .....	238
	The ProductApp class .....	240
	The Product, Book, and Software classes .....	242
	The ProductDB class .....	242
	<b>More skills for working with inheritance .....</b>	<b>246</b>
	How to get information about an object's type .....	246
	How to cast objects .....	248
	How to compare objects .....	250
	<b>How to work with the abstract and final keywords .....</b>	<b>252</b>
	How to work with the abstract keyword .....	252
	How to work with the final keyword .....	254
<b>Chapter 8</b>	<b>How to work with interfaces</b>	
	<b>An introduction to interfaces .....</b>	<b>262</b>
	A simple interface .....	262
	Interfaces compared to abstract classes .....	264
	Some interfaces of the Java API .....	266
	<b>How to work with interfaces .....</b>	<b>268</b>
	How to code an interface .....	268
	How to implement an interface .....	270
	How to inherit a class and implement an interface .....	272
	How to use an interface as a parameter .....	274
	How to use inheritance with interfaces .....	276
	<b>A Product Maintenance application that uses interfaces .....</b>	<b>278</b>
	The class diagram .....	278
	The console .....	280
	The DAOFactory class .....	282
	The ProductTextFile class .....	282
	The ProductMaintApp class .....	284
	<b>How to implement the Cloneable interface .....</b>	<b>288</b>
<b>Chapter 9</b>	<b>Other object-oriented programming skills</b>	
	<b>How to work with packages .....</b>	<b>296</b>
	How to store classes in a package .....	296
	How to compile the classes in a package .....	298
	How to make the classes of a package available to other classes .....	300
	<b>How to use javadoc to document a package .....</b>	<b>302</b>
	How to add javadoc comments to a class .....	302
	How to use HTML and javadoc tags in javadoc comments .....	304
	How to generate the documentation for a package .....	306
	How to view the documentation for a package .....	306

<b>How to code classes that are closely related .....</b>	<b>308</b>
How to code more than one class per file .....	308
An introduction to nested classes .....	310
<b>How to work with enumerations .....</b>	<b>312</b>
How to declare an enumeration .....	312
How to use an enumeration .....	312
How to enhance an enumeration .....	314
How to work with static imports .....	314

## **Section 3 More Java essentials**

---

### **Chapter 10 How to work with arrays**

<b>Basic skills for working with arrays .....</b>	<b>322</b>
How to create an array .....	322
How to assign values to the elements of an array .....	324
How to use for loops with arrays .....	326
How to use enhanced for loops with arrays .....	328
<b>More skills for working with arrays .....</b>	<b>330</b>
The methods of the Arrays class .....	330
Code examples that work with the Arrays class .....	332
How to implement the Comparable interface .....	334
How to create a reference to an array .....	336
How to copy an array .....	336
<b>How to work with two-dimensional arrays .....</b>	<b>338</b>
How to work with rectangular arrays .....	338
How to work with jagged arrays .....	340

### **Chapter 11 How to work with collections and generics**

<b>An introduction to Java collections .....</b>	<b>346</b>
A comparison of arrays and collections .....	346
An overview of the Java collection framework .....	348
Commonly used collection classes .....	348
An introduction to generics .....	350
<b>How to use the ArrayList class .....</b>	<b>352</b>
The ArrayList class .....	352
Code examples that work with array lists .....	354
<b>An Invoice application that uses an array list .....</b>	<b>356</b>
<b>How to use the LinkedList class .....</b>	<b>362</b>
The LinkedList class .....	362
Code examples that work with linked lists .....	364
A class that uses a linked list to implement a generic queue .....	366
<b>An enhanced version of the Invoice application .....</b>	<b>368</b>
<b>How to work with maps .....</b>	<b>374</b>
The HashMap and TreeMap classes .....	374
Code examples that work with hash maps and tree maps .....	376
<b>How to work with legacy collections .....</b>	<b>378</b>
An introduction to legacy collection classes .....	378
How to use an untyped collection .....	380
How to use wrapper classes with untyped collections .....	382



**Chapter 12 How to work with dates and strings**

<b>How to work with dates and times .....</b>	<b>388</b>
How to use the <code>GregorianCalendar</code> class to set dates and times .....	388
How to use the <code>Calendar</code> and <code>GregorianCalendar</code> fields and methods .....	390
How to use the <code>Date</code> class .....	392
How to use the <code>DateFormat</code> class to format dates and times .....	394
A <code>DateUtils</code> class that provides methods for handling dates .....	396
An <code>Invoice</code> class that includes an invoice date .....	398
<b>How to work with the <code>String</code> class .....</b>	<b>400</b>
Constructors of the <code>String</code> class .....	400
Code examples that create strings .....	400
Methods of the <code>String</code> class .....	402
Code examples that work with strings .....	404
<b>How to work with the <code>StringBuilder</code> class .....</b>	<b>406</b>
Constructors and methods of the <code>StringBuilder</code> class .....	406
Code examples that work with the <code>StringBuilder</code> class .....	408

**Chapter 13 How to handle exceptions**

<b>An introduction to exceptions .....</b>	<b>414</b>
The exception hierarchy .....	414
How exceptions are propagated .....	416
<b>How to work with exceptions .....</b>	<b>418</b>
How to use the <code>try</code> statement .....	418
How to use the <code>finally</code> clause .....	420
How to use the <code>throws</code> clause .....	422
How to use the <code>throw</code> statement .....	424
How to use the constructors and methods of the <code>Throwable</code> class .....	426
<b>How to work with custom exception classes .....</b>	<b>428</b>
How to create your own exception class .....	428
How to use exception chaining .....	430
<b>How to work with assertions .....</b>	<b>432</b>
How to code <code>assert</code> statements .....	432
How to enable and disable assertions .....	432

**Chapter 14 How to work with threads**

<b>An introduction to threads .....</b>	<b>438</b>
How threads work .....	438
Typical uses for threads .....	438
Classes and interfaces for working with threads .....	440
The life cycle of a thread .....	442
<b>How to create threads .....</b>	<b>444</b>
Constructors and methods of the <code>Thread</code> class .....	444
How to create a thread by extending the <code>Thread</code> class .....	446
How to create a thread by implementing the <code>Runnable</code> interface .....	448
<b>How to manipulate threads .....</b>	<b>450</b>
How to put a thread to sleep .....	450
How to set a thread's priority .....	452
How to interrupt a thread .....	454
<b>How to synchronize threads .....</b>	<b>456</b>
How to create synchronized threads .....	456
How to use the <code>wait</code> and <code>notifyAll</code> methods to communicate among threads .....	458
<b>The Order Queue application .....</b>	<b>460</b>

## **Section 4 GUI programming with Java**

---

### **Chapter 15 How to get started with Swing**

<b>An introduction to the Swing classes .....</b>	<b>476</b>
The user interface for the Future Value Calculator application .....	476
The inheritance hierarchy for Swing components .....	478
Methods of the Component class .....	480
<b>How to work with frames .....</b>	<b>482</b>
How to display a frame .....	482
How to set the default close operation .....	484
How to center a frame using the Toolkit class .....	486
<b>How to work with panels, buttons, and events .....</b>	<b>488</b>
How to add a panel to a frame .....	488
How to add buttons to a panel .....	490
How to handle button events .....	492
<b>An introduction to layout managers .....</b>	<b>494</b>
How to use the Flow layout manager .....	494
How to use the Border layout manager .....	496
<b>How to work with labels and text fields .....</b>	<b>498</b>
How to work with labels .....	498
How to work with text fields .....	500
<b>The Future Value Calculator application .....</b>	<b>502</b>

### **Chapter 16 How to work with controls and layout managers**

<b>How to work with components .....</b>	<b>510</b>
A summary of the Swing components presented in this chapter .....	510
How to work with text areas .....	512
How to work with scroll panes .....	514
How to work with check boxes .....	516
How to work with radio buttons .....	518
How to work with borders .....	520
How to work with combo boxes .....	522
How to use event listeners with a combo box .....	524
How to work with lists .....	526
How to work with multiple selections in a list .....	528
How to work with list models .....	530
<b>How to work with layout managers .....</b>	<b>532</b>
A summary of the layout managers .....	532
How to work with the Grid Bag layout manager .....	534
How to set the constraints for a Grid Bag layout .....	536
An application that uses the Grid Bag layout manager .....	538

### **Chapter 17 How to handle events and validate data**

<b>How to handle events .....</b>	<b>548</b>
The Java event model .....	548
Two types of Java events .....	550
How to structure event handling code .....	552
How to implement an event listener in a panel class .....	554
How to implement an event listener as a separate class .....	556
How to implement an event listener as an inner class .....	558
How to implement separate event listeners for each event .....	560
How to implement event listeners as anonymous inner classes .....	562

<b>How to code low-level events .....</b>	<b>564</b>
A summary of low-level events .....	564
How to work with focus events .....	566
How to work with keyboard events .....	568
How to work with adapter classes .....	570
<b>How to validate Swing input data .....</b>	<b>572</b>
How to display error messages .....	572
How to validate the data entered into a text field .....	574
The SwingValidator class .....	576
How to validate multiple entries .....	578
<b>The Product Maintenance application .....</b>	<b>580</b>
 <b>Chapter 18 How to develop applets</b>	
<b>An introduction to applets .....</b>	<b>596</b>
A brief history of applets .....	596
Applet security issues .....	598
The inheritance hierarchy for applets .....	600
Four methods of an applet .....	600
<b>How to develop and test applets .....</b>	<b>602</b>
How to code an applet .....	602
How to code the HTML page for an applet .....	604
How to test an applet with the Applet Viewer .....	606
How to convert the HTML page for an applet .....	608
The code for the converted HTML page .....	610
How to test an applet from a web browser .....	612
How to deploy an applet .....	612
<b>How to use JAR files with applets .....</b>	<b>614</b>
How to create and work with JAR files .....	614
How to include a JAR file that contains an applet in an HTML page .....	616
 <b>Section 5 Data access programming with Java</b>	
 <b>Chapter 19 How to work with text and binary files</b>	
<b>Introduction to files and directories .....</b>	<b>624</b>
Constructors and methods of the File class .....	624
Code examples that work with directories and files .....	626
<b>Introduction to file input and output .....</b>	<b>628</b>
How files and streams work .....	628
A file I/O example .....	630
How to work with I/O exceptions .....	632
<b>How to work with text files .....</b>	<b>634</b>
How to connect a character output stream to a file .....	634
How to write to a text file .....	636
How to connect a character input stream to a file .....	638
How to read from a text file .....	640
An interface for working with file I/O .....	642
A class that works with a text file .....	644
<b>How to work with binary files .....</b>	<b>650</b>
How to connect a binary output stream to a file .....	650
How to write to a binary file .....	652
How to connect a binary input stream to a file .....	654
How to read from a binary file .....	656
Two ways to work with binary strings .....	658

<b>How to work with random-access files .....</b>	<b>660</b>
How to connect to a random-access file .....	660
How to read to and write from a random-access file .....	662
How to read and write fixed-length strings .....	664
A class that works with a random-access file .....	666
<b>Chapter 20    How to work with XML</b>	
<b>Introduction to XML .....</b>	<b>680</b>
An XML document .....	680
XML tags, declarations, and comments .....	682
XML elements .....	682
XML attributes .....	684
An introduction to DTDs .....	686
<b>How to view and edit an XML file .....</b>	<b>688</b>
How to view an XML file .....	688
How to edit an XML file .....	688
<b>An introduction to three XML APIs .....</b>	<b>690</b>
DOM .....	690
SAX .....	690
StAX .....	690
<b>How to use StAX to work with XML .....</b>	<b>692</b>
How to create an XMLStreamWriter object .....	692
How to write XML .....	694
How to create an XMLStreamReader object .....	696
How to read XML .....	698
A class that works with an XML file .....	702
<b>Chapter 21    How to use JDBC to work with databases</b>	
<b>How a relational database is organized .....</b>	<b>714</b>
How a table is organized .....	714
How the tables in a database are related .....	716
How the fields in a database are defined .....	718
<b>How to use SQL to work with the data in a database .....</b>	<b>720</b>
How to query a single table .....	720
How to join data from two or more tables .....	722
How to add, update, and delete data in a table .....	724
<b>An introduction to Java database drivers .....</b>	<b>726</b>
The four driver types .....	726
How to configure an ODBC driver .....	728
<b>How to use Java to work with a database .....</b>	<b>730</b>
How to connect to a database .....	730
How to return a result set .....	734
How to move the cursor through a result set .....	736
How to return data from a result set .....	738
How to modify data in a database .....	740
How to work with prepared statements .....	742
<b>Two classes for working with databases .....</b>	<b>744</b>
A utility class for working with strings .....	744
A class that works with a database .....	746
<b>An introduction to working with metadata .....</b>	<b>752</b>
How to work with metadata .....	752
How SQL data types map to Java data types .....	754

**Chapter 22 How to work with a Derby database**

<b>An introduction to Derby .....</b>	<b>760</b>
An overview of Derby .....	760
How to configure your system to work with a Derby database .....	762
<b>How to use the ij tool to work with a Derby database .....</b>	<b>764</b>
How to start and stop the ij tool .....	764
How to connect to and disconnect from a database .....	764
How to create a database and connect to it .....	766
How to run SQL statements .....	766
How to run SQL scripts from the ij prompt .....	768
How to run SQL scripts from the command prompt .....	770
<b>How to use JDBC to work with an embedded database .....</b>	<b>772</b>
How to set the default directory for the database .....	772
How to connect to the database and start the database engine .....	772
How to disconnect from the database and shut down the database engine .....	774
A class that creates an embedded Derby database .....	776
<b>How to use JDBC to work with a networked database .....</b>	<b>782</b>
How to start the Derby server .....	782
How to connect to a database on the server .....	784
<b>How to learn more about Derby .....</b>	<b>786</b>
How to view the Derby documentation .....	786
How to navigate through the documentation .....	786



# Introduction

Since its release in 1996, the Java language has established itself as one of the leading languages for object-oriented programming. Today, despite competition from Microsoft's .NET Platform, Java continues to be one of the leading languages for application development, especially for web applications. And that's going to continue for many years to come, for several reasons.

First, developers can obtain Java and a wide variety of tools for working with Java for free. Second, Java code can run on any modern operating system. Third, Java's development has been guided largely by the Java community, and Sun has committed to releasing Java as open source software. As a result, the Java platform is able to evolve according to the needs of the programmers who use the language.

## Who this book is for

---

This book is for anyone who wants to learn the core features of the Java language. It works if you have no programming experience at all. It works if you have programming experience with another language. It works if you already know an older version of Java and you want to get up-to-speed with the latest version. And it works if you've already read three or four other Java books and still don't know how to develop a real-world application.

If you're completely new to programming, the prerequisites are minimal. You just need to be familiar with the operation of the platform that you're using. If, for example, you're developing programs using Windows on a PC, you should know how to use Windows to perform tasks like opening, saving, printing, closing, copying, and deleting files.

## What version of Java this book supports

---

This book is designed to work with the Java Platform, Standard Edition 6 (Java SE 6). This edition of Java includes the Java Development Kit (JDK). For marketing reasons, Sun sometimes refers to this version of the JDK as JDK 6. However, from a developer's point of view, this version of the JDK is commonly referred to as version 1.6.0 or just 1.6, and Java SE 6 is sometimes referred to as Java 1.6 or Java 6.0.

As you work with Java SE 6, please keep in mind that all Java versions are upwards-compatible. That means that everything in this book will also work with any future versions of the JDK.

## **How to get the software you need**

---

You can download all of the software that you need for this book for free from the Internet. To make that easier for you, chapter 1 shows you how to download and install the JDK from the web site for Sun Microsystems. It also shows you how to download and install TextPad, a text editor that's designed for working with Java. As you will quickly see as you read this book, that's all the software you need for developing professional Java applications on your own.

## **How to use this book with Eclipse or NetBeans**

---

Although using the TextPad text editor that's described in chapter 1 is an easy way to get started with Java programming, you may prefer using an IDE (Integrated Development Environment) with this book, especially if you have previous experience with one. That's why we have created free tutorials that show how to use this book with two of the most popular IDEs for Java development: Eclipse and NetBeans. Although these tutorials don't present all of the features available from these powerful IDEs, they quickly teach the skills you need to create the applications presented in this book.

If you want to use one of these IDEs with this book, you can download the appropriate tutorial from our web site at this address:

`www.murach.com/books/jse6/ides.htm`

You can also download the source code for this book in a format that's compatible with the IDE that you choose. That way, you won't have to waste time creating a project for each application that you work on. Instead, you'll open existing projects.

## **What operating systems this book supports**

---

As you will see when you download Java, the Sun web site provides a version of JDK 1.6 that works with the Windows, Linux, and Solaris operating systems. As this book goes to press, the Mac OS X operating system includes JDK 1.4 and provides support for JDK 1.5, which you can use to develop most of the applications presented in this book. Before long, OS X will provide support for JDK 1.6. You can visit the Apple web site to learn more about Java development on a Mac and to download the latest version of the JDK.

However, since most Java development today is done under Windows, this book uses Windows to illustrate any platform-dependent procedures. As a result, if you're working on another platform, you may need to download information from the Sun web site about how to do some of those procedures on your system. Fortunately, this book requires just a few platform-dependent procedures.



## What you'll learn in this book

---

Unlike competing books, this one focuses on the practical features that you'll need for developing professional Java applications. Here's a quick tour:

- In section 1, you'll quickly master the basics of the Java language. In chapter 1, you'll learn how to install and configure the JDK and its documentation. In chapter 2, you'll learn how to write complete console applications that use the `Scanner` class to get input from the user. And by the end of chapter 5, you'll know how to code applications that use custom methods to validate user input so they won't crash and how to use the `BigDecimal` class to make accurate calculations for decimal data.
- In section 2, you'll learn how to use Java for object-oriented programming. In chapter 6, you'll learn how to create and use your own classes, which is the basis for developing applications that are easier to test, debug, and maintain. Then, in chapters 7 through 9, you'll learn how to develop more sophisticated classes that use inheritance, interfaces, packages, type-safe enumerations, and the factory pattern. In addition, you'll learn how to use the three-tiered architecture that's the standard used by most professionals for designing and developing object-oriented, database applications.
- In section 3, you'll learn more of the core Java features that you'll use all the time. In chapters 10 through 14, for instance, you'll learn how to work with arrays, collections, dates, strings, and exceptions. And in chapter 15, you'll learn how to use threads so your applications can perform two or more tasks at the same time. Along the way, you'll learn features that were introduced with JDK 1.5 like enhanced for loops, typed collections, generics, autoboxing, assertions, and the `StringBuilder` class. You'll also learn a couple new JDK 1.6 features for array reallocation and empty string checking.
- In section 4, you'll learn how to develop graphical user interfaces (GUIs) with Java. First, you'll learn how to use Swing components to develop real-world GUI applications that handle events, validate data, and populate objects. Then, you'll learn how to develop applets, a special type of Java application that can be downloaded from the Internet and run within a web browser.
- Because storing data is critical to most applications, section 5 shows you how to store the data for objects in a file or database. In chapter 19, you'll learn how to work with text files and binary files, including random-access files. In chapter 20, you'll learn how to use an API known as StAX (the Streaming API for XML) that was introduced with JDK 1.6 to work with XML documents and files. In chapter 21, you'll learn how to use JDBC to work with databases. And in chapter 22, you'll learn how to use the open-source Apache Derby database that's included as part of the Java SE 6 download.

## **Why you'll learn faster and better with this book**

---

Like all our books, this one has features that you won't find in competing books. That's why we believe that you'll learn faster and better with our book than with any other. Here are just three of those features.

- To help you develop applications at a professional level, this book presents complete, non-trivial applications. For example, chapter 17 presents a Product Maintenance application that uses presentation classes, business classes, and database classes. You won't find complete, real-world applications like this in other Java books even though studying these types of applications is the best way to master Java development.
- All of the information in this book is presented in our unique paired-pages format, with the essential syntax, guidelines, and examples on the right page and the perspective and extra explanation on the left page. This helps you learn more while reading less, and it helps you quickly find the information that you need when you use this book for reference.
- The exercises at the end of each chapter give you a chance to try out what you've just learned and to gain valuable, hands-on experience in Java programming. They guide you through the development of some of the book's applications, and they challenge you to apply what you've learned in new ways. And because we provide the starting code for the exercises from our web site, you get more practice in less time.

## **How our downloadable files make learning easier**

---

To make learning easier, you can download the source code, files, and databases for all the applications presented in this book from our web site ([www.murach.com](http://www.murach.com)). Then, you can view the complete code for these applications as you read each chapter; you can compile and run these applications to see how they work; and you can copy portions of code for use in your own applications.

You can also download the source code, files, and databases that you need for doing the exercises in this book. That way, you don't have to start every exercise from scratch. This takes the busywork out of doing these exercises. For more information about these downloads, please see appendix A.

## **Support materials for trainers and instructors**

---

If you're a corporate trainer or a college instructor who would like to use this book for a course, we offer an Instructor's CD that includes: (1) a complete set of PowerPoint slides that you can use to review and reinforce the content of the book; (2) instructional objectives that describe the skills a student should have upon completion of each chapter; (3) the solutions to the exercises in this

book; (4) projects that the students start from scratch; (5) solutions to those projects; and (6) test banks that measure mastery of those skills.

To learn more about this Instructor's CD and to find out how to get it, please go to our web site at [www.murach.com](http://www.murach.com) and click on the Trainers link or the Instructors link. Or, if you prefer, you can call Kelly at 1-800-221-5528 or send an email to [kelly@murach.com](mailto:kelly@murach.com).

## About Murach's Java Servlets and JSP

---

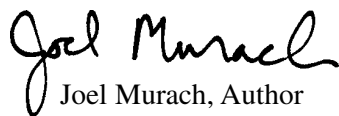
Since web programming is one of the primary uses of Java, we also offer a book on web programming called *Murach's Java Servlets and JSP*. It shows you how to use Java servlets and JavaServer Pages as you develop professional web applications. As you read that book, you'll discover that Java web programming requires most of the skills that are presented in this book. In fact, you need to know everything that's in this book except the GUI skills in section 4. That's why we see this book as the perfect companion for *Murach's Java Servlets and JSP*.

## Please let us know how this book works for you

---

When we started the first edition of this book, our goals were (1) to teach you Java as quickly and easily as possible and (2) to teach you the practical Java concepts and skills that you need for developing real-world business applications. Now, as this third edition goes to press, we hope that we've made the book even more effective. So if you have any comments about this book, we would appreciate hearing from you at [murachbooks@murach.com](mailto:murachbooks@murach.com).

Thanks for buying this book. We hope you enjoy reading it, and we wish you great success with your Java programming.

  
Joel Murach, Author

  
Andrea Steelman, Author



# Section 1

## Essential Java skills

This section gets you started quickly with Java programming. First, chapter 1 shows you how to compile and run Java applications, and chapter 2 introduces you to the basic skills that you need for developing Java applications. When you complete these chapters, you'll be able to write, test, and debug simple applications of your own.

After that, chapter 3 presents the details for working with numeric data. Chapter 4 presents the details for coding control statements. And chapter 5 shows you how to validate the data that's entered by the user. These are the essential skills that you'll use in almost every Java application that you develop. When you finish these chapters, you'll be able to write solid programs of your own. And you'll have the background that you need for learning how to develop object-oriented programs.



# 1

## How to get started with Java

Before you can begin learning the Java language, you need to install Java and learn how to use some tools for working with Java. So that's what you'll learn in this chapter. Since most Java developers use Windows, the examples in this chapter show how to use Java with Windows. However, the principles illustrated by these examples apply to all operating systems including Linux, Mac (OS X), and Solaris.

<b>Introduction to Java .....</b>	<b>4</b>
Toolkits and platforms .....	4
Java compared to C++ .....	4
Java compared to C# .....	4
Applications, applets, and servlets .....	6
How Java compiles and interprets code .....	8
<b>How to prepare your system for using Java .....</b>	<b>10</b>
How to install the JDK .....	10
A summary of the directories and files of the JDK .....	12
How to set the command path .....	14
How to set the class path .....	16
<b>How to use TextPad to work with Java .....</b>	<b>18</b>
How to install TextPad .....	18
How to use TextPad to save and edit source code .....	20
How to use TextPad to compile source code .....	22
How to use TextPad to run an application .....	22
Common error messages and solutions .....	24
<b>How to use the command prompt to work with Java .....</b>	<b>26</b>
How to compile source code .....	26
How to run an application .....	26
How to compile source code with a switch .....	28
Essential DOS skills for working with Java .....	30
<b>How to use the documentation for the Java SE API .....</b>	<b>32</b>
How to install the API documentation .....	32
How to navigate the API documentation .....	34
<b>Introduction to Java IDEs .....</b>	<b>36</b>
The Eclipse IDE for Java .....	36
The NetBeans IDE .....	38
The BlueJ IDE .....	38
<b>Perspective .....</b>	<b>40</b>

# Introduction to Java

---

In 1996, Sun Microsystems released a new programming language called Java. Today, Java has established itself as one of the most widely used object-oriented programming languages.

## Toolkits and platforms

---

Figure 1-1 describes all major releases of Java to date starting with version 1.0 and ending with version 1.6. Throughout Java's history, Sun has used the terms *Java Development Kit (JDK)* and *Software Development Kit (SDK)* to describe the Java toolkit. In this book, we'll use the term *JDK* since it's the most current and commonly used term. In addition, for marketing reasons, Sun uses the terms Java 5.0 and Java 6 to refer to versions 1.5 and 1.6 of Java. In this book, we'll use the 1.x style of numbering since it's consistent across all versions of Java and since this numbering is used by the documentation for Java.

With versions 1.2 through 1.5 of the JDK, the *Standard Edition (SE)* of Java was known as *Java 2 Platform, Standard Edition (J2SE)*, and the *Enterprise Edition (EE)* was known as the *Java 2 Platform, Enterprise Edition (J2EE)*. With version 1.6 of the JDK, Sun has simplified this naming convention. Now, the Standard Edition of Java is known as *Java SE*, and the Enterprise Edition is known as *Java EE*. This book will show you how to use the *Java SE 6*.

## Java compared to C++

---

When Sun's developers created Java, they tried to keep the syntax for Java similar to the syntax for C++ so it would be easy for C++ programmers to learn Java. In addition, they designed Java so its applications can be run on any computer platform. In contrast, C++ needs to have a specific compiler for each platform. Java was also designed to automatically handle many operations involving the creation and destruction of memory. This led to improved productivity for Java programmers, and it's a key reason why it's easier to develop programs and write bug-free code with Java than with C++.

To provide these features, the developers of Java had to sacrifice some speed (or performance) when compared to C++. For many types of applications, however, Java's relative slowness is not an issue.

## Java compared to C#

---

Microsoft's Visual C# language is similar to Java in many ways. Like Java, C# uses a syntax that's similar to C++ and that automatically handles memory operations. Also like Java, C# applications can run on any system that has the appropriate interpreter. Currently, however, only Windows provides the interpreter needed to run C# applications. In addition, C# applications are optimized for Windows. Because of that, C# is a good choice for developing applications for a Windows-only environment. However, many of the server computers that store critical enterprise data use Solaris or Linux. As a result, Java remains popular for developing programs that run on these servers.



## Java timeline

Year	Month	Event
1996	January	Sun releases Java Development Kit 1.0 (JDK 1.0).
1997	February	Sun releases Java Development Kit 1.1 (JDK 1.1).
1998	December	Sun releases the Java 2 Platform with version 1.2 of the Software Development Kit (SDK 1.2).
1999	August	Sun releases Java 2 Platform, Standard Edition (J2SE).
	December	Sun releases Java 2 Platform, Enterprise Edition (J2EE).
2000	May	Sun releases J2SE with version 1.3 of the SDK.
2002	February	Sun releases J2SE with version 1.4 of the SDK.
2004	September	Sun releases J2SE 5.0 with version 1.5 of the JDK.
2006	December	Sun releases Java SE 6 with version 1.6 of the JDK.

## Operating systems supported by Sun

Windows (2000, XP, Vista)	Solaris
Linux	Macintosh (OS X)

## Java compared to C++

Feature	Description
Syntax	Java syntax is similar to C++ syntax.
Platforms	Compiled Java code can be run on any platform that has a Java interpreter. C++ code must be compiled once for each type of system that it is going to be run on.
Speed	C++ runs faster than Java, but Java is getting faster with each new version.
Memory	Java handles most memory operations automatically, while C++ programmers must write code that manages memory.

## Java compared to C#

Feature	Description
Syntax	Java syntax is similar to C# syntax.
Platforms	Like compiled Java code, compiled C# code (MSIL) can be run on any system that has the appropriate interpreter. Currently, only Windows has an interpreter for MSIL.
Speed	C# runs faster than Java.
Memory	Both C# and Java handle most memory operations automatically.

## Description

- Versions 1.0, 1.1, 5, and 6 of Java are called the *Java Development Kit (JDK)*.
- Versions 1.2, 1.3, and 1.4 of Java are called the *Software Development Kit (SDK)*.
- The *Standard Edition (SE)* of Java contains the core class libraries that are necessary to develop Java applications.
- The *Enterprise Edition (EE)* of Java contains the additional class libraries that are typically used to create server-side Java applications such as web applications.

## Applications, applets, and servlets

---

Figure 1-2 describes the three types of programs that you can create with Java. First, you can use Java to create *applications*. This figure shows an application that uses a *graphical user interface (GUI)* to get user input and perform a calculation. In this book, you'll be introduced to a variety of applications with the emphasis on GUI applications that get data from files and databases.

One of the unique characteristics of Java is that you can use it to create a special type of web-based application known as an *applet*. For instance, this figure shows an applet that works the same way as the application above it. The main difference between an application and an applet is that an applet can be downloaded from a web server and can run inside a Java-enabled browser. As a result, you can distribute applets via the Internet or an intranet.

Although applets can be useful for creating a complex user interface within a browser, they have their limitations. First, you need to make sure a plug-in is installed on each client machine to be able to use the newer Java GUI components (such as Swing components). Second, since an applet runs within a browser on the client, it's not ideal for working with resources that run on the server, such as enterprise databases.

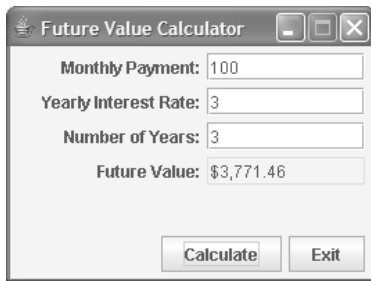
To provide access to enterprise databases, many developers use the Enterprise Edition of Java (Java EE) to create applications that are based on servlets. A *servlet* is a special type of Java application that runs on the server and can be called by a client, which is usually a web browser. This is also illustrated in this figure. Here, you can see that the servlet works much the same way as the applet. The main difference is that the code for the application runs on the server.

When a web browser calls a servlet, the servlet performs its task and returns the result to the browser, typically in the form of an HTML page. For example, suppose a browser requests a servlet that displays all unprocessed invoices that are stored in a database. Then, when the servlet is executed, it reads data from the database, formats that data within an HTML page, and returns the HTML page to the browser.

When you create a servlet-based application like the one shown here, all the processing takes place on the server and only HTML is returned to the browser. That means that anyone with an Internet or intranet connection, a web browser, and adequate security clearance can access and run a servlet-based application. Because of that, you don't need to install any special software on the client.

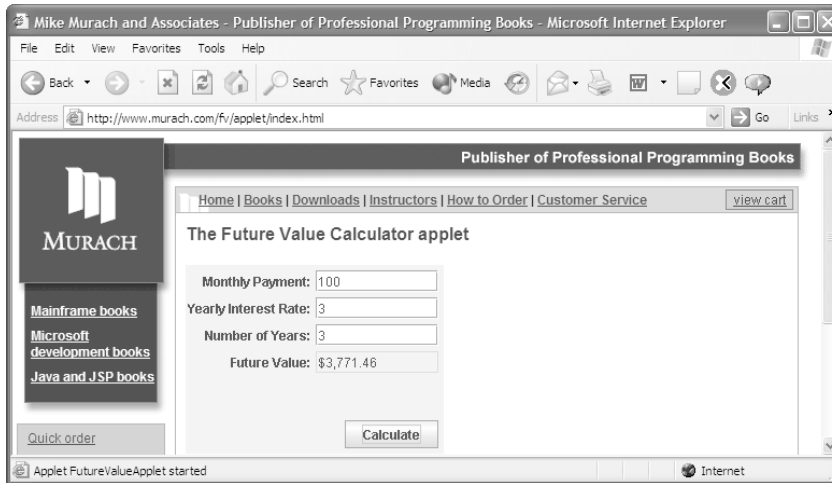
To make it easy to store the results of a servlet within an HTML page, the Java EE specification provides for *JavaServer Pages (JSPs)*. Most developers use JSPs together with servlets when developing server-side Java applications. Although servlets and JSPs aren't presented in this book, we cover this topic in a companion book, *Murach's Java Servlets and JSP*. For more information about this book, please visit our web site at [www.murach.com](http://www.murach.com).

## An application



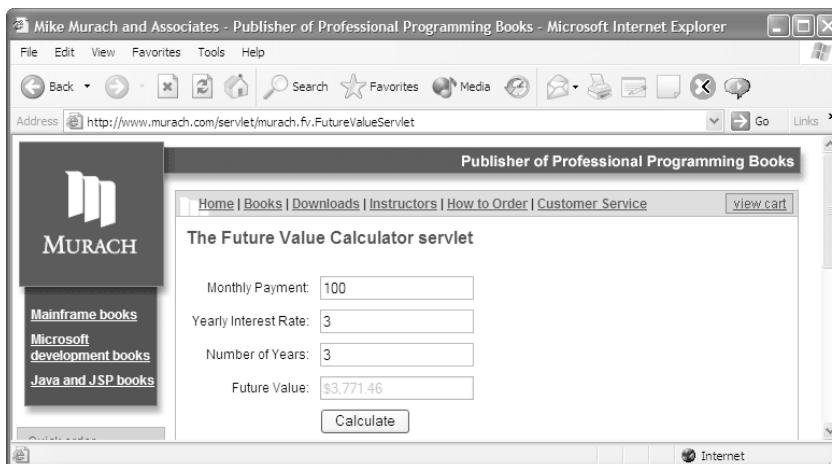
A screenshot of a Java application window titled "Future Value Calculator". It contains four input fields: "Monthly Payment:" with the value 100, "Yearly Interest Rate:" with the value 3, "Number of Years:" with the value 3, and "Future Value:" with the calculated value \$3,771.46. At the bottom are two buttons: "Calculate" and "Exit".

## An applet



A screenshot of a Microsoft Internet Explorer browser window showing the "Future Value Calculator applet" on the website "Mike Murach and Associates - Publisher of Professional Programming Books". The browser address bar shows "http://www.murach.com/fv/applet/index.html". The applet interface is identical to the standalone application, with input fields for Monthly Payment (100), Yearly Interest Rate (3), Number of Years (3), and Future Value (\$3,771.46), and "Calculate" and "Exit" buttons. A status bar at the bottom indicates "Applet FutureValueApplet started".

## A servlet



A screenshot of a Microsoft Internet Explorer browser window showing the "Future Value Calculator servlet" on the website "Mike Murach and Associates - Publisher of Professional Programming Books". The browser address bar shows "http://www.murach.com/servlet/murach.fv.FutureValueServlet". The servlet interface is identical to the standalone application, with input fields for Monthly Payment (100), Yearly Interest Rate (3), Number of Years (3), and Future Value (\$3,771.46), and a "Calculate" button.

## Description

- You can run the applet and servlet versions of the Future Value Calculator application shown above by going to [www.murach.com/fv](http://www.murach.com/fv).

Figure 1-2 Applications, applets, and servlets

## How Java compiles and interprets code

---

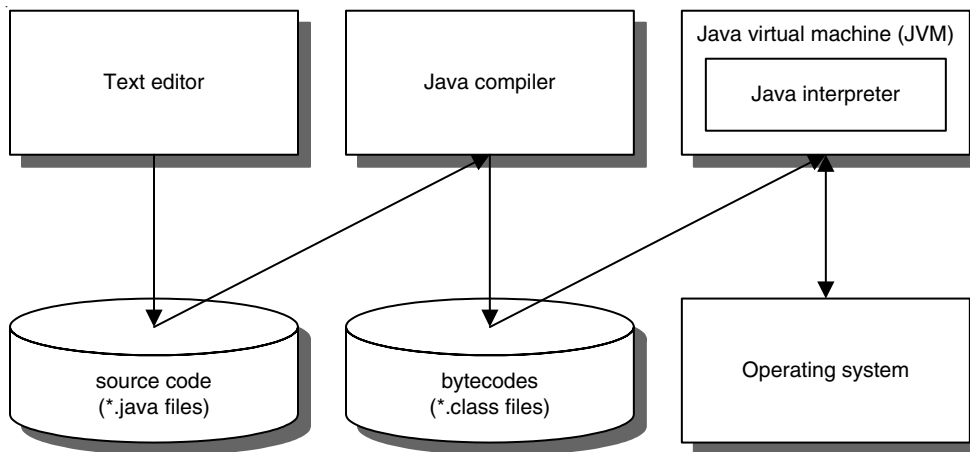
When you develop a Java application, you develop one or more *classes*. For each class, you write the Java statements that direct the operation of the class. Then, you use a Java tool to translate the Java statements into instructions that can be run by the computer. This process is illustrated in figure 1-3.

To start, you can use any text editor to enter and edit the Java *source code* for a class. These are the Java statements that tell the application what to do. Then, you use the *Java compiler* to compile the source code into a format known as Java *bytecodes*. At this point, the bytecodes can be run on any platform that has a *Java interpreter* to *interpret* (or translate) the Java bytecodes into code that can be understood by the underlying operating system. The Java interpreter is a concrete implementation of an abstract specification known as the *Java virtual machine (JVM)*. As a result, these two terms are often used interchangeably.

Since Java interpreters are available for all major operating systems, you can run Java on most platforms. This is what gives Java applications their *platform independence*. In contrast, C++ requires a specific compiler for each type of platform that its programs are going to run on.

In addition, most modern web browsers can be Java enabled. This allows applets, which are bytecodes that are downloaded from the Internet or an intranet, to run within a web browser. To make this work, Sun developed the *Java Plug-in*. This piece of software is similar to other browser plug-ins such as Apple QuickTime. It allows the browser to run Sun's current version of the Java interpreter. You'll learn more about this in chapter 18.

## How Java compiles and interprets code



### Description

- When you develop a Java application, you develop one or more *classes*.
- You can use any text editor to create, edit, and save the *source code* for a Java class. Source code files have the *java* extension.
- The *Java compiler* translates Java source code into a *platform-independent* format known as Java *bytecodes*. Files that contain Java bytecodes have the *class* extension.
- The *Java interpreter* executes Java bytecodes. Since Java interpreters exist for all major operating systems, Java bytecodes can be run on most platforms. A Java interpreter is an implementation of a *Java virtual machine (JVM)*.
- Most modern web browsers can be Java enabled. This lets applets run within these browsers. Sun provides a tool known as the *Java Plug-in* that allows you to specify the version of the Java interpreter that you want to use.

Figure 1-3 How Java compiles and interprets code

## How to prepare your system for using Java

---

Before you can develop Java applications, the JDK must be installed on your system. In addition, your system may need to be configured to work with the JDK. Once you install the JDK, you'll be ready to create your first Java application.

### How to install the JDK

---

Figure 1-4 shows how to install the JDK. To start, you download the exe file for the setup program for the most recent version of the JDK from the Java web site. Then, you navigate to the directory that holds the exe file, run the setup file, and respond to the resulting dialog boxes.

Since Sun periodically updates the Java web site, we've kept the procedure shown in this figure somewhat general. As a result, you may have to do some searching to find the current version of the JDK. In general, you can start by looking at the downloads for Java SE. Then, you can find the most current version of the JDK for your operating system.

By the way, all of the examples in this book have been tested against version 6 of the JDK. Since Java has a good track record of being upwards compatible, however, these examples should work equally well with later versions of the JDK.

## The Java web site

`java.sun.com`

### How to download the JDK from the Java web site

1. Go to the Java web site.
2. Locate the download page for Java SE 6.
3. Click on the Download button for JDK 6 and follow the instructions.
4. Save the exe file for the setup program to your hard disk.

### How to install the JDK

- Run the exe file and respond to the resulting dialog boxes. When you're prompted for the JDK directory, use the default directory. For most Windows systems, the default directory is C:\Program Files\Java\jdk1.6.0 for Java SE 6.

### Notes

- For more information about installing the JDK, you can refer to the Java web site.
- If you are installing the Windows version of the JDK, you can perform either an offline or an online installation. An online installation is faster because only a small setup file is downloaded. However, you must remain online during the entire installation so the required files can be installed.

## A summary of the directories and files of the JDK

---

Figure 1-5 shows the directories and files that are created when you install the JDK. Here, the JDK is stored in the C:\Program Files\Java\jdk1.6.0 directory. This directory has multiple subdirectories, but the *bin*, *jre*, *lib*, and *docs* directories are the most important.

The *bin* directory holds all the tools necessary for developing and testing a program, including the Java compiler. Later in this chapter, you'll learn how to use these tools to compile and run Java applications. The *lib* directory contains libraries and support files required by the development tools.

The *jre* directory contains the Java interpreter, or *Java Runtime Environment (JRE)*, that's needed to run Java applications once they've been compiled. Although the JDK uses this internal version of the JRE, you can also download a standalone version of the JRE from the Java web site. Once you're done developing a Java application, for example, you can distribute the standalone JRE to other computers so they can run your application.

The *docs* directory can be used to store the Java documentation. Later in this chapter, you'll learn how to download and install this documentation.

In the JDK directory, you can find an HTML *readme* file that contains much of the information that's presented in this figure as well as more technical and detailed information about the JDK. You can view the HTML file with a web browser.

The JDK directory also contains the *src.zip* file. This is a compressed file that holds the source code for the JDK. If you want to view the source code, you can extract the source files from this zip file. If you're curious to see the Java code of the JDK, you may want to do that once you understand Java better.

When you work with Windows, you'll find that it sometimes uses the terms *folder* and *subfolder* to refer to directories and subdirectories. For consistency, though, we use the term *directory* throughout this book. In practice, these terms are often used interchangeably.



## The default directory for the JDK on a Windows machine

C:\Program Files\Java\jdk1.6.0\bin

## Four important subdirectories of the JDK

Directory	Description
bin	The Java development tools and commands.
jre	The root directory of the Java Runtime Environment (JRE).
lib	Additional libraries of code that are required by the development tools.
docs (optional)	The on-line documentation that you can download (see figure 1-15).

## Two important files stored in the JDK directory

File	Description
readme.html	An HTML page that provides information on Java SE, including system requirements, features, and documentation links.
src.zip	A zip file containing the source code for the Java SE API. If you use a zip tool such as WinZip to extract these directories and files, you can view the source code for the JDK.

## Description

- The *Java Runtime Environment (JRE)* is the Java interpreter that allows you to run compiled programs in Java. The jre directory is an internal copy of the runtime environment that works with the JDK. You can also download a standalone version of the JRE for computers that don't have the JDK installed on them.

Figure 1-5 A summary of the directories and files of the JDK

## How to set the command path

---

Figure 1-6 shows you how to configure Windows to make it easier to work with the JDK. If you're not using Windows, you can refer to the Java web site to see what you need to do to configure Java for your system.

To configure Windows to work with the JDK, you need to add the bin directory to the *command path*. That way, Windows will know where to look to find the Java commands that you use.

If you're using Windows 2000, NT, or XP, you can use the first procedure in this figure to set the command path. To start, you display the System Properties dialog box. One easy way to do that is to press the Windows key and the Pause / Break key at the same time. Then, you select the Advanced tab of this dialog box, and select the Environment Variables button. Finally, you use the Environment Variables dialog box to edit the system variable named Path.

When you find the system variable named Path in the Environment Variables dialog box, you can usually add the path for the Java bin directory to the end of the list of paths. To do that, you type a semicolon and the complete path (shaded at the top of this figure). However, if you've installed previous versions of Java on your system, you need to make sure that the path for JDK 1.6 is in front of the path for earlier versions.

If you're using Vista, you still use the Environment Variables dialog box to edit the system variable named Path. However, due to increased security, the procedure for displaying this dialog box is more elaborate when you're using Vista. To start, you can press the Windows key and the Pause / Break key at the same time to display the System section of the Control Panel. Then, you can select the Advanced System Settings link. When you do, Vista will prompt you with a User Account Control dialog box. If your user account has administrative permissions, you can proceed by selecting the Continue button. If not, you will be required to enter a password for one of the administrator accounts for the computer before you can continue. Either way, selecting the Continue button in this dialog box temporarily elevates your permissions to an administrator account when you change the Path variable and lowers your permissions to a normal user account after you're done.

Whenever you edit the command path, be careful! Since the command path may affect the operation of other programs on your PC, you don't want to delete or modify any of the other paths. You only want to add one directory to the command path. If that doesn't work, be sure that you're able to restore the command path to its original condition.

If you don't configure Windows in this way, you can still compile and run Java programs, but it's more difficult. For instance, instead of typing a command like this to compile a Java class

```
javac
```

you may need to type this command:

```
\Program Files\Java\jdk1.6.0\bin\javac
```

As you can see, then, setting the command path makes this much simpler. That's why it's so important to get this set right.

## A typical Path variable

```
%SystemRoot%;%SystemRoot%\system32;C:\Program Files\Java\jdk1.6.0\bin;
```

### How to set the path for Windows 2000/NT/XP

1. Display the System Properties dialog box. To do that, press the Windows key and the Pause / Break key at the same time. Or, right-click on the My Computer icon that's available from the desktop or the Start menu and select the Properties command.
2. In the System Properties dialog box, select the Advanced tab and click on the Environment Variables button.
3. Use the Environment Variables dialog box to edit the system variable named Path. If you haven't installed earlier versions of Java, type a semicolon and the path for the bin subdirectory of JDK 1.6 to the far right of the list of paths. Otherwise, add the 1.6 path followed by a semicolon before the paths for earlier JDK versions.

### How to set the path for Windows Vista

1. Display the System section of the Control Panel. To do that, display the Control Panel, select the System and Maintenance link, and select the System link. Or, press the Windows key and the Pause / Break key at the same time.
2. Select the Advanced System Settings link. When you do, Vista will prompt you with a User Account Control dialog box.
3. In the User Account Control dialog box, proceed by selecting the Continue button. If necessary, enter the password for an administrator account before selecting the Continue button.
4. Follow steps 2 and 3 from the Windows 2000/NT/XP procedure shown above.

### How to check the current path

- Start a Command Prompt or DOS Prompt as described in figure 1-12. Then, enter the Path command by typing the word path. This will display the current path statement.

### Description

- The *command path* on a Windows system tells Windows where to look for the commands that it is told to execute. When you use Java tools, you need to add the path for the jdk1.6.0\bin directory that's shown above.

### Notes

- For more information about setting the path for Windows or for information about configuring non-Windows operating systems, you can refer to the Java web site.

## How to set the class path

---

The *class path* is used to tell the operating system where to look for the compiled Java files (.class files) that are needed to run your program. When you run a program, the JRE needs to know where to find the all of class files needed to run the program and it uses the class path to search for these files.

By default, the class path for most systems includes the current directory, which is usually all you need. Then, before you run a program from a command prompt, you change the current directory to the directory that contains the .class files for the program. You'll learn more about how to use the command prompt later in this chapter.

The JRE also uses the class path when you run a program using some Java tools such as TextPad. In that case, the tool automatically changes the current directory to the directory that contains the class files for the program.

So as long as the class path for your system includes the current directory, the tool will work correctly. If the class path doesn't include the current directory, however, you'll need to add it before the tool will work correctly. Figure 1-7 describes how you do that.

To set the class path, you use the same techniques that you use for setting the command path (see figure 1-6), but you add a path to the Classpath variable instead of the Path variable. Specifically, you add a dot (.) for the current directory followed by a semicolon at the start of the list of paths. This is illustrated in the class path at the top of this figure.

To determine what the current class path is, you can enter *set* at the prompt. This runs the Set command, which displays a variety of settings, including the class path. This will also show you whether you've successfully added the current directory.

## A typical Classpath variable

```
.;c:\java\classes;
```

## When and how to modify the class path

- If your system doesn't have a Classpath variable, the default class path will allow you to run most of the programs described in this book. As a result, you won't need to modify the class path until you reach some of the later chapters.
- If your system has a Classpath variable that doesn't include the current directory, you need to add the current directory to the class path.
- To modify the class path, follow the procedure shown in figure 1-6 for modifying the command path, but modify the Classpath variable instead. To include the current directory in the class path, just code a period for the directory followed by a semicolon at the start of the list of paths as shown above.

## How to check the current class path

- Start a Command Prompt or DOS Prompt as described in figure 1-12, and enter the Set command by typing the word *set*. This will display a variety of settings including the current class path.

## Description

- The *class path* tells the JRE where to find the .class files needed to run a program.

## Note

- For more information about configuring the class path or for information about configuring non-Windows operating systems, you can refer to the Java web site.

## How to use TextPad to work with Java

---

Once the JDK is installed and configured for your operating system, you're ready to create your first application. Since most Java development is still done under Windows, this topic shows how to install and use a free trial version of TextPad, one of the most popular *text editors* that's designed for Java development.

Unfortunately, TextPad only runs under Windows. As a result, if you're using a non-Windows computer, you'll need to search the web to find a text editor for Java development that runs on the operating system that you're using. Fortunately, you can find a variety of these types of text editors on the web, and many of them are available for free (freeware) or for a small fee (shareware).

### How to install TextPad

---

Figure 1-8 shows how to download and install a free trial version of TextPad. Once you save the exe for the setup file to your hard disk, you simply run this file and respond to the resulting dialog boxes. Since this version of TextPad is a trial version, you should pay for TextPad if you decide to use it beyond the initial trial period. Fortunately, this program is relatively inexpensive (about \$30), especially when you consider how much time and effort it can save you.

## The TextPad web site

[www.textpad.com](http://www.textpad.com)

## How to download TextPad

1. Go to the TextPad web site.
2. Find the Free Trial version of TextPad and save the exe file for the setup program to your hard disk. This should take just a few minutes.

## How to install TextPad

- Run the exe file and respond to the resulting dialog boxes.

## Notes

- The trial version of TextPad is free, but if you like TextPad and continue to use it, you can pay the small fee of approximately \$30 to purchase it.
- The examples in this chapter were created with version 4.7 of TextPad.
- Although TextPad is a popular text editor for doing Java development under Windows, it doesn't run on Linux, Macintosh, or Solaris. As a result, if you're using a non-Windows operating system, you'll need to find a text editor for Java development that runs on your operating system. For example, VI and Emacs are two popular text editors for Linux.

## How to use TextPad to save and edit source code

---

Figure 1-9 shows how to use TextPad to save and edit source code. In short, you can use the standard Windows shortcut keystrokes and menus to enter, edit, and save your code. You can use the File menu to open and close files. You can use the Edit menu to cut, copy, and paste text. And you can use the Search menu to find and replace text. In addition, TextPad color codes the code in the source files so it's easier to recognize the Java syntax.

To edit as efficiently as possible, you can use the View menu to set the editing options for a single file. And you can use the Configure→Preferences command to set the options for all files. In particular, you may want to turn the Line Number option on, and you may want to set the tab settings so you can easily align the code in an application.

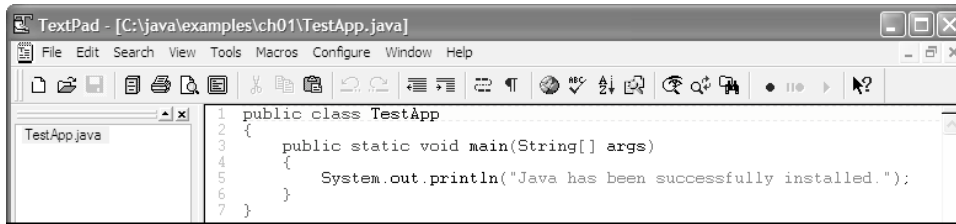
When you save Java source code, you must give the file the same name as the class name. Since Java is a *case-sensitive* language, you must also use the same capitalization in the file and class names. If you don't, you'll get an error message when you try to compile the code. In this figure, for example, you can see that "TestApp" is used for both the class name and the file name.

You must also save a Java source file with the four-letter *java* extension. When you use TextPad to save your source code, you can use the Save As Type drop-down list in the Save As dialog box to apply this extension. However, if you're using a text editor that isn't designed for working with Java, you may need to add the *java* extension to the end of the file name and enclose the file name in quotation marks like this: "TextApp.java". Otherwise, the text editor may truncate the extension to *jav* or change the capitalization in the file name. This will lead to errors when you try to compile the source code.

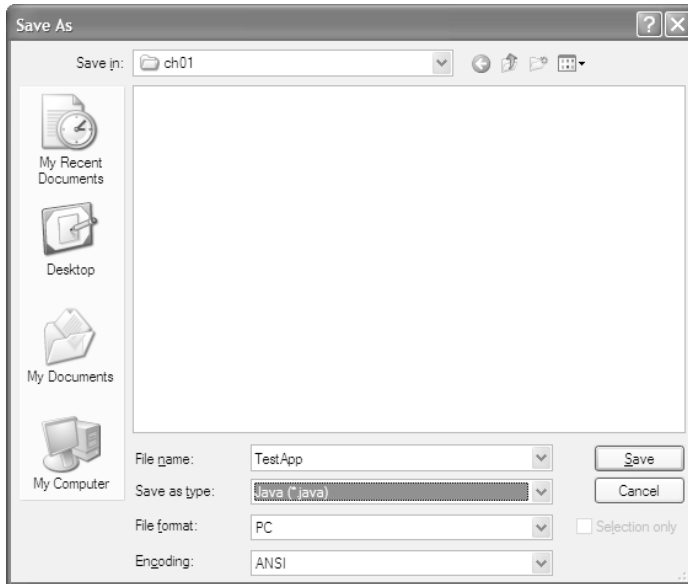
You must also save Java source code in a standard text-only format such as the *ASCII format* or the *ANSI format*. By default, TextPad saves code in the ANSI format and that's usually what you want. If, however, you need to save a file in another format such as Unicode, TextPad can do that too.



## The TextPad text editor with source code in it



## TextPad's Save As dialog box



## How to enter, edit, and save source code

- To enter and edit source code, you can use the same techniques that you use for working with any other Windows text editor.
- To save the source code, select the Save command from the File menu (Ctrl+S). Then, enter the file name so it's exactly the same as the class name, and select the Java option from the Save As Type list so TextPad adds the four-letter java extension to the file name.

## How to display line numbers and set options for one source file

- To display the line numbers for the source code, select View→Line Numbers.
- To set formatting options like tab settings, select View→Document Properties.

## How to display line numbers and set the options for all source files

- Select Configure→Preferences. Then, click on the type of default that you want to set. For instance, to display line numbers for all files, click on View and click on Line Numbers. To set the tab stops, click on Document Classes, Java, and Tabulation.

Figure 1-9 How to use TextPad to save and edit source code

## How to use TextPad to compile source code

---

Figure 1-10 shows how to use TextPad to compile the source code for a Java application. The quickest way to do that is to press Ctrl+1 to execute the Compile Java command of the Tools menu. If the source code compiles cleanly, TextPad will generate a Command Results window and return you to the original source code window.

However, if the source code doesn't compile cleanly, TextPad will leave you at a Command Results window like the one shown in this figure. In this case, you can read the error message, switch to the source code window, correct the error, and compile the source code again. Since each error message identifies the line number of the error, you can make it easier to find the error by selecting the Line Number option from the View menu. That way, TextPad will display line numbers as shown in this figure.

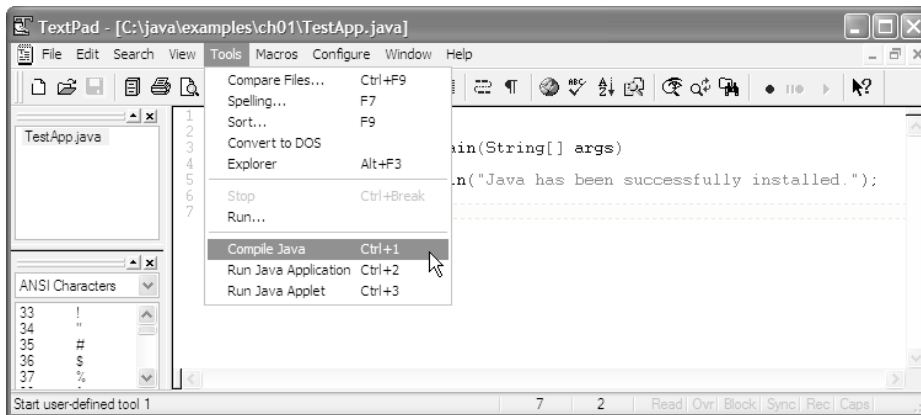
When you have several Java files open at once, you can use the Document Selector pane to switch between files. In this figure, only two documents are open (TestApp and Command Results), but you can open as many files as you like. You can also use the Window menu and standard Windows keystrokes to switch between windows.

## How to use TextPad to run an application

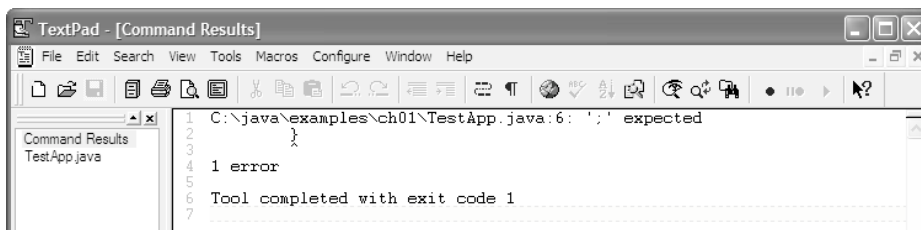
---

Once you've compiled the source code for an application, you can run that application by pressing Ctrl+2. In this figure, the application prints text to the *console*. As a result, TextPad starts a console window like the one shown in this figure. Then, you can usually press any key to end the application and close the window. Sometimes, however, you may need to click on the Close button in the upper right corner of the window to close it.

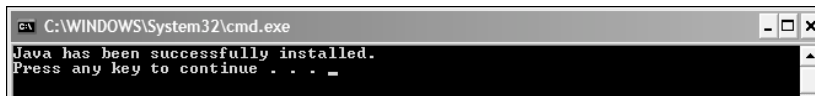
## The Tools menu



## A compile-time error



## Text printed to the console



## How to compile and run an application

- To compile the current source code, press Ctrl+1 or select Tools→Compile Java.
- To run the current application, press Ctrl+2 or select Tools→Run Java Application.
- If you encounter compile-time errors, TextPad will print them to a window named Command Results.
- To switch between the Command Results window and the source code window, press Ctrl+F6, press Ctrl+Tab, or double-click on the window that you want in the Document Selector pane that's on the left side of the TextPad window.
- When you print to the *console*, a DOS window like the one above is displayed. To close the window, press any key or click the Close button in the upper right corner.

Figure 1-10 How to use TextPad to compile and run an application

## Common error messages and solutions

---

Figure 1-11 summarizes some common error messages that you may encounter when you try to compile or run a Java application. The first two errors illustrate *compile-time errors*. These are errors that occur when the Java compiler attempts to compile the program. In contrast, the third error illustrates a *runtime error*. That is an error that occurs when the Java interpreter attempts to run the program but can't do it.

The first error message in this figure involves a syntax error. When the compiler encounters a syntax error, it prints two lines for each error. The first line prints the name of the .java file, followed by a colon, the line number for the error, another colon, and a brief description of the error. The second line prints the code that caused the error, including a caret character that tries to identify the location where the syntax error occurred. In this example, the caret points to the right brace, but the problem is that the previous line didn't end with a semicolon.

The second error message in this figure involves a problem defining the *public class* for the file. The compiler displays an error message like this when the file name for the .java file doesn't match the name of the public class defined in the source code. For example, the TextApp.java file must contain this code

```
public class TestApp{
```

If the name of the file doesn't match the name of the public class (including capitalization), the compiler will give you an error like the one shown in this figure. You'll learn more about the syntax for defining a public class in the next chapter.

The third error message in this figure occurs if you try to run an application that doesn't have a main method. You'll learn how to code a main method in the next chapter. For now, just realize that every application must contain a main method. To correct the error shown in this figure, then, you must add a main method to the class or run another class that has a main method.

Most of the time, the information displayed by an error message will give you an idea of how to fix the problem. Sometimes, though, the compiler gets confused and doesn't give you accurate error messages. In that case, you'll need to double-check all of your code. You'll learn more about debugging error messages like these as you progress through this book.

## A common compile-time error message and solution

Error: `C:\java\examples\ch01\TestApp.java:6: ';' expected`  
 `}`  
 `^`

Description: The first line in this error message displays the file name of the .java file, a number indicating the line where the error occurred, and a brief description of the error. The second line displays the line of code that may have caused the error with a caret symbol (^) below the location where there may be improper syntax.

Solution: Edit the source code to correct the problem and compile again.

## Another common compile-time error message and solution

Error: `C:\java\examples\ch01\TestApp.java:1: class testapp is public, should be declared in a file named testapp.java`  
`public class testapp`  
 `^`

Description: The .java file name doesn't match the name of the public class. You must save the file with the same name as the name that's coded after the words "public class". In addition, you must add the java extension to the file name.

Solution: Edit the class name so it matches the file name (including capitalization), or change the file name so it matches the class name. Then, compile again.

## A common runtime error message and solution

Error: `Exception in thread "main" java.lang.NoSuchMethodError: main`

Description: The class doesn't contain a main method.

Solution: Run a different class that does have a main method, or enter a main method for the current class.

## Description

- When you compile an application that has some statements that aren't coded correctly, the compiler cancels the compilation and displays messages that describe the *compile-time errors*. Then, you can fix the causes of these errors and compile again.
- When the application compiles without errors (a "clean compile") and you run the application, a *runtime error* occurs when the Java Virtual Machine can't execute a compiled statement. Then, the application is cancelled and an error message is displayed.

## How to use the command prompt to work with Java

---

If you're using TextPad, you can use its commands to compile and run most of your Java applications. Even so, there may be times when you will need to compile and run Java applications from the *command prompt*. And if you're using another text editor that doesn't provide compile and run commands, you can use the command prompt to compile and run all of your Java applications.

### How to compile source code

---

Figure 1-12 shows how to use a command prompt to compile and run applications. In particular, this figure shows how to use the Windows command prompt, sometimes called the *DOS prompt*.

To start, you enter the change directory (`cd`) command to change the current directory to the directory that holds the application. In this figure, for example, you can see that the directory has been changed to `c:\java\examples\ch01` because that's the directory that contains the `TestApp.java` file. Then, to compile the application, you use the *javac command* to start the Java compiler. When you enter the `javac` command, you follow it by a space and the complete name of the `.java` file that you want to compile. Since Java is case-sensitive, you need to use the same capitalization that you used when you saved the `.java` file.

If the application doesn't compile successfully, you can use your text editor to correct and resave the `.java` file, and you can compile the program again. Since this means that you'll be switching back and forth between the text editor and the command prompt, you'll want to leave both windows open.

When you compile an application successfully, the Java compiler will create a `.class` file that has the same file name as the `.java` file. For example, a successful compilation of the `TestApp.java` file will create the `TestApp.class` file. This `.class` file is the file that contains the Java bytecodes that can be run by the Java interpreter.

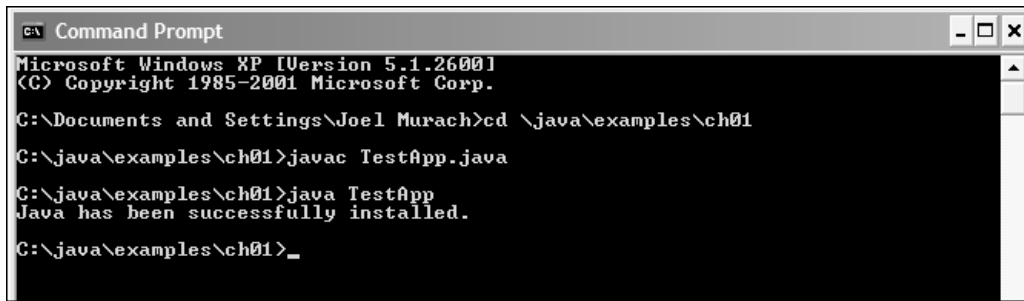
### How to run an application

---

To run a program from the command prompt, you use the *java command* to start the Java interpreter. Although you need to use the proper capitalization when you use the `java` command, you don't need to include an extension for the file. When you enter the `java` command correctly, the Java interpreter will run the `.class` file for the application.

Running a Java program often displays a graphical user interface like the one shown in figure 1-2. However, you can also print information to the console and get input from the console. For example, the `TestApp` file in this figure prints a single line of text to the console.

## The commands for compiling and running an application



```
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\Joel Murach>cd \java\examples\ch01
C:\java\examples\ch01>javac TestApp.java
C:\java\examples\ch01>java TestApp
Java has been successfully installed.
C:\java\examples\ch01>_
```

### Syntax to compile an application

```
javac ProgramName.java
```

### Syntax to run an application

```
java ProgramName
```

### Description

- The *command prompt* is the prompt that indicates that the operating system is waiting for the next command. This prompt usually shows the current directory, and it always ends with `>`.
- In Windows, the command prompt is sometimes referred to as the *DOS prompt* or the *DOS window*. You can enter DOS commands at the DOS prompt.

### Operation

- To open a command prompt in Windows, click on the Start button, find MS-DOS Prompt or Command Prompt, and select it. If its location isn't obvious, try looking in Accessories.
- To change the current directory to the directory that contains the file with your source code, use the change directory command (`cd`) as shown above.
- To compile the source code, enter the Java compile command (`javac`), followed by the file name (including the java extension). Since this is a case-sensitive command, make sure to use the same capitalization that you used when naming the file.
- If the code compiles successfully, the compiler generates another file with the same name, but with `class` as the extension. This file contains the bytecodes.
- If the code doesn't compile successfully, the java compiler generates error messages for the compile-time errors. Then, you must switch back to your text editor, fix the errors, save your changes, and compile the program again.
- To run the compiled version of your source code, enter the Java command (`java`), followed by the program name (without any extension). Since this is a case-sensitive command, make sure to use the same capitalization that you used when naming the file.

### Note

- The code shown in the command prompt above will only work if the `bin` subdirectory of the JDK 6 directory has been added to the command path as shown in figure 1-6.

Figure 1-12 How to use the command prompt to compile and run an application

## How to compile source code with a switch

---

Most of the time, you can use TextPad's compile command or the plain `javac` command shown in the last figure to compile Java source code. Sometimes, however, you need to supply a *switch* to use certain features of the `javac` command as summarized in figure 1-13.

Note in the syntax summary in this figure that the brackets around the switch name indicate that it's optional. In addition, the ellipsis (...) indicates that you can code as many switches as you need. You'll see this notation used throughout this book.

When new versions of Java become available, the designers of Java mark some older features as *deprecated*. When a feature is deprecated, it means that its use is not recommended and it may not be supported in future versions of Java. As a result, you should try to avoid using deprecated features whenever possible.

The first example in this figure shows how to use the deprecation switch to get more information about code that uses deprecated features of the JDK. As you can see, if you don't use this switch, you'll get a brief message that indicates that your code uses deprecated features. On the other hand, if you use this switch, you'll get detailed information about the code that uses deprecated features including the line number for that code. That makes it easier to modify your code so it doesn't use these features.

The second example shows how to use the source switch to compile code so it can run on another computer that's using an older Java interpreter. In particular, this example shows how to compile code so it can run on a computer with version 1.4 of the JRE. When you use this switch, though, you won't be able to use the new features of Java 5 or 6. As a result, you shouldn't use this switch if want to use the new Java features that are presented in this book.

This figure also shows how to use a switch with TextPad. To do that, you begin by selecting the **Configure→Preferences** command, which displays the Preferences dialog box. Then, you expand the Tools group and select **Compile Java**, which displays the options for compiling Java. At that point, you can add a switch by typing it at the end of the parameters in the Parameters text box. In this figure, for example, the deprecation switch has been added to the end of the Parameters text box. As a result, the deprecation feature will be on for future TextPad sessions until you remove it from this dialog box.

Keep in mind, though, that you don't need to use either of the two switches shown in this figure as you use this book. Since the book is designed to teach you Java 6, you shouldn't need the source switch to specify the use of an earlier version. Since this book doesn't teach the deprecated features, you shouldn't need more information about them. However, you may occasionally need to use one or both of these switches.





## Essential DOS skills for working with Java

---

Figure 1-14 summarizes some of the most useful commands and keystrokes for working with DOS. In addition, it shows how to install and use a DOS program called DOSKey, which makes entering and editing DOS commands easier. If you're going to use DOS to work with Java, you should review these DOS commands and keystrokes, and you will probably want to turn on the DOSKey program if it isn't already on. If you aren't going to use DOS, of course, you can skip this figure.

At the top of this figure, you can see a DOS window that shows two DOS commands and a directory listing. Here, the first command changes the current directory to `c:\java\exercises\ch01`. The next command displays a directory listing. If you study this listing, you can see that this directory contains two files with one line of information for each file. At the right side of each line, you can see the complete file names for these two files (`TestApp.class` and `TestApp.java`), and you can see the capitalization for these files as well.

If you master the DOS commands summarized in this figure, you should be able to use DOS to work with Java. To switch to another drive, type the letter of the drive followed by a colon. To change the current directory to another directory, use the `cd` command. To display a directory listing for the current directory, use the `dir` command. Although DOS provides many more commands that let you create directories, move files, copy files, and rename files, you can also use the Windows Explorer to perform those types of tasks.

Although you don't need to use the DOSKey program, it can save you a lot of typing and frustration. If, for example, you compile a program and you encounter a syntax error, you will need to use a text editor to fix the error in the source code. Then, you will need to compile the program again. If you're using DOSKey, you can do that by pressing the up-arrow key to display the last command that was executed and then pressing the Enter key to execute the command. And if you make a mistake when entering a command, you can use the left- and right-arrow keys to edit the command instead of having to enter the entire command again.

## A directory listing

```

Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\Joel Murach>cd \java\exercises\ch01

C:\java\exercises\ch01>dir
Volume in drive C has no label.
Volume Serial Number is A0E5-29E9

Directory of C:\java\exercises\ch01

07/08/2004  05:11 PM    <DIR>          .
07/08/2004  05:11 PM    <DIR>          ..
07/08/2004  05:12 PM                445 TestApp.class
07/08/2004  04:09 PM                139 TestApp.java
                2 File(s)                584 bytes
                2 Dir(s)  21,394,849,792 bytes free

C:\java\exercises\ch01>_

```

## A review of DOS commands and keystrokes

Command	Description
dir	Displays a directory listing.
dir /p	Displays a directory listing that pauses if the listing is too long to fit on one screen.
cd \	Changes the current directory to the root directory for the current drive.
cd ..	Changes the current directory to the parent directory.
cd <i>directory name</i>	Changes the current directory to the subdirectory with the specified name.
<i>letter:</i>	Changes the current drive to the drive specified by the letter. For example, entering d: changes the current drive to the d drive.

## How to start the DOSKey program

- To start the DOSKey program, enter “doskey /insert” at the command prompt.
- To automatically start the DOSKey program for all future sessions, add the “doskey/insert” statement after the “path” statement in the autoexec.bat file. For help on editing the autoexec.bat file, see the second procedure in figure 1-6.

## How to use the DOSKey program

Key	Description
Up or down arrow	Cycles through previous DOS commands in the current session.
Left or right arrow	Moves cursor to allow normal editing of the text on the command prompt.

Figure 1-14 Essential DOS skills for working with Java

## How to use the documentation for the Java SE API

---

When you write Java code, you'll often need to look up information about the *Application Program Interface (API)* for the Java SE. The API provides the Java classes that you can use as you build your Java applications. Since Sun provides HTML-based documentation for the Java SE API, you can browse this documentation with any web browser to get the detailed information you need about any Java class.

## How to install the API documentation

---

Although you can use your browser to view the documentation for the Java SE API from the Java web site, you will want to install this documentation on your hard drive instead. That way, you can browse the documentation more quickly, and you can browse it even if you aren't connected to the Internet.

To download and install this documentation, you can use the procedure shown in figure 1-15. Since the documentation comes in a compressed format called a *zip file*, you need to use an unzip tool to extract the HTML pages from the zip file. When you use this tool, it creates a docs directory that contains many files and subdirectories. Although you can store the docs directory anywhere you like, it's common to store it as a subdirectory of the JDK directory.

If your operating system doesn't include a tool for working with zip files, you can download one from the web. For example, WinZip is a popular program for working with zip files. You can download a free evaluation copy from [www.winzip.com](http://www.winzip.com).

## How to download the API documentation

1. Go to the Java web site ([java.sun.com](http://java.sun.com)).
2. Go to the download page for the version of the JDK that you're using (see figure 1-4) and find the hyperlink for the Java SE 6 documentation download.
3. Follow the instructions for the download and save the zip file to your hard drive.

## How to install the API documentation

- Extract all files in the zip file to your hard drive. This will create a directory named docs that contains several files and subdirectories.
- Although it's common to store the API documentation in the JDK directory (usually C:\Program Files\Java\jdk1.6.0 for version 6), you can store it anywhere you like.

## Description

- The *Application Programming Interface*, or *API*, provides all the classes that are included as part of the JDK. To learn about these classes, you can use the API documentation.
- You can view the API documentation from the Java web site, or you can download and install it on your system.

## Notes

- If you're using an older operating system that doesn't automatically work with zip files, you may need to use a tool such as WinZip to extract the files from the zip file. To download a free evaluation copy of WinZip, go to [www.winzip.com](http://www.winzip.com).

## How to navigate the API documentation

---

Figure 1-16 shows how to navigate the documentation for the Java SE API. To start, you point your web browser to the index page that's stored in the docs\api directory. To do that for the first time, use the Windows Explorer to navigate to the docs\api directory and double-click on the index.html file, or enter the location of this page into the address area of your web browser:

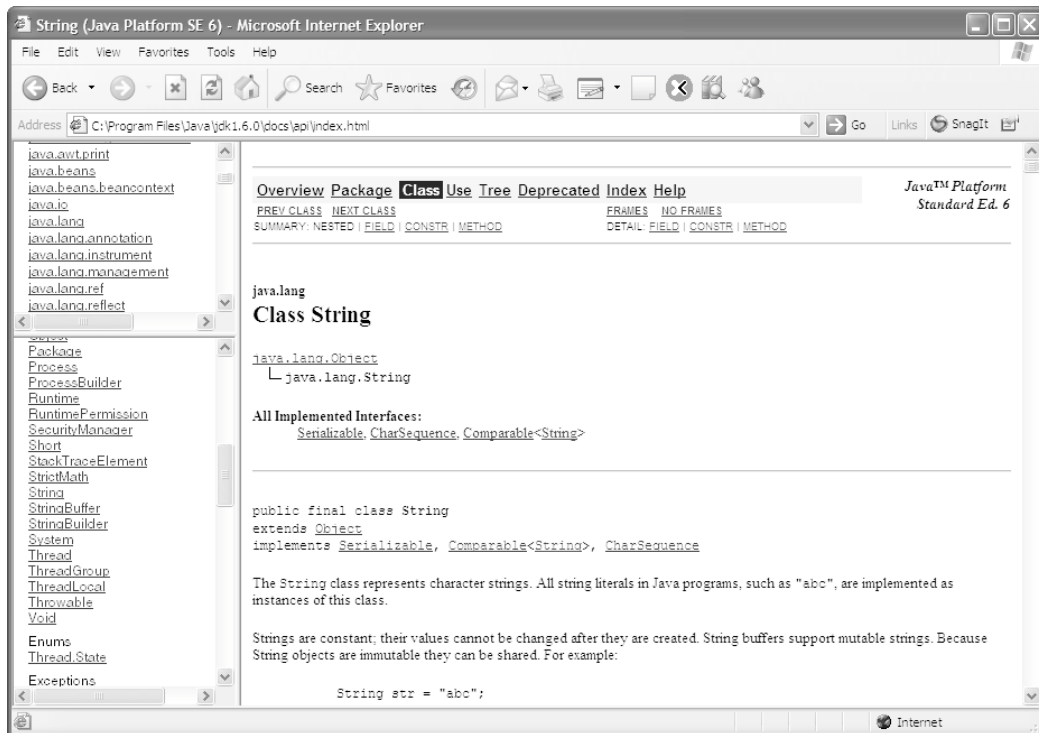
```
C:\Program Files\Java\jdk1.6.0\docs\api\index.html
```

Since you'll need to access this page often as you learn about Java, you should use your web browser's favorites or bookmark feature to mark this page. Then, the next time you need to use this page, you can go right to it.

To browse the Java documentation, you need to know that all the classes in the Java API are organized into *packages*. As a result, the index page for the documentation provides a way to navigate through packages and classes. To start, you can select a package from the upper left frame. When you do, the lower left frame displays all the classes in that package. Then, you can select a class to display information about that class in the right frame.

As you progress through this book, you'll learn about many different packages and classes, and you'll learn much more about how packages and classes work. Along the way, you can always use the documentation for the Java API to help clarify the discussion or further your knowledge.

## The index for the documentation



## Description

- If you've installed the API documentation on your hard drive, you can display an index like the one shown above by using your web browser to go to the `index.html` file in the `docs\api` directory. If you haven't installed the documentation, you can browse through it on the Java web site.
- Related classes in the Java API are organized into *packages*, which are listed in the upper left frame of the documentation page. When you select a package, all the classes for that package are listed in the lower left frame.
- You can select a class from the lower left frame to display the documentation for that class in the right frame. You can also select the name of the package at the top of the lower left frame to display information about the package and the classes it contains.
- Once you display the documentation for a class, you can scroll through it or click on a hyperlink to get more information.
- The documentation for a class usually provides a wide range of information, including a summary of all of its methods. You'll learn more about methods and how they're used throughout this book.
- To make it easier to access the API documentation, you should bookmark the index page. To do that with the Internet Explorer, select the Add To Favorites command from the Favorites menu and accept the default name for the page or assign your own name to it. Then, you can redisplay this page later by selecting it from the Favorites menu.

Figure 1-16 How to navigate the API documentation

## Introduction to Java IDEs

---

Dozens of *Integrated Development Environments (IDEs)* are available for working with Java. A typical IDE provides not only a text editor, but also tools for compiling, running, and debugging code as well as a tool for building graphical user interfaces. Eclipse and NetBeans are two of the most popular professional IDEs for working with Java. And BlueJ is an IDE that's popular for beginning Java students.

Although you've already learned how to develop Java applications with TextPad, we realize that you may prefer to use an IDE, especially if you have previous experience with one. That's why we developed free tutorials that show how to use this book with popular IDEs like Eclipse, NetBeans, and BlueJ. Each tutorial shows how to download and install a specific IDE, how use it to work with the applications presented in this book, and how to use it to debug an application. To download one of these *free* tutorials, please visit our web site at:

`www.murach.com/books/jse6/ides.htm`

## The Eclipse IDE for Java

---

Eclipse is a software framework for developing IDEs. Eclipse is open-source, available for free from [www.eclipse.org](http://www.eclipse.org), runs on all modern operating systems, and includes a popular Java IDE.

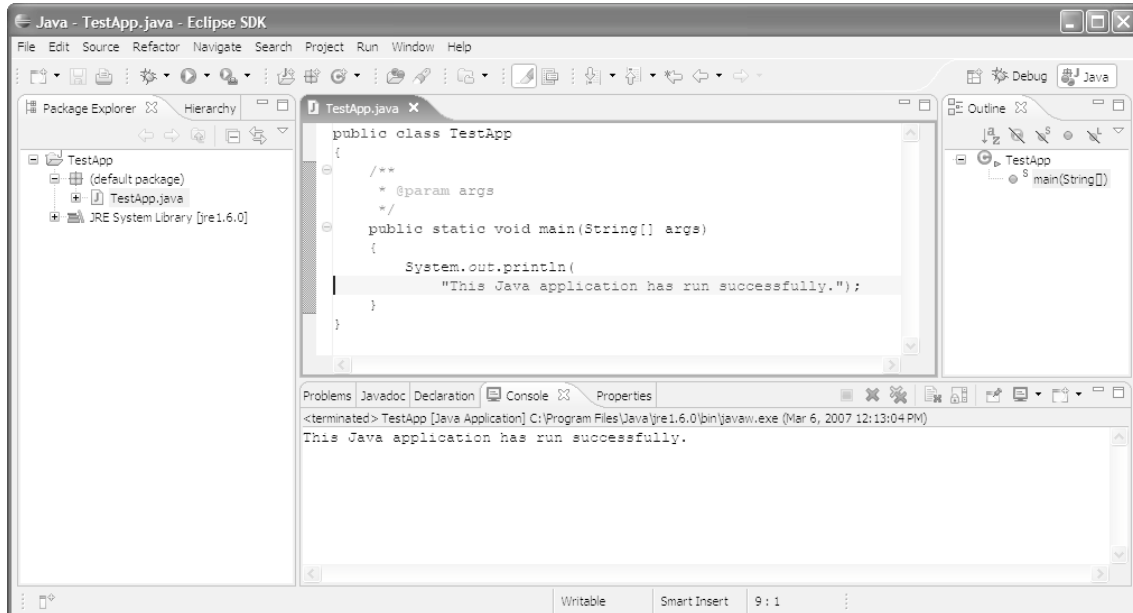
The first screen in figure 1-17 shows the window you use to edit Java code using Eclipse. As you work with this code editor, Eclipse can help you complete your code and notify you of potential compile-time errors. In addition, Eclipse will automatically compile your programs when you run them. And when you run a program that prints text to the console, Eclipse displays that text in its own Console window.

The second screen in this figure shows the windows that are displayed when you use Eclipse to debug a Java application. Here, you can use the code editor to set breakpoints, and you can use the buttons in the toolbar to step through your code. As you do, you can use the Variables window to view the values of all of the active variables. If you've used any modern debugging tool, you should understand how this works.

In addition, Eclipse provides many other advanced features that aren't available from a simple tool like TextPad. For example, you can use Eclipse with JUnit, which helps you test your applications, and also with Ant, which helps you document, package, and deploy your applications.



## The Eclipse code editor



## The Eclipse debugger

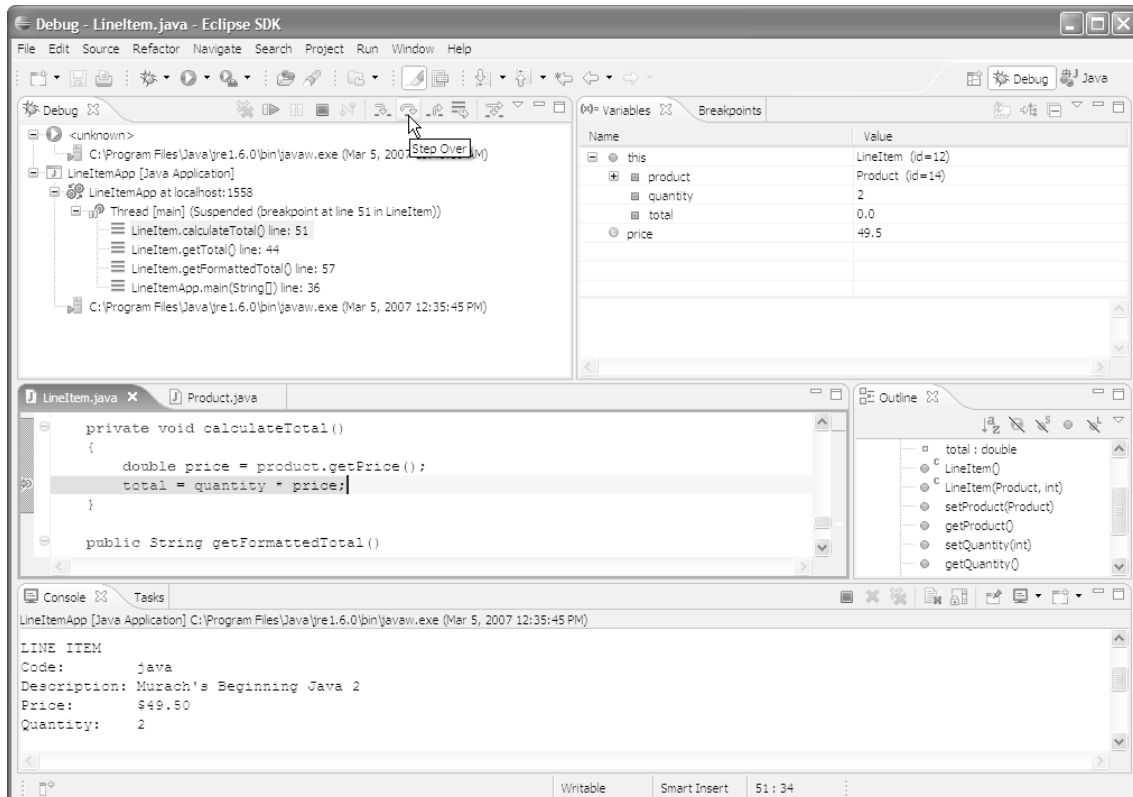


Figure 1-17 Two views of the Eclipse IDE for Java

## The NetBeans IDE

---

NetBeans is open-source, available for free from [www.netbeans.org](http://www.netbeans.org), runs on all modern operating systems, and includes a popular GUI builder that can be used to develop GUIs for Java applications.

The first screen in figure 1-18 shows the window you use to edit code using the NetBeans IDE. This code editor provides many of the same features that are available from the code editor that's available from Eclipse. Like Eclipse, NetBeans will automatically compile your programs when you run them. Also, when you run a program that prints text to the console, NetBeans displays that text in its Output window along with some other text for the application.

The second screen in this figure shows the popular GUI builder that's available from NetBeans. This GUI builder is sometimes referred to as the Matisse GUI builder, and many programmers consider it to be the most intuitive and easy-to-use tool for building GUIs for Java applications. To use it, you can drag Swing controls from the Palette window onto a form in Design view. Then, you can use Design view to align the controls, and you can use the Properties window to set the properties for the controls. As you do this, the code that displays the IDE is automatically generated. Finally, you can generate event handlers for the events that you want to handle and use Source view to write the code that handles these events.

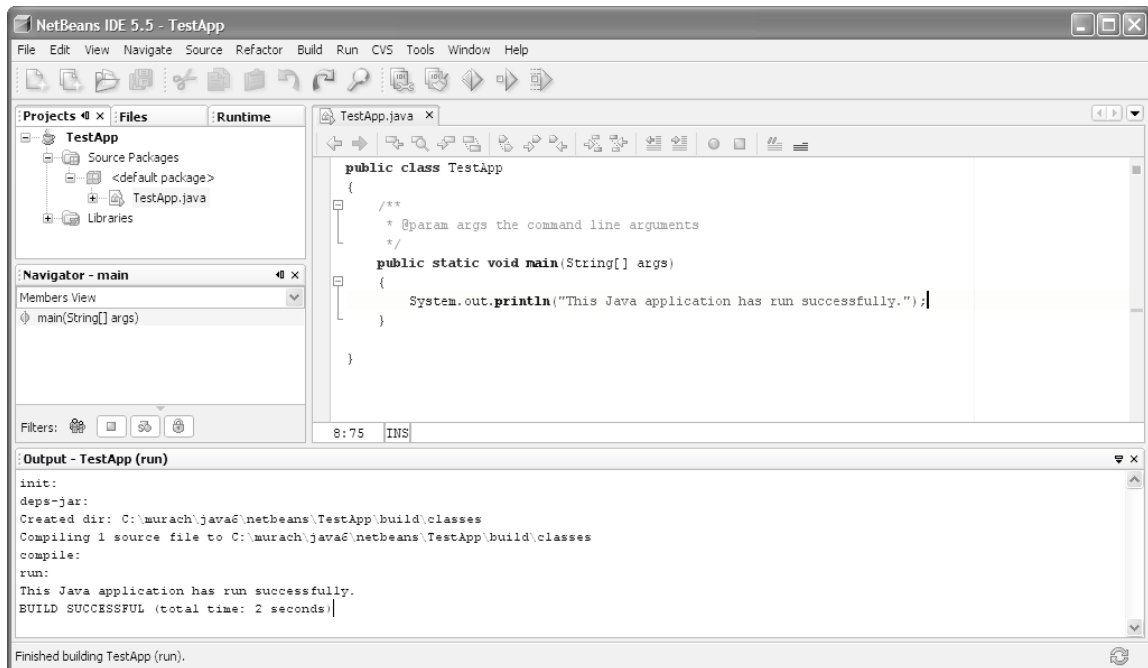
Like Eclipse, NetBeans also provides a professional debugger and other advanced features that aren't available from a simple tool like TextPad. For example, NetBeans provides support for the JUnit and Ant tools.

## The BlueJ IDE

---

BlueJ is a free IDE that was developed as part of a university research project to teach object-oriented programming to first-year Java students. BlueJ is available for free, and you can download it from [www.bluej.org](http://www.bluej.org). Although BlueJ doesn't provide as many features as IDEs like Eclipse and NetBeans, it provides some of the advantages of an IDE without some of the drawbacks. For example, BlueJ includes a debugger that's easy for students to learn. In addition, BlueJ maintains a diagram for each project that can help students visualize how the classes in a project work together. As a result, if you're new to object-oriented programming, you might want to try using BlueJ.

## The NetBeans code editor



## The NetBeans GUI builder

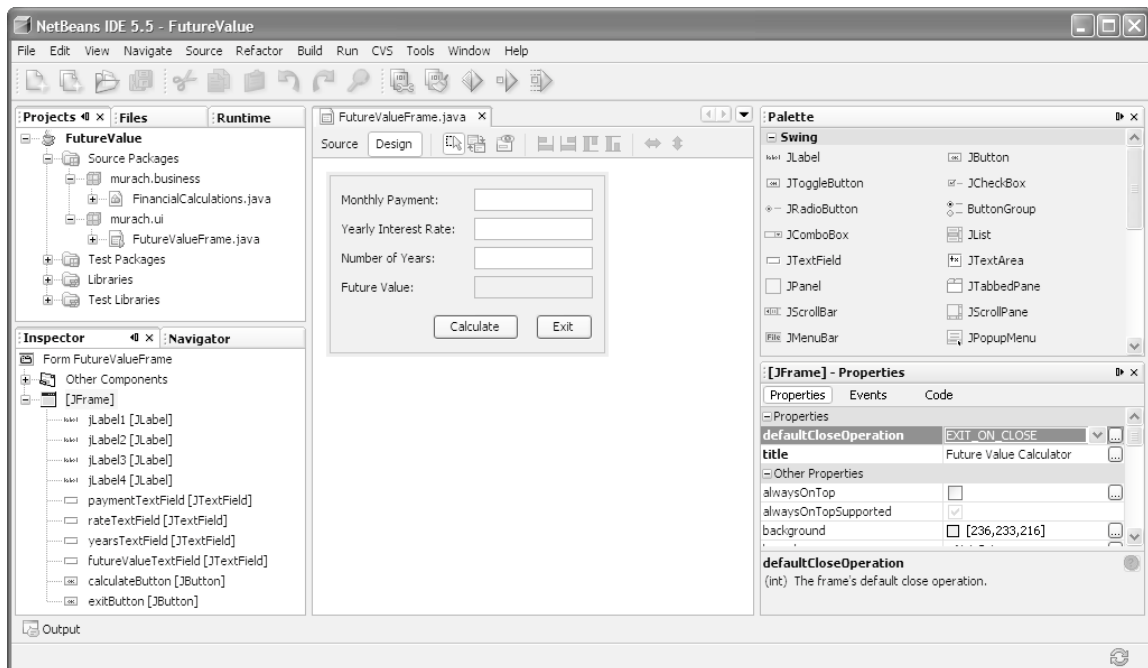


Figure 1-18 Two views of the NetBeans IDE

## Perspective

---

In this chapter, you learned how to install and configure the JDK for developing Java programs. You learned how to use TextPad to enter, edit, compile, and run a program. You learned how to use the command prompt to compile and run programs. And you learned how to install and view the API documentation for the JDK. With that as background, you're ready to learn how to write your own Java applications.

## Summary

---

- You use the *Java Development Kit (JDK)* to develop Java applications. This used to be called the *Software Development Kit (SDK)* for Java. As of version 6, the *Standard Edition (SE)* of Java is called *Java SE*. In older versions, it was called the *Java 2 Platform, Standard Edition (J2SE)*.
- You can use Java SE to create *applications* and a special type of Internet-based application known as an *applet*. In addition, you can use the *Enterprise Edition (EE)* of Java, which is known as *Java EE*, to create server-side applications using *servlets* and *JavaServer Pages (JSPs)*.
- The *Java compiler* translates *source code* into a *platform-independent* format known as *Java bytecodes*. Then, the *Java interpreter*, or *Java Runtime Environment (JRE)*, translates the bytecodes into instructions that can be run by a specific operating system. A Java interpreter is an implementation of a *Java virtual machine (JVM)*.
- When you use the JDK with Windows, you should add the bin directory (usually C:\Program Files\Java\jdk1.6.0\bin) to the *command path* and you should add the current directory to the *classpath*.
- A *text editor* that's designed for working with Java provides features that make it easier to enter, edit, and save Java code.
- Some text editors such as TextPad include commands for compiling and running Java applications. You can also use the *command prompt* to enter the commands for compiling and running an application.
- When you compile a program, you may get *compile-time errors*. When you run a program, you may get *runtime errors*.
- To compile code from the command prompt, you use the *javac command* to start the Java compiler. To run an application from the command prompt, you use the *java command* to start the Java interpreter.
- You can get detailed information about any class in the Java SE API by using a web browser to browse the HTML-based documentation for its *Application Programming Interface (API)*.
- An *Integrated Development Environment (IDE)* like Eclipse, NetBeans, or BlueJ can make working with Java easier.

## Before you do the exercises for this chapter

Before you do any of the exercises in this book, you need to download the folders and files for this book from our web site ([www.murach.com](http://www.murach.com)) and install them on your system. For complete instructions, please refer to appendix A. Then, you should follow the procedures shown in this chapter to install and configure the JDK (figure 1-4 through 1-7), TextPad (figure 1-8) or an equivalent text editor, and the documentation for the Java API (figure 1-15).

## Exercise 1-1 Use TextPad to develop an application

This exercise will guide you through the process of using TextPad to enter, save, compile, and run a simple application.

### Enter and save the source code

1. Start TextPad by clicking on the Start button and selecting Programs or All Programs→TextPad.
2. Enter this code (type carefully and use the same capitalization):

```
public class TestApp
{
    public static void main(String[] args)
    {
        System.out.println(
            "This Java application has run successfully");
    }
}
```

3. Use the Save command in the File menu to display the Save As dialog box. Next, navigate to the c:\java1.6\ch01 directory and enter TestApp in the File name box. If necessary, select the Java option from the Save as Type combo box. Then, click on the Save button to save the file.

### Compile the source code and run the application

4. Press Ctrl+1 to compile the source code. If you get an error message, read the error message, edit the text file, save your changes, and compile the application again. Repeat this process until you get a clean compile (the code is displayed with no error messages).
5. Press Ctrl+2 to run the application. This application should display a console window that says “This Java application has run successfully” followed by a line that reads “Press any key to continue...”.
6. Press any key. This should close the console window. If it doesn’t, click on the Close button in the upper right corner of the window to close it.

### Introduce and correct a compile-time error

7. In the TextPad window, delete the semicolon at the end of the System.out.println statement. Then, press Ctrl+1 to compile the source code. TextPad should display an error message that indicates that the semicolon is missing in the Command Results window.

8. In the Document Selector pane, click on the TestApp.java file to switch back to the source code, and press Ctrl+F6 twice to toggle back and forth between the Command Result window and the source code. Then, select View→Line Numbers to display the line numbers for the source code lines.
9. Correct the error and compile the file again (this automatically saves your changes). This time the file should compile cleanly, so you can run it again and make sure that it works correctly.
10. Select Configure→Preferences, click on View, and check Line Numbers. That will add line numbers to the source statements in all your applications. If you want to look through the other options and set any of them, do that now. When you're done, close the file and exit TextPad.

## Exercise 1-2 Use any Java development tool to develop an application

If you aren't going to use TextPad to develop your Java programs, you can try whatever tools you are going to use with this generic exercise.

### Use any text editor to enter and save the source code

1. Start the text editor or IDE and enter this code (type carefully and use the same capitalization):

```
public class TestApp
{
    public static void main(String[] args)
    {
        System.out.println(
            "This Java application has run successfully");
    }
}
```

2. Save this code in a file named "TestApp.java".

### Compile the source code and run the application

3. Compile the source code. If you're using a text editor or IDE that has a compile command, use this command. (Note that some IDEs automatically compile the source code when you run the application). Otherwise, use your command prompt to compile the source code. To do that, start your command prompt and use the cd command to change to the directory that contains the source code. Then, enter the javac command like this (make sure to use the same capitalization):

```
javac TestApp.java
```

4. Run the application. If you're using a text editor or IDE that has a run or execute command, use this command. Otherwise, use your command prompt to run the application. To do that, enter the java command like this (make sure to use the same capitalization):

```
java TestApp
```

5. When you enter this command, the application should print "This Java application has run successfully" to the console window.

### Exercise 1-3 Use the command prompt to run any compiled application

This exercise shows how to use the command prompt to run any Java application.

1. Open the command prompt window. Then, change the current directory to `c:\java1.6\ch01`.
2. Use the `java` command to run the `LoanCalculatorApp` application. This application calculates the monthly payment for a loan amount at the interest rate and number of years that you specify. This shows how the JRE can run any application whether or not it has been compiled on that machine. When you're done, close the application to return to the command prompt.

### Exercise 1-4 Navigate the API documentation

This exercise will give you some practice using the API documentation to look up information about a class.

1. Start a web browser and navigate to the index page that contains the API documentation for the JDK (usually `C:\Program Files\Java\jdk1.6.0\docs\api\index.htm`). This page should look like the one shown in figure 1-16.
2. Bookmark this page so you can easily access it later. To do that with the Internet Explorer, select the `Add To Favorites` item from the `Favorites` menu. Then, close your web browser.
3. Start your web browser again and use the bookmark to return to the API documentation for the JDK. To do that with the Internet Explorer, select the `Java Platform SE` item from the `Favorites` menu.
4. Select the `java.lang` package in the upper left frame and notice that the links in the lower left frame change. Select the `System` class from this frame to display information about it in the right frame.
5. Scroll down to the `Field Summary` area in the right frame and click on the out link for the standard output stream. When you do, an HTML page that gives some information about how to use the standard output stream will be displayed. In the next chapter, you'll learn more about using the `out` field of the `System` class to print data to the console.
6. Continue experimenting with the documentation until you're comfortable with how it works. Then, close the browser.





# Introduction to Java programming

Once you've got Java on your system, the quickest and best way to *learn* Java programming is to *do* Java programming. That's why this chapter shows you how to write complete Java programs that get input from a user, make calculations, and display output. When you finish this chapter, you should be able to write comparable programs of your own.

<b>Basic coding skills .....</b>	<b>46</b>
How to code statements .....	46
How to code comments .....	46
How to create identifiers .....	48
How to declare a class .....	50
How to declare a main method .....	52
<b>How to work with numeric variables .....</b>	<b>54</b>
How to declare and initialize variables .....	54
How to code assignment statements .....	56
How to code arithmetic expressions .....	56
<b>How to work with string variables .....</b>	<b>58</b>
How to create a String object .....	58
How to join and append strings .....	58
How to include special characters in strings .....	60
<b>How to use Java classes, objects, and methods .....</b>	<b>62</b>
How to import Java classes .....	62
How to create objects and call methods .....	64
How to use the API documentation to research Java classes .....	66
<b>How to use the console for input and output .....</b>	<b>68</b>
How to use the System.out object to print output to the console .....	68
How to use the Scanner class to read input from the console .....	70
Examples that get input from the console .....	72
<b>How to code simple control statements .....</b>	<b>74</b>
How to compare numeric variables .....	74
How to compare string variables .....	74
How to code if/else statements .....	76
How to code while statements .....	78
<b>Two illustrative applications .....</b>	<b>80</b>
The Invoice application .....	80
The Test Score application .....	82
<b>How to test and debug an application .....</b>	<b>84</b>
How to test an application .....	84
How to debug an application .....	84
<b>Perspective .....</b>	<b>86</b>

## Basic coding skills

---

This chapter starts by introducing you to some basic coding skills. You'll use these skills for every Java program you develop. Once you understand these skills, you'll be ready to learn how to write simple programs.

### How to code statements

---

The *statements* in a Java program direct the operation of the program. When you code a statement, you can start it anywhere in a coding line, you can continue it from one line to another, and you can code one or more spaces anywhere a single space is valid. In the first example in figure 2-1, the statements aren't shaded.

To end most statements, you use a semicolon. But when a statement requires a set of braces { }, it ends with the right brace. Then, the statements within the braces are referred to as a *block* of code.

To make a program easier to read, you should use indentation and spacing to align statements and blocks of code. This is illustrated by the program in this figure and by all of the programs and examples in this book.

### How to code comments

---

*Comments* are used in Java programs to document what the program does and what specific blocks and lines of code do. Since the Java compiler ignores comments, you can include them anywhere in a program without affecting your code. In the first example in figure 2-1, the comments are shaded.

A *single-line comment* is typically used to describe one or more lines of code. This type of comment starts with two slashes (//) that tell the compiler to ignore all characters until the end of the current line. In the first example in this figure, you can see four single-line comments that are used to describe groups of statements. The other comment is coded after a statement to describe what that statement does. This type of comment is sometimes referred to as an *end-of-line comment*.

The second example in this figure shows how to code a *block comment*. This type of comment is typically used to document information that applies to a block of code. This information can include the author's name, program completion date, the purpose of the code, the files used by the code, and so on.

Although many programmers sprinkle their code with comments, that shouldn't be necessary if you write code that's easy to read and understand. Instead, you should use comments only to clarify code that's difficult to understand. In this figure, for example, an experienced Java programmer wouldn't need any of the single-line comments.

One problem with comments is that they may not accurately represent what the code does. This often happens when a programmer changes the code, but doesn't change the comments that go along with it. Then, it's even harder to understand the code because the comments are misleading. So if you change the code that you've written comments for, be sure to change the comments too.

## An application consists of statements and comments

```
import java.util.Scanner;

public class InvoiceApp
{
    public static void main(String[] args)
    {
        // display a welcome message
        System.out.println("Welcome to the Invoice Total Calculator");
        System.out.println(); // print a blank line

        // get the input from the user
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter subtotal: ");
        double subtotal = sc.nextDouble();

        // calculate the discount amount and total
        double discountPercent = .2;
        double discountAmount = subtotal * discountPercent;
        double invoiceTotal = subtotal - discountAmount;

        // format and display the result
        String message = "Discount percent: " + discountPercent + "\n"
            + "Discount amount: " + discountAmount + "\n"
            + "Invoice total: " + invoiceTotal + "\n";
        System.out.println(message);
    }
}
```

## A block comment that could be coded at the start of a program

```
/*
 * Date: 9/1/04
 * Author: A. Steelman
 * Purpose: This program uses the console to get a subtotal from the user.
 * Then, it calculates the discount amount and total
 * and displays these values on the console.
 */
```

## Description

- Java *statements* direct the operations of a program, while *comments* are used to help document what the program does.
- You can start a statement at any point in a line and continue the statement from one line to the next. To make a program easier to read, you should use indentation and extra spaces to align statements and parts of statements.
- Most statements end with a semicolon. But when a statement requires a set of braces { }, the statement ends with the right brace. Then, the code within the braces can be referred to as a *block* of code.
- To code a *single-line comment*, type // followed by the comment. You can code a single-line comment on a line by itself or after a statement. A comment that's coded after a statement is sometimes called an *end-of-line comment*.
- To code a *block comment*, type /\* at the start of the block and \*/ at the end. You can also code asterisks to identify the lines in the block, but that isn't necessary.

Figure 2-1 How to code comments and statements

## How to create identifiers

---

As you code a Java program, you need to create and use *identifiers*. These are the names in the program that you define. In each program, for example, you need to create an identifier for the name of the program and for the variables that are used by the program.

Figure 2-2 shows you how to create identifiers. In brief, you must start each identifier with a letter, underscore, or dollar sign. After that first character, you can use any combination of letters, underscores, dollar signs, or digits.

Since Java is case-sensitive, you need to be careful when you create and use identifiers. If, for example, you define an identifier as `CustomerAddress`, you can't refer to it later as `Customeraddress`. That's a common compile-time error.

When you create an identifier, you should try to make the name both meaningful and easy to remember. To make a name meaningful, you should use as many characters as you need, so it's easy for other programmers to read and understand your code. For instance, `netPrice` is more meaningful than `nPrice`, and `nPrice` is more meaningful than `np`.

To make a name easy to remember, you should avoid abbreviations. If, for example, you use `nwCst` as an identifier, you may have difficulty remembering whether it was `nCust`, `nwCust`, or `nwCst` later on. If you code the name as `newCustomer`, though, you won't have any trouble remembering what it was. Yes, you type more characters when you create identifiers that are meaningful and easy to remember, but that will be more than justified by the time you'll save when you test, debug, and maintain the program.

For some common identifiers, though, programmers typically use just one or two lowercase letters. For instance, they often use the letters *i*, *j*, and *k* to identify counter variables. You'll see examples of this later in this chapter.

Notice that you can't create an identifier that is the same as one of the Java *keywords*. These 50 keywords are reserved by the Java language and are the basis for that language. To help you identify keywords in your code, Java-enabled text editors and Java IDEs display these keywords in a different color than the rest of the Java code. For example, TextPad displays keywords in blue. As you progress through this book, you'll learn how to use almost all of these keywords.

## Valid identifiers

InvoiceApp	\$orderTotal	i
Invoice	_orderTotal	x
InvoiceApp2	input_string	TITLE
subtotal	_get_total	MONTHS_PER_YEAR
discountPercent	\$_64_Valid	

## The rules for naming an identifier

- Start each identifier with a letter, underscore, or dollar sign. Use letters, dollar signs, underscores, or digits for subsequent characters.
- Use up to 255 characters.
- Don't use Java keywords.

## Keywords

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

## Description

- An *identifier* is any name that you create in a Java program. These can be the names of classes, methods, variables, and so on.
- A *keyword* is a word that's reserved by the Java language. As a result, you can't use keywords as identifiers.
- When you refer to an identifier, be sure to use the correct uppercase and lowercase letters because Java is a case-sensitive language.

Figure 2-2 How to create identifiers

## How to declare a class

---

When you write Java code, you store your code in a *class*. As a result, to write a Java program, you must write at least one class. As you learned in chapter 1, the code for each class is stored in a \*.java file, and the compiled code is stored in a \*.class file. To write a class, you must begin with a *class declaration* like the ones shown in figure 2-3.

In the syntax for declaring a class, the boldfaced words are Java keywords, and the words that aren't boldfaced represent code that the programmer supplies. The bar ( | ) in this syntax means that you have a choice between the two items that the bar separates. In this case, the bar means that you can start the declaration with the public keyword or the private keyword.

The public and private keywords are *access modifiers* that control the *scope* of a class. Usually, a class is declared public, which means that other classes can access it. In fact, you must declare one (and only one) public class for every \*.java file. Later in this book, you'll learn when and how to use private classes.

After the public keyword and the class keyword, you code the name of the class using the basic rules for creating an identifier. When you do, it's a common coding convention to start a class name with a capital letter and to use letters and digits only. It's also common to start every word within the name with a capital letter. We also recommend that you use a noun or a noun that's preceded by one or more adjectives for your class names. In this figure, all four class names adhere to these rules and guidelines.

After the class name, the syntax summary shows a left brace, the statements that make up the class, and a right brace. It's a good coding practice, though, to type your ending brace right after you type the starting brace, and then type your code between the two braces. That prevents missing braces, which is a common compile-time error.

The two InvoiceApp classes in this figure show how a class works. Here, the class declaration and its braces are shaded and the code for the class is between the braces. In this simple example, the class contains a single block of code that displays a message. You'll learn more about how this code works in the next figure.

Notice that the only difference between the two classes in this figure is where the opening braces for the class and the block of code within the class are placed. Some programmers prefer to code the brace on a separate line because they think that the additional vertical spacing makes the code easier to read. Others prefer to code the brace immediately following the class or method name because it saves vertical space and because they believe that it is equally easy to read. Although either technique is acceptable, we've chosen to use the first technique for this book whenever possible.

As you learned in the last chapter, when you save your class on disk, you save it with a name that consists of the public class name and the *java* extension. As a result, you save the class in this figure with the name InvoiceApp.java.

## The syntax for declaring a class

```
public|private class ClassName
{
    statements
}
```

## Typical class declarations

```
public class InvoiceApp{}
public class ProductOrderApp{}
public class Product{}
public class ProductOrder{}
```

## A public class named InvoiceApp

```
public class InvoiceApp                // declare the class
{                                     // begin the class
    public static void main(String[] args)
    {
        System.out.println("Welcome to the Invoice Total Calculator");
    }
}                                     // end the class
```

## The same class with different brace placement

```
public class InvoiceApp{                // declare and begin the class
    public static void main(String[] args){
        System.out.println("Welcome to the Invoice Total Calculator");
    }
}                                     // end the class
```

## The rules for naming a class

- Start the name with a capital letter.
- Use letters and digits only.
- Follow the other rules for naming an identifier.

## Naming recommendations for classes

- Start every word within a class name with an initial cap.
- Each class name should be a noun or a noun that's preceded by one or more adjectives.

## Description

- When you develop a Java application, you code one or more *classes* for it. For each class, you must code a *class declaration*. Then, you write the code for the class within the opening and closing braces of the declaration.
- The public and private keywords are *access modifiers* that control what parts of the program can use the class. If a class is public, the class can be used by all parts of the program.
- Most classes are declared public, and each file must contain one and only one public class. The file name for a class is the same as the class name with *java* as the extension.

Figure 2-3 How to declare a class

## How to declare a main method

---

Every Java program contains one or more *methods*, which are pieces of code that perform tasks (they're similar to *functions* in some programming languages). The *main method* is a special kind of method that's automatically executed when the class that holds it is run. All Java programs contain a main method that starts the program.

To code a main method, you begin by coding a *main method declaration* as shown in figure 2-4. For now, you can code every main method declaration using the code exactly as it's shown, even if you don't understand the code. Although this figure gives a partial explanation for each keyword in the method, you can skip that if you like because you'll learn more about each of these keywords as you progress through this book. We included this summary for those who are already familiar with object-oriented programming.

As you can see in this figure, the main method of the `InvoiceApp` class is coded within the class declaration. To make this structure clear, the main method is indented and the starting and ending braces are aligned so it's easy to see where the method begins and ends. Then, between the braces, you can see the one statement that this main method performs.



## The syntax for declaring a main method

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

## The main method of the InvoiceApp class

```
public class InvoiceApp
{
    public static void main(String[] args)
    {
        // begin main method
        System.out.println("Welcome to the Invoice Total Calculator");
        // end main method
    }
}
```

## Description

- A *method* is a block of code that performs a task.
- Every Java application contains one *main method* that you can declare exactly as shown above. This is called the *main method declaration*.
- The statements between the braces in a main method declaration are run when the program is executed.

## Partial explanation of the terms in the main method declaration

- The *public* keyword in the declaration means that other classes can access the main method. The *static* keyword means that the method can be called directly from the other classes without first creating an object. And the *void* keyword means that the method won't return any values.
- The *main* identifier is the name of the method. When you code a method, always include parentheses after the name of the method.
- The code in the parentheses lists the *arguments* that the method uses, and every main method receives an argument named *args*, which is defined as an array of strings.

## How to work with numeric variables

---

In this topic, you'll learn how to work with numeric variables. This will introduce you to the use of variables, assignment statements, arithmetic expressions, and two of the eight primitive data types that are supported by Java. Then, you can learn all the details about working with the primitive data types in the next chapter.

### How to declare and initialize variables

---

A *variable* is used to store a value that can change as the program executes. Before you can use a variable, you must *declare* its data type and name, and you must *assign* a value to it to *initialize* it. The easiest way to do that is shown in figure 2-5. Just code the data type, the variable name, the equals sign, and the value that you want to assign to the variable.

This figure also summarizes two of the eight Java *data types*. You can use the *int* data type to store *integers*, which are numbers that don't contain decimal places (whole numbers), and you can use the *double* data type to store numbers that contain decimal places. In the next chapter, you'll learn how to use the six other primitive data types, but these are the two that you'll probably use the most.

As you can see in the summary, the double data type can be used to store very large and very small numbers with up to 16 significant digits. In case you aren't familiar with E notation, .123456E+7 means .123456 times  $10^7$ , which is 1234560. And 1234E-5 means 1234 times  $10^{-5}$ , which is .01234. Here, the first example has six significant digits, and the second one has four significant digits. For business applications, you usually don't need to use this notation, and 16 significant digits are usually enough. In the next chapter, though, you'll learn how you can go beyond that 16 digit limitation whenever that's necessary.

To illustrate the declaration of variables, the first example in this figure declares an *int* variable named `scoreCounter` with an initial value of 1. And the second example declares a *double* variable named `unitPrice` with an initial value of 14.95. When you assign values to double types, it's a good coding practice to include a decimal point, even if the initial value is a whole number. If, for example, you want to assign the number 29 to the variable, you should code the number as 29.0.

If you follow the naming recommendations in this figure as you name the variables, it will make your programs easier to read and understand. In particular, you should capitalize the first letter in each word of the variable name, except the first word, as in `scoreCounter` or `unitPrice`. This is commonly referred to as *camel notation*.

When you initialize a variable, you can assign a *literal* value like 1 or 14.95 to a variable as illustrated by the examples in this figure. However, you can also initialize a variable to the value of another variable or to the value of an expression like the arithmetic expressions shown in the next figure.

## Two of the eight primitive data types

Type	Bytes	Description
int	4	Integers from -2,147,483,648 to 2,147,483,647.
double	8	Numbers with decimal places that range from -1.7E308 to 1.7E308 with up to 16 significant digits.

## How to declare and initialize a variable in one statement

### Syntax

```
type variableName = value;
```

### Examples

```
int scoreCounter = 1;           // initialize an integer variable
double unitPrice = 14.95;       // initialize a double variable
```

## How to code assignment statements

```
int quantity = 0;               // initialize an integer variable
int maxQuantity = 100;          // initialize another integer variable

// two assignment statements
quantity = 10;                  // quantity is now 10
quantity = maxQuantity;          // quantity is now 100
```

## Description

- A *variable* stores a value that can change as a program executes.
- Java provides for eight *primitive data types* that you can use for storing values in memory. The two that you'll use the most are the `int` and `double` data types. In the next chapter, you'll learn how to use the other primitive data types.
- The *int* data type is used for storing *integers* (whole numbers). The *double* data type is used for storing numbers that can have one or more decimal places.
- To *initialize* a variable, you *declare* a data type and *assign* an initial value to the variable. As default values, it's common to initialize integer variables to 0 and double variables to 0.0.
- An *assignment statement* assigns a value to a variable. This value can be a literal value, another variable, or an expression like the arithmetic expressions that you'll learn how to code in the next figure. If a variable has already been declared, the assignment statement doesn't include the data type of the variable.

## Naming recommendations for variables

- Start variable names with a lowercase letter and capitalize the first letter in all words after the first word.
- Each variable name should be a noun or a noun preceded by one or more adjectives.
- Try to use meaningful names that are easy to remember.

Figure 2-5 How to declare and initialize variables

## How to code assignment statements

---

After you declare a variable, you can assign a new value to it. To do that, you code an *assignment statement*. In a simple assignment statement, you code the variable name, an equals sign, and a new value. The new value can be a literal value or the name of another variable as shown in figure 2-5. Or, the new value can be the result of an expression like the arithmetic expressions shown in figure 2-6.

## How to code arithmetic expressions

---

To code simple *arithmetic expressions*, you can use the four *arithmetic operators* that are summarized in figure 2-6. As the first group of statements shows, these operators work the way you would expect them to with one exception. If you divide one integer into another integer, any decimal places are truncated. In contrast, if you divide a double into a double, the decimal places are included in the result.

If you code more than one operator in an expression, all of the multiplication and division operations are done first, from left to right. Then, the addition and subtraction operations are done, from left to right. If this isn't the way you want the expression to be evaluated, you can use parentheses to specify the sequence of operations in much the same way that you use parentheses in algebraic expressions. In the next chapter, you'll learn more about how this works.

When you code assignment statements, it's common to code the same variable on both sides of the equals sign. For example, you can add 1 to the value of a variable named `counter` with a statement like this:

```
counter = counter + 1;
```

In this case, if `counter` has a value of 5 when the statement starts, it will have a value of 6 when the statement finishes. This concept is illustrated by the second and third groups of statements.

What happens when you mix integer and double variables in the same arithmetic expression? The integers are *cast* (converted) to doubles so the decimal places can be included in the result. To retain the decimal places, though, the result variable must be a double. This is illustrated by the fourth group of statements.

## The basic operators that you can use in arithmetic expressions

Operator	Name	Description
+	Addition	Adds two operands.
-	Subtraction	Subtracts the right operand from the left operand.
*	Multiplication	Multiplies the right operand and the left operand.
/	Division	Divides the right operand into the left operand. If both operands are integers, then the result is an integer.

## Statements that use simple arithmetic expressions

```
// integer arithmetic
int x = 14;
int y = 8;
int result1 = x + y;           // result1 = 22
int result2 = x - y;           // result2 = 6
int result3 = x * y;           // result3 = 112
int result4 = x / y;           // result4 = 1

// double arithmetic
double a = 8.5;
double b = 3.4;
double result5 = a + b;        // result5 = 11.9
double result6 = a - b;        // result6 = 5.1
double result7 = a * b;        // result7 = 28.9
double result8 = a / b;        // result8 = 2.5
```

## Statements that increment a counter variable

```
int invoiceCount = 0;
invoiceCount = invoiceCount + 1;           // invoiceCount = 1
invoiceCount = invoiceCount + 1;           // invoiceCount = 2
```

## Statements that add amounts to a total

```
double invoiceAmount1 = 150.25;
double invoiceAmount2 = 100.75;
double invoiceTotal = 0.0;
invoiceTotal = invoiceTotal + invoiceAmount1; // invoiceTotal = 150.25
invoiceTotal = invoiceTotal + invoiceAmount2; // invoiceTotal = 251.00
```

## Statements that mix int and double variables

```
int result9 = invoiceTotal / invoiceCount // result9 = 125
double result10 = invoiceTotal / invoiceCount // result10 = 125.50
```

## Description

- An *arithmetic expression* consists of one or more *operands* and *arithmetic operators*.
- When an expression mixes the use of int and double variables, Java automatically *casts* the int types to double types. To retain the decimal places, the variable that receives the result must be a double.
- In the next chapter, you'll learn how to code expressions that contain two or more operators.

Figure 2-6 How to code arithmetic expressions

## How to work with string variables

---

In the topics that follow, you'll learn some basic skills for working with strings. For now, these skills should be all you need for many of the programs you develop. Keep in mind, though, that many programs require extensive string operations. That's why chapter 12 covers strings in more detail.

### How to create a String object

---

A *string* can consist of any letters, numbers, and special characters. To declare a string variable, you use the syntax shown in figure 2-7. Although this is much like the syntax for declaring a numeric variable, a *String object* is created from the `String` class when a string variable is declared. Then, the string data is stored in that object. When you declare a string variable, you must capitalize the `String` keyword because it is the name of a class, not a primitive data type.

In the next topic and in chapter 6, you'll learn much more about classes and objects. For now, though, all you need to know is that string variables work much like numeric variables, except that they store string data instead of numeric data.

When you declare a `String` object, you can assign a *string literal* to it by enclosing the characters within double quotes. You can also assign an *empty string* to it by coding a set of quotation marks with nothing between them. Finally, you can use the `null` keyword to assign a *null value* to a `String` object. That indicates that the value of the string is unknown.

### How to join and append strings

---

If you want to *join*, or *concatenate*, two or more strings into one, you can use the `+` operator. For example, you can join a first name, a space, and a last name as shown in the second example in this figure. Then, you can assign that string to a variable. Notice that when concatenating strings, you can use string variables or string literals.

You can also join a string with a primitive data type. This is illustrated in the third example in this figure. Here, a variable that's defined with the `double` data type is appended to a string. When you use this technique, Java automatically converts the `double` value to a string.

You can use the `+` and `+=` operators to *append* a string to the end of a string that's stored in a string variable. If you use the `+` operator, you need to include the variable on both sides of the `=` operator. Otherwise, the assignment statement will replace the old value with the new value instead of appending the old value to the new value. Since the `+=` operator provides a shorter and safer way to append strings, this operator is commonly used.

## The syntax for declaring and initializing a string variable

```
String variableName = value;
```

### Example 1: How to declare and initialize a string

```
String message1 = "Invalid data entry.";
String message2 = "";
String message3 = null;
```

### Example 2: How to join strings

```
String firstName = "Bob";           // firstName is Bob
String lastName = "Smith";          // lastName is Smith
String name = firstName + " " + lastName; // name is Bob Smith
```

### Example 3: How to join a string and a number

```
double price = 14.95;
String priceString = "Price: " + price;
```

### Example 4: How to append one string to another with the + operator

```
firstName = "Bob";           // firstName is Bob
lastName = "Smith";          // lastName is Smith
name = firstName + " ";      // name is Bob followed by a space
name = name + lastName;      // name is Bob Smith
```

### Example 5: How to append one string to another with the += operator

```
firstName = "Bob";           // firstName is Bob
lastName = "Smith";          // lastName is Smith
name = firstName + " ";      // name is Bob followed by a space
name += lastName;            // name is Bob Smith
```

## Description

- A *string* can consist of any characters in the character set including letters, numbers, and special characters like \*, &, and #.
- In Java, a string is actually a String object that's created from the String class that's part of the Java API.
- To specify the value of a string, you can enclose text in double quotation marks. This is known as a *string literal*.
- To assign a *null value* to a string, you can use the null keyword. This means that the value of the string is unknown.
- To assign an *empty string* to a String object, you can code a set of quotation marks with nothing between them. This means that the string doesn't contain any characters.
- To *join* (or *concatenate*) a string with another string or a data type, use a plus sign. Whenever possible, Java will automatically convert the data type so it can be used as part of the string.
- When you *append* one string to another, you add one string to the end of another. To do that, you can use assignment statements.
- The += operator is a shortcut for appending a string expression to a string variable.

Figure 2-7 How to create and use strings

## How to include special characters in strings

---

Figure 2-8 shows how to include certain types of special characters within a string. In particular, this figure shows how to include backslashes, quotation marks, and control characters such as new lines, tabs, and returns in a string. To do that, you can use the *escape sequences* shown in this figure.

As you can see, each escape sequence starts with a backslash. The backslash tells the compiler that the character that follows should be treated as a special character and not interpreted as a literal value. If you code a backslash followed by the letter *n*, for example, the compiler will include a new line character in the string. You can see how this works in the first example in this figure. If you omitted the backslash, of course, the compiler would just include the letter *n* in the string value. The escape sequences for the tab and return characters work similarly, as you can see in the second example.

To code a string literal, you enclose it in double quotes. If you want to include a double quote within a string literal, then, you must use an escape sequence. This is illustrated in the third example. Here, the `\` escape sequence is used to include two double quotes within the string literal.

Finally, you need to use an escape sequence if you want to include a backslash in a string literal. To do that, you code two backslashes as shown in the fourth example. If you code a single backslash, the compiler will treat the next character as a special character. That will cause a compiler error if the character isn't a valid special character. And if the character is a valid special character, the results won't be what you want.



## Common escape sequences

Sequence	Character
<code>\n</code>	New line
<code>\t</code>	Tab
<code>\r</code>	Return
<code>\"</code>	Quotation mark
<code>\\</code>	Backslash

### Example 1: New line

**String**

```
"Code: JSPS\nPrice: $49.50"
```

**Result**

```
Code: JSPS
```

```
Price: $49.50
```

### Example 2: Tabs and returns

**String**

```
"Joe\tSmith\rKate\tLewis"
```

**Result**

```
Joe      Smith
```

```
Kate     Lewis
```

### Example 3: Quotation marks

**String**

```
"Type \"x\" to exit"
```

**Result**

```
Type "x" to exit
```

### Example 4: Backslash

**String**

```
"C:\\java\\files"
```

**Result**

```
C:\java\files
```

## Description

- Within a string, you can use *escape sequences* to include certain types of special characters.

---

Figure 2-8 How to include special characters in strings

# How to use Java classes, objects, and methods

---

So far, you've learned how to create String objects from the String class in the Java API. As you develop Java applications, though, you need to use dozens of different Java classes and objects. To do that, you need to know how to import Java classes, how to create objects from Java classes, and how to call Java methods.

## How to import Java classes

---

In the API for the Java SE, groups of related classes are organized into *packages*. In figure 2-9, you can see a list of some of the commonly used packages. Since the java.lang package contains the classes that are used in almost every Java program (such as the String class), this package is automatically made available to all programs.

To use a class from a package other than java.lang, though, you'll typically include an import statement for that class at the beginning of the program. If you don't, you'll still be able to use the class, but you'll have to qualify it with the name of the package that contains it each time you refer to it. Since that can lead to a lot of unnecessary typing, we recommend that you always code an import statement for the classes you use.

When you code an import statement, you can import a single class by specifying the class name, or you can import all of the classes in the package by typing an asterisk (\*) in place of the class name. The first two statements in this figure, for example, import a single class, while the next two import all of the classes in a package. Although it requires less code to import all of the classes in a package at once, importing one class at a time clearly identifies the classes you're using.

As this figure shows, Java provides two different technologies for building a graphical user interface (GUI) that contains text boxes, command buttons, combo boxes, and so on. The older technology, known as the *Abstract Window Toolkit (AWT)*, was used with versions 1.0 and 1.1 of Java. Its classes are stored in the java.awt package. Since version 1.2 of Java, though, a new technology known as *Swing* has been available. The Swing classes are stored in the javax.swing package. In general, many of the newer package names begin with javax instead of java. Here, the x indicates that these packages can be considered extensions to the original Java API.

In addition to the packages provided by the Java API, you can get packages from third party sources, either as shareware or by purchasing them. For more information, check the Java web site. You can also create packages that contain classes that you've written. You'll learn how to do that in chapter 9.

## Common packages

Package name	Description
<code>java.lang</code>	Provides classes fundamental to Java, including classes that work with primitive data types, strings, and math functions.
<code>java.text</code>	Provides classes to handle text, dates, and numbers.
<code>java.util</code>	Provides various utility classes including those for working with collections.
<code>java.io</code>	Provides classes to read data from files and to write data to files.
<code>java.sql</code>	Provides classes to read data from databases and to write data to databases.
<code>java.applet</code>	An older package that provides classes to create an applet.
<code>java.awt</code>	An older package called the <i>Abstract Window Toolkit</i> (AWT) that provides classes to create graphical user interfaces.
<code>java.awt.event</code>	A package that provides classes necessary to handle events.
<code>javax.swing</code>	A newer package called <i>Swing</i> that provides classes to create graphical user interfaces and applets.

## The syntax of the import statement

```
import packagename.ClassName;
or
import packagename.*;
```

## Examples

```
import java.text.NumberFormat;
import java.util.Scanner;
import java.util.*;
import javax.swing.*;
```

## How to use the Scanner class to create an object

### With an import statement

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

### Without an import statement the package name is required as a qualifier

```
java.util.Scanner sc = new java.util.Scanner(System.in);
```

## Description

- The API for the Java SE provides a large library of classes that are organized into *packages*.
- All classes stored in the `java.lang` package are automatically available to all Java programs.
- To use classes that aren't in the `java.lang` package, you can code an import statement as shown above. To import one class from a package, specify the package name followed by the class name. To import all classes in a package, specify the package name followed by an asterisk (\*).

Figure 2-9 How to import Java classes

## How to create objects and call methods

---

To use a Java class, you usually start by creating an *object* from a Java *class*. As the syntax in figure 2-10 shows, you do that by coding the Java class name, the name that you want to use for the object, an equals sign, the new keyword, and the Java class name again followed by a set of parentheses. Within the parentheses, you code any *arguments* that are required by the *constructor* of the object that's defined in the class.

In the examples, the first statement shows how to create a Scanner object named `sc`. The constructor for this object requires just one argument (`System.in`), which represents console input. In contrast, the second statement creates a Date object named `now` that represents the current date, but its constructor doesn't require any arguments. As you go through this book, you'll learn how to create objects with constructors that require two or more arguments, and you'll see that a single class can provide more than one constructor for creating objects.

When you create an object, you can think of the class as the template for the object. That's why the object can be called an *instance* of the class, and the process of creating the object can be called *instantiation*. Whenever necessary, you can create more than one object or instance from the class. For instance, you often use several String objects in a single program.

Once you've created an object from a class, you can use the *methods* of the class. To *call* one of these methods, you code the object name, a dot (period), and the method name followed by a set of parentheses. Within the parentheses, you code the arguments that are required by the method.

In the examples, the first statement calls the `nextDouble` method of the Scanner object named `sc` to get data from the console. The second statement calls the `toString` method of the Date object named `now` to convert the date and time that's stored in the object to a string. Neither one of these methods requires an argument, but you'll soon see some that do.

Besides methods that you can call from an object, some classes provide *static methods* that can be called directly from the class. To do that, you substitute the class name for the object name as illustrated by the third set of examples. Here, the first statement calls the `toString` method of the Double class, and the second statement calls the `parseDouble` method of the Double class. Both of these methods require one argument.

Incidentally, you can also use the syntax shown in this figure to create a String object. However, the preferred way to create a String object is to use the syntax shown in figure 2-7. Once a String object is created, though, you use the syntax in this figure to use one of its methods. You'll see examples of this later in this chapter.

In the pages and chapters that follow, you'll learn how to use dozens of classes and methods. For now, though, you just need to focus on the syntax for creating an object from a class, for calling a method from an object, and for calling a static method from a class. Once you understand that, you're ready to learn how to research the Java classes and methods that you might want to use.

## How to create an object from a class

### Syntax

```
ClassName objectName = new ClassName(arguments);
```

### Examples

```
Scanner sc = new Scanner(System.in);    // creates a Scanner object named sc
Date now = new Date();                  // creates a Date object named now
```

## How to call a method from an object

### Syntax

```
objectName.methodName(arguments)
```

### Examples

```
double subtotal = sc.nextDouble();    // get a double entry from the console
String currentDate = now.toString();  // convert the date to a string
```

## How to call a static method from a class

### Syntax

```
ClassName.methodName(arguments)
```

### Examples

```
String sPrice = Double.toString(price);    // convert a double to a string
double total = Double.parseDouble(userEntry); // convert a string to a double
```

## Description

- When you create an *object* from a Java class, you are creating an *instance* of the *class*. Then, you can use the *methods* of the class by *calling* them from the object.
- Some Java classes contain *static methods*. These methods can be called directly from the class without creating an object.
- When you create an object from a class, the *constructor* may require one or more *arguments*. These arguments must have the required data types, and they must be coded in the correct sequence separated by commas.
- When you call a method from an object or a class, the method may require one or more arguments. Here again, these arguments must have the required data types and they must be coded in the correct sequence separated by commas.
- Although you can use the syntax shown in this figure to create a String object, the syntax in figure 2-7 is the preferred way to do that. Once a String object is created, though, you call its methods from the object as shown above.
- In this book, you'll learn how to use dozens of the Java classes and methods that you'll use the most in your applications. You will also learn how to create your own classes and methods.

Figure 2-10 How to create objects and call methods

## How to use the API documentation to research Java classes

---

If you refer back to the list of keywords in figure 2-2, you can see that the Java language consists of just 50 keywords that you can master with relative ease. What's difficult about using Java, though, is mastering the hundreds of classes and methods that your applications will require. To do that, you frequently need to study the API documentation that comes with Java, and that is one of the most time-consuming aspects of Java programming.

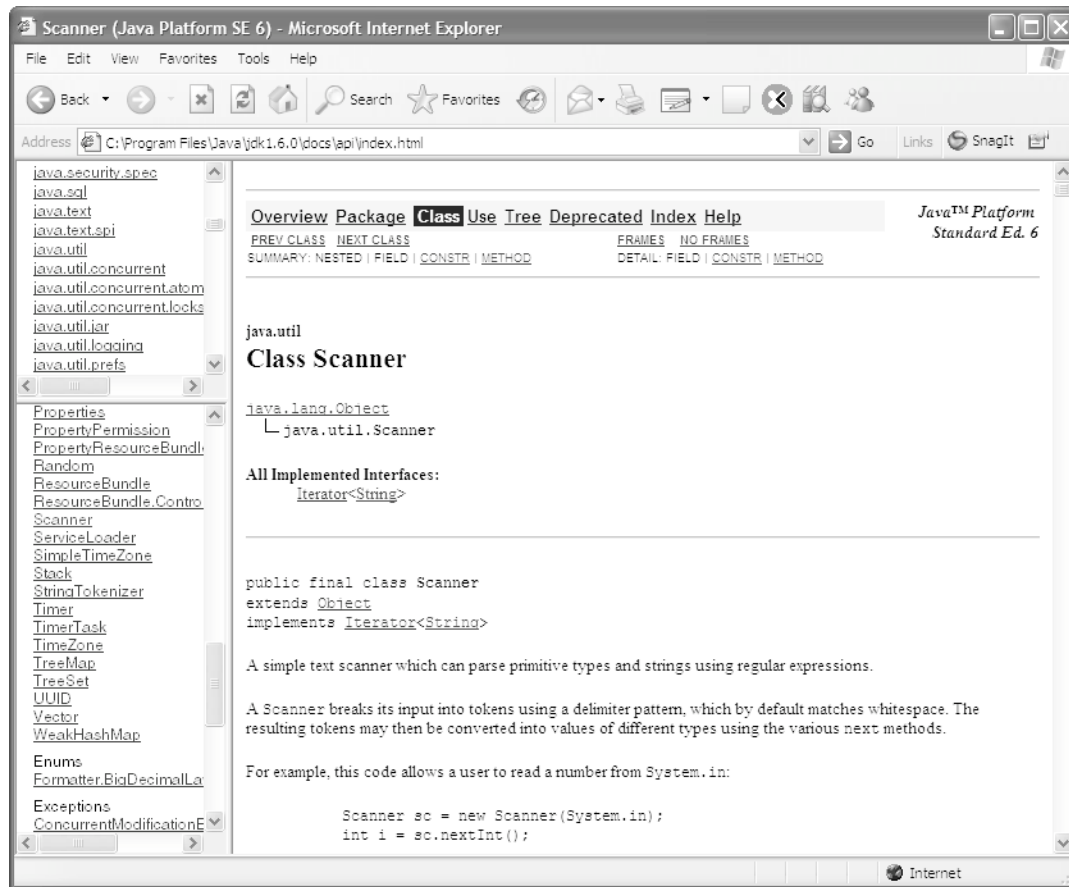
Figure 2-11 summarizes some of the basic techniques for navigating through the API documentation. Here, you can see the start of the documentation for the `Scanner` class, which goes on for many pages. To get there, you click on the package name in the upper left frame and then on the class name in the lower left frame.

If you scroll through the documentation for this class, you'll get an idea of the scale of the documentation that you're dealing with. After a few pages of descriptive information, you come to a summary of the eight constructors for the class. After that, you come to a summary of the dozens of methods that the class offers. That in turn is followed by more detail about the constructors, and then by more detail about the methods.

At this point in your development, this is far more information than you can handle. That's why one of the goals of this book is to introduce you to the dozens of classes and methods that you'll use in most of the applications that you develop. Once you've learned those, the API documentation will make more sense to you, and you'll be able to use that documentation to research classes and methods that aren't presented in this book. To get you started with the use of objects and methods, the next topic will show you how to use the `Scanner` class.

It's never too early to start using the documentation, though. So by all means use the documentation to get more information about the methods that are presented in this book and to research the other methods that are offered by the classes that are presented in this book. After you learn how to use the `Scanner` class, for example, take some time to do some research on that class. You'll get a chance to do that in exercise 2-4.

## The documentation for the Scanner class



## Description

- The Java SE API contains thousands of classes and methods that can help you do most of the tasks that your applications require. However, researching the Java API documentation to find the classes and methods that you need is one of the most time-consuming aspects of Java programming.
- If you've installed the API documentation on your hard drive, you can display an index by using your web browser to go to the `index.html` file in the `docs\api` directory. If you haven't installed the documentation, you can browse through it on the Java web site.
- You can select the name of the package in the top left frame to display information about the package and the classes it contains. Then, you can select a class in the lower left frame to display the documentation for that class in the right frame.
- Once you display the documentation for a class, you can scroll through it or click on a hyperlink to get more information.
- To make it easier to access the API documentation, you should bookmark the index page. Then, you can easily redisplay this page whenever you need it.

Figure 2-11 How to use the API documentation to research Java classes

## How to use the console for input and output

---

Most of the applications that you write will require some type of user interaction. In particular, most applications will get input from the user and display output to the user. Ever since version 1.5 of Java, the easiest way to get input is to use the new `Scanner` class to retrieve data from the console. And the easiest way to display output is to print it to the console.

### How to use the `System.out` object to print output to the console

---

To print output to the *console*, you can use the `println` and `print` methods of the `System.out` object as shown in figure 2-12. Here, `System.out` actually refers to an instance of the `PrintStream` class, where `out` is a public variable that's defined by the `System` class. From a practical point of view, though, you can just think of `System.out` as an object that represents console output. And you don't need to code any other statements in your program to create that object.

Both the `println` and `print` methods require a string argument that specifies the data to be printed. The only difference between the two is that the `println` method starts a new line after it displays the data, and the `print` method doesn't.

If you study the examples in this figure, you shouldn't have any trouble using these methods. For instance, the first statement in the first example uses the `println` method to print the words "Welcome to the Invoice Total Calculator" to the console. The second statement prints the string "Total: " followed by the value of the `total` variable (which is automatically converted to a string by this `join`). The third statement prints the value of the variable named `message` to the console. And the fourth statement prints a blank line since no argument is coded.

Because the `print` method doesn't automatically start a new line, you can use it to print several data arguments on the same line. For instance, the three statements in the second example use the `print` method to print "Total: ", followed by the `total` variable, followed by a new line character. Of course, you can achieve the same result with a single line of code like this:

```
System.out.println("Total: " + total);
```

This figure also shows an application that uses the `println` method to print seven lines to the console. In the `main` method of this application, the first four statements set the values for four variables. Then, the next seven statements print a welcome message, a blank line, the values for the four variables, and another blank line.

When you work with console applications, you should know that the appearance of the console may differ slightly depending on the operating system. The example in this figure shows a console for Windows. Even if the console looks a little different, however, it should work the same.

Later in this book, you'll learn how to format data before you use the `print` and `println` methods to print it to the console. However, if this isn't sufficient for your application, you can use the `printf` method of the `System.out` object to apply



## Two methods of the System.out object

Method	Description
<code>println(data)</code>	Prints the data argument followed by a new line character to the console.
<code>print(data)</code>	Prints the data to the console without starting a new line.

### Example 1: The println method

```
System.out.println("Welcome to the Invoice Total Calculator");
System.out.println("Total: " + total);
System.out.println(message);
System.out.println();           // print a blank line
```

### Example 2: The print method

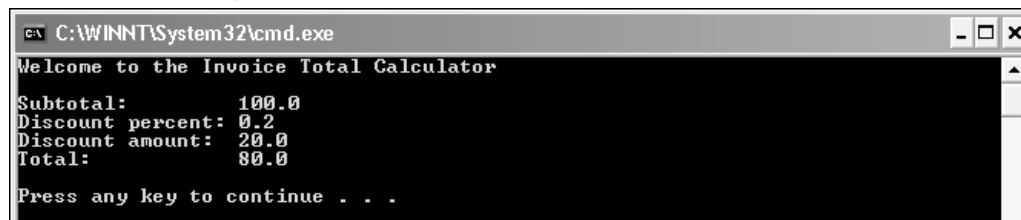
```
System.out.print("Total: ");
System.out.print(total);
System.out.print("\n");
```

### Example 3: An application that prints data to the console

```
public class InvoiceApp
{
    public static void main(String[] args)
    {
        // set and calculate the numeric values
        double subtotal = 100;           // set subtotal to 100
        double discountPercent = .2;     // set discountPercent to 20%
        double discountAmount = subtotal * discountPercent;
        double invoiceTotal = subtotal - discountAmount;

        // print the data to the console
        System.out.println("Welcome to the Invoice Total Calculator");
        System.out.println();
        System.out.println("Subtotal:      " + subtotal);
        System.out.println("Discount percent: " + discountPercent);
        System.out.println("Discount amount: " + discountAmount);
        System.out.println("Total:      " + invoiceTotal);
        System.out.println();
    }
}
```

### The console output



```
C:\WINNT\System32\cmd.exe
Welcome to the Invoice Total Calculator
Subtotal:      100.0
Discount percent: 0.2
Discount amount: 20.0
Total:      80.0
Press any key to continue . . .
```

### Description

- Although the appearance of a console may differ from one system to another, you can always use the `print` and `println` methods to print data to the console.

Figure 2-12 How to use the `System.out` object to print output to the console

the formatting you need. For more information about using this method, please download the free tutorial from our web site at [www.murach.com](http://www.murach.com).

## How to use the Scanner class to read input from the console

---

Figure 2-13 shows how you can use the Scanner class to read input from the console. To start, you create a Scanner object by using a statement like the one in this figure. Here, `sc` is the name of the scanner object that is created and `System.in` represents console input, which is the keyboard. You can code this statement this way whenever you want to get console input.

Once you've created a Scanner object, you can use the next methods to read data from the console, but the method you use depends on the type of data you need to read. To read string data, for example, you use the `next` method. To read integer data, you use the `nextInt` method. To read double data, you use the `nextDouble` method. And to read all of the data on a line, you use the `nextLine` method.

To use one of these methods, you code the object name (`sc`), a period (`.`), the method name, and a set of parentheses. Then, the method gets the next user entry. For instance, the first statement in the examples gets a string and assigns it to a string variable named `name`. The second statement gets an integer and assigns it to an `int` variable named `count`. The third statement gets a double and assigns it to a double variable named `subtotal`. And the fourth statement reads any remaining characters on the line.

Each entry that a user makes is called a *token*, and a user can enter more than one token before pressing the Enter key. To do that, the user separates the entries by one or more space, tab, or return characters. This is called *whitespace*. Then, each next method gets the next token that has been entered. If, for example, you press the Enter key (a return character), type 100, press the Tab key, type 20, and press the Enter key again, the first token is 100 and the second one is 20.

If you want to get string data that includes whitespace, you can use the `nextLine` method. If, for example, the user enters "New York" and presses the Enter key, you can use the `nextLine` method to get the entire entry as a single string.

If the user doesn't enter the type of data that the next method is looking for, an error occurs and the program ends. In Java, an error like this is also known as an *exception*. If, for example, the user enters a double value but the `nextInt` method is used to get it, an exception occurs. In chapter 5, you'll learn how to prevent this type of error.

Although this figure only shows methods for working with `String`, `int`, and `double` types, the Scanner class includes methods for working with most of the other data types that you'll learn about in the next chapter. It also includes methods that let you check what type of data the user entered. As you'll see in chapter 5, you can use these methods to avoid exceptions by checking the data type before you issue the next method.

## The Scanner class

```
java.util.Scanner
```

### How to create a Scanner object

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

### Common methods of a Scanner object

Method	Description
<code>next()</code>	Returns the next token stored in the scanner as a String object.
<code>nextInt()</code>	Returns the next token stored in the scanner as an int value.
<code>nextDouble()</code>	Returns the next token stored in the scanner as a double value.
<code>nextLine()</code>	Returns any remaining input on the current line as a String object and advances the scanner to the next line.

### How to use the methods of a Scanner object

```
String name = sc.next();
int count = sc.nextInt();
double subtotal = sc.nextDouble();
String cityName = sc.nextLine();
```

### Description

- To create a Scanner object, you use the `new` keyword. To create a Scanner object that gets input from the *console*, specify `System.in` in the parentheses.
- To use one of the methods of a Scanner object, code the object name, a dot (period), the method name, and a set of parentheses.
- When one of the next methods of the Scanner class is run, the application waits for the user to enter data with the keyboard. To complete the entry, the user presses the Enter key.
- Each entry that a user makes is called a *token*. A user can enter two or more tokens by separating them with *whitespace*, which consists of one or more spaces, a tab character, or a return character.
- The entries end when the user presses the Enter key. Then, the first `next`, `nextInt`, or `nextDouble` method gets the first token; the second `next`, `nextInt`, or `nextDouble` method gets the second token; and so on. In contrast, the `nextLine` method gets all of the input or remaining input on the current line.
- If the user doesn't enter the type of data that the next method expects, an error occurs and the program ends. In Java, this type of error is called an *exception*. You'll learn more about this in chapter 5.
- Since the Scanner class is in the `java.util` package, you'll want to include an import statement whenever you use this class.

### Note

- The Scanner class was introduced in version 1.5 of the JDK.

Figure 2-13 How to use the Scanner class to read input from the console

## Examples that get input from the console

---

Figure 2-14 presents two examples that get input from the console. The first example starts by creating a Scanner object. Then, it uses the print method of the System.out object to prompt the user for three values, and it uses the next methods of the Scanner object to read those values from the console. Because the first value should be a string, the next method is used to read this value. Because the second value should be a double, the nextDouble method is used to read it. And because the third value should be an integer, the nextInt method is used to read it.

After all three values are read, a calculation is performed using the int and double values. Then, the data is formatted and the println method is used to display the data on the console. You can see the results of this code in this figure.

Unlike the first example, which reads one value per line, the second example reads three values in a single line. Here, the first statement uses the print method to prompt the user to enter three integer values. Then, the next three statements use the nextInt method to read those three values. This works because a Scanner object uses whitespace (spaces, tabs, or returns) to separate the data that's entered at the console into tokens.

**Example 1: Code that gets three values from the user**

```
// create a Scanner object
Scanner sc = new Scanner(System.in);

// read a string
System.out.print("Enter product code: ");
String productCode = sc.next();

// read a double value
System.out.print("Enter price: ");
double price = sc.nextDouble();

// read an int value
System.out.print("Enter quantity: ");
int quantity = sc.nextInt();

// perform a calculation and display the result
double total = price * quantity;
System.out.println();
System.out.println(quantity + " " + productCode
    + " @ " + price + " = " + total);
System.out.println();
```

**The console after the program finishes****Example 2: Code that reads three values from one line**

```
// read three int values
System.out.print("Enter three integer values: ");
int i1 = sc.nextInt();
int i2 = sc.nextInt();
int i3 = sc.nextInt();

// calculate the average and display the result
int total = i1 + i2 + i3;
int avg = total / 3;
System.out.println("Average: " + avg);
System.out.println();
```

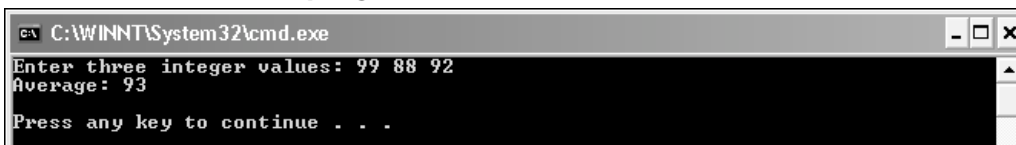
**The console after the program finishes**

Figure 2-14 Examples that get input from the console typewriter

## How to code simple control statements

---

As you write programs, you need to determine when certain operations should be done and how long repetitive operations should continue. To do that, you code *control statements* like the if/else and while statements. This topic will get you started with the use of these statements, but first you need to learn how to write expressions that compare numeric and string variables.

### How to compare numeric variables

---

Figure 2-15 shows how to code *Boolean expressions* that use the six *relational operators* to compare two primitive data types. This type of expression evaluates to either true or false based on the result of the comparison, and the operands in the expression can be either variables or literals.

For instance, the first expression in the first set of examples is true if the value of the variable named `discountPercent` is equal to the literal value 2.3. The second expression is true if the value of `subtotal` is not equal to zero. And the sixth example is true if the value of the variable named `quantity` is less than or equal to the value of the variable named `reorderPoint`.

Although you shouldn't have any trouble coding simple expressions like these, you must remember to code two equals signs instead of one for the equality comparison. That's because a single equals sign is used for assignment statements. As a result, if you try to code a Boolean expression with a single equals sign, your code won't compile.

When you compare numeric values, you usually compare values of the same data type. However, if you compare different types of numeric values, Java will automatically cast the less precise numeric type to the more precise type. For example, if you compare an `int` type to a `double` type, the `int` type will be cast to the `double` type before the comparison is made.

### How to compare string variables

---

Because a string is an object, not a primitive data type, you can't use the relational operators to compare strings. Instead, you must use the `equals` or `equalsIgnoreCase` methods of the `String` class that are summarized in figure 2-15. As you can see, both of these methods require an argument that provides the `String` object or literal that you want to compare with the current `String` object.

In the examples, the first expression is true if the value in the string named `userEntry` equals the literal value "Y". In contrast, the second expression uses the `equalsIgnoreCase` method so it's true whether the value in `userEntry` is "Y" or "y". Then, the third expression shows how you can use the not operator (!) to reverse the value of a Boolean expression that compares two strings. Here, the expression will evaluate to true if the `lastName` variable is *not* equal to "Jones". The fourth expression is true if the string variable named `code` equals the string variable named `productCode`.

## Relational operators

Operator	Name	Description
<code>==</code>	Equality	Returns a true value if both operands are equal.
<code>!=</code>	Inequality	Returns a true value if the left and right operands are not equal.
<code>&gt;</code>	Greater Than	Returns a true value if the left operand is greater than the right operand.
<code>&lt;</code>	Less Than	Returns a true value if the left operand is less than the right operand.
<code>&gt;=</code>	Greater Than Or Equal	Returns a true value if the left operand is greater than or equal to the right operand.
<code>&lt;=</code>	Less Than Or Equal	Returns a true value if the left operand is less than or equal to the right operand.

## Examples of conditional expressions

```
discountPercent == 2.3    // equal to a numeric literal
subtotal != 0             // not equal to a numeric literal
years > 0                 // greater than a numeric literal
i < months                // less than a variable
subtotal >= 500           // greater than or equal to a numeric literal
quantity <= reorderPoint // less than or equal to a variable
```

## Two methods of the String class

Method	Description
<code>equals (String)</code>	Compares the value of the String object with a String argument and returns a true value if they are equal. This method makes a case-sensitive comparison.
<code>equalsIgnoreCase (String)</code>	Works like the equals method but is not case-sensitive.

## Examples

```
userEntry.equals("Y")           // equal to a string literal
userEntry.equalsIgnoreCase("Y") // equal to a string literal
(!lastName.equals("Jones"))     // not equal to a string literal
code.equalsIgnoreCase(productCode) // equal to another string variable
```

## Description

- You can use the *relational operators* to compare two numeric operands and return a *Boolean value* that is either true or false.
- To compare two numeric operands for equality, make sure to use two equals signs. If you only use one equals sign, you'll code an assignment statement, and your code won't compile.
- If you compare an int with a double, Java will cast the int to a double.
- To test two strings for equality, you must call one of the methods of the String object. If you use the equality operator, you will get unpredictable results (more about this in chapter 4).

Figure 2-15 How to compare numeric and string variables

## How to code if/else statements

---

Figure 2-16 shows how to use the *if/else statement* (or just *if statement*) to control the logic of your applications. This statement is the Java implementation of a control structure known as the *selection structure* because it lets you select different actions based on the results of a Boolean expression.

As you can see in the syntax summary, you can code this statement with just an if clause, you can code it with one or more else if clauses, and you can code it with a final else clause. In any syntax summary, the ellipsis (...) means that the preceding element (in this case the else if clause) can be repeated as many times as it is needed. And the brackets [ ] mean that the element is optional.

When an if statement is executed, Java begins by evaluating the Boolean expression in the if clause. If it's true, the statements within this clause are executed and the rest of the if/else statement is skipped. If it's false, Java evaluates the first else if clause (if there is one). Then, if its Boolean expression is true, the statements within this else if clause are executed, and the rest of the if/else statement is skipped. Otherwise, Java evaluates the next else if clause.

This continues with any remaining else if clauses. Finally, if none of the clauses contains a Boolean expression that evaluates to true, Java executes the statements in the else clause (if there is one). However, if none of the Boolean expressions are true and there is no else clause, Java doesn't execute any statements.

If a clause only contains one statement, you don't need to enclose that statement in braces. This is illustrated by the first statement in the first example in this figure. However, if you want to code two or more statements within a clause, you need to code the statements in braces. The braces identify the block of statements that is executed for the clause.

If you declare a variable within a block, that variable is available only to the other statements in the block. This can be referred to as *block scope*. As a result, if you need to access a variable outside of the block, you should declare it before the if statement. You'll see this illustrated by the program at the end of this chapter.

When coding if statements, it's a common practice to code one if statement within another if statement. This is known as *nesting* if statements. When you nest if statements, it's a good practice to indent the nested statements and their clauses. Since this allows the programmer to easily identify where the nested statement begins and ends, this makes the code easier to read. In this figure, for example, Java only executes the nested statement if the customer type is "R". Otherwise, it executes the statements in the outer else clause.



## The syntax of the if/else statement

```
if (booleanExpression) {statements}
[else if (booleanExpression) {statements}] ...
[else {statements}]
```

### Example 1: If statements without else if or else clauses

#### With a single statement

```
if (subtotal >= 100)
    discountPercent = .2;
```

#### With a block of statements

```
if (subtotal >= 100)
{
    discountPercent = .2;
    status = "Bulk rate";
}
```

### Example 2: An if statement with an else clause

```
if (subtotal >= 100)
    discountPercent = .2;
else
    discountPercent = .1;
```

### Example 3: An if statement with else if and else clauses

```
if (customerType.equals("T"))
    discountPercent = .4;
else if (customerType.equals("C"))
    discountPercent = .2;
else if (subtotal >= 100)
    discountPercent = .2;
else
    discountPercent = .1;
```

### Example 4: Nested if statements

```
if (customerType.equals("R"))
{
    if (subtotal >= 100)                // begin nested if
        discountPercent = .2;
    else
        discountPercent = .1;
}
else
    discountPercent = .4;                // end nested if
```

## Description

- An *if/else statement*, or just *if statement*, always contains an if clause. In addition, it can contain one or more else if clauses, and a final else clause.
- If a clause requires just one statement, you don't have to enclose the statement in braces. You can just end the clause with a semicolon.
- If a clause requires more than one statement, you enclose the block of statements in braces.
- Any variables that are declared within a block have *block scope* so they can only be used within that block.

Figure 2-16 How to code if/else statements

## How to code while statements

---

Figure 2-17 shows how to code a *while statement*. This is one way that Java implements a control structure known as the *iteration structure* because it lets you repeat a block of statements. As you will see in chapter 4, though, Java also offers other implementations of this structure.

When a while statement is executed, the program repeats the statements in the block of code within the braces *while* the expression in the statement is true. In other words, the statement ends when the expression becomes false. If the expression is false when the statement starts, the statements in the block of code are never executed.

Because a while statement loops through the statements in the block as many times as needed, the code within a while statement is often referred to as a *while loop*. Here again, any variables that are defined within the block have block scope, which means that they can't be accessed outside the block.

The first example in this figure shows how to code a loop that executes a block of statements while a variable named *choice* is equal to either "y" or "Y". In this case, the statements within the block get input data from the user, process it, and display output. This is a common way to control the execution of a program, and you'll see this illustrated in detail in the next figure.

The second example shows how to code a loop that adds the numbers 1 through 4 to a variable named *sum*. Here, a *counter variable* (or just *counter*) named *i* is initialized to 1 and the *sum* variable is initialized to zero before the loop starts. Then, each time through the loop, the value of *i* is added to *sum* and one is added to *i*. When the value of *i* becomes 5, though, the expression in the while statement is no longer true and the loop ends. The use of a counter like this is a common coding practice, and single letters like *i*, *j*, and *k* are commonly used as the names of counters.

When you code loops, you must be careful to avoid *infinite loops*. If, for example, you forget to code a statement that increments the counter variable in the second example, the loop will never end because the counter will never get to 5. Then, you have to press Ctrl+C or close the console to cancel the application so you can debug your code.

## The syntax of the while loop

```
while (booleanExpression)
{
    statements
}
```

### Example 1: A loop that continues while choice is “y” or “Y”

```
String choice = "y";
while (choice.equalsIgnoreCase("y"))
{
    // get the invoice subtotal from the user
    Scanner sc = new Scanner(System.in);
    System.out.print("Enter subtotal:  ");
    double subtotal = sc.nextDouble();

    // the code that processes the user's entry goes here

    // see if the user wants to continue
    System.out.print("Continue? (y/n): ");
    choice = sc.next();
    System.out.println();
}
```

### Example 2: A loop that adds the numbers 1 through 4 to sum

```
int i = 1;
int sum = 0;
while (i < 5)
{
    sum = sum + i;
    i = i + 1;
}
```

## Description

- A *while statement* executes the block of statements within its braces as long as the Boolean expression is true. When the expression becomes false, the while statement skips its block of statements so the program continues with the next statement in sequence.
- The statements within a while statement can be referred to as a *while loop*.
- Any variables that are declared in the block of a while statement have block scope.
- If the Boolean expression in a while statement never becomes false, the statement never ends. Then, the program goes into an *infinite loop*. You can cancel an infinite loop by closing the console window or pressing Ctrl+C.

## Two illustrative applications

---

You have now learned enough about Java to write simple applications of your own. To show you how you can do that, this chapter ends by presenting two illustrative applications.

### The Invoice application

---

Figure 2-18 shows the console and code for an Invoice application. Although this application is simple, it gets input from the user, performs calculations that use this input, and displays the results of the calculations. It continues until the user enters anything other than “Y” or “y” in response to the Continue? prompt.

The Invoice application starts by displaying a welcome message at the console. Then, it creates a Scanner object named `sc` that will be used in the while loop of the program. Although this object could be created within the while loop, that would mean that the object would be recreated each time through the loop, and that would be inefficient.

Before the while statement is executed, a String object named `choice` is initialized to “y”. Then, the loop starts by getting a double value from the user and storing it in a variable named `subtotal`. After that, the loop uses an if/else statement to calculate the discount amount based on the value of `subtotal`. If, for example, `subtotal` is greater than or equal to 200, the discount amount is .2 times the subtotal (a 20% discount). If that condition isn’t true but `subtotal` is greater than or equal to 100, the discount is .1 times subtotal (a 10% discount). Otherwise, the discount amount is zero. When the if/else statement is finished, an assignment statement calculates the invoice total by subtracting `discountAmount` from `subtotal`.

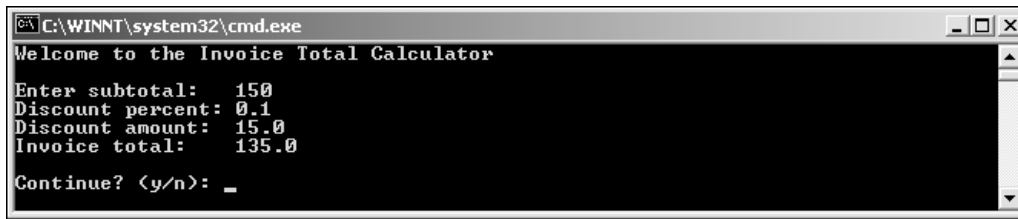
At that point, the program displays the discount percent, discount amount, and invoice total on the console. Then, it displays a message that asks the user if he or she wants to continue. If the user enters “y” or “Y”, the loop is repeated. Otherwise, the program ends.

Although this application illustrates most of what you’ve learned in this chapter, you should realize that it has a couple of shortcomings. First, the numeric values that are displayed should be formatted with two decimal places since these are currency values. In the next chapter, you’ll learn how to do that type of formatting.

Second, an exception will occur and the program will end prematurely if the user doesn’t enter one valid double value for the subtotal each time through the loop. This is a serious problem that isn’t acceptable in a professional program, and you’ll learn how to prevent problems like this in chapter 5.

In the meantime, if you’re new to programming, you can learn a lot by writing simple programs like the Invoice program. That will give you a chance to become comfortable with the coding for input, calculations, output, if/else statements, and while statements. And that will prepare you for the chapters that follow.

## The console input and output for a test run



```

C:\WINNT\system32\cmd.exe
Welcome to the Invoice Total Calculator
Enter subtotal: 150
Discount percent: 0.1
Discount amount: 15.0
Invoice total: 135.0
Continue? (y/n): _

```

## The code for the application

```

import java.util.Scanner;

public class InvoiceApp
{
    public static void main(String[] args)
    {
        // welcome the user to the program
        System.out.println("Welcome to the Invoice Total Calculator");
        System.out.println(); // print a blank line

        // create a Scanner object named sc
        Scanner sc = new Scanner(System.in);

        // perform invoice calculations until choice isn't equal to "y" or "Y"
        String choice = "y";
        while (choice.equalsIgnoreCase("y"))
        {
            // get the invoice subtotal from the user
            System.out.print("Enter subtotal: ");
            double subtotal = sc.nextDouble();

            // calculate the discount amount and total
            double discountPercent = 0.0;
            if (subtotal >= 200)
                discountPercent = .2;
            else if (subtotal >= 100)
                discountPercent = .1;
            else
                discountPercent = 0.0;
            double discountAmount = subtotal * discountPercent;
            double total = subtotal - discountAmount;

            // display the discount amount and total
            String message = "Discount percent: " + discountPercent + "\n"
                + "Discount amount: " + discountAmount + "\n"
                + "Invoice total: " + total + "\n";
            System.out.println(message);

            // see if the user wants to continue
            System.out.print("Continue? (y/n): ");
            choice = sc.next();
            System.out.println();
        }
    }
}

```

Figure 2-18 The Invoice application

## The Test Score application

---

Figure 2-19 presents another Java application that will give you more ideas for how you can apply what you've learned so far. If you look at the console input and output for this application, you can see that it lets the user enter one or more test scores. To end the application, the user enters a value of 999. Then, the application displays the number of test scores that were entered, the total of the scores, and the average of the scores.

If you look at the code for this application, you can see that it starts by displaying the instructions for using the application. Then, it declares and initializes three variables, and it creates a Scanner object that will be used to get console input.

The while loop in this program continues until the user enters a test score that's greater than 100. To start, this loop gets the next test score. Then, if that test score is less than or equal to 100, the program adds one to `scoreCount`, which keeps track of the number of scores, and adds the test score to `scoreTotal`, which accumulates the total of the scores. The if statement that does this is needed, because you don't want to increase `scoreCount` and `scoreTotal` if the user enters 999 to end the program. When the loop ends, the program calculates the average score and displays the score count, total, and average.

To include decimal places in the score average, this program declares `scoreTotal` and `averageScore` as a double data types. Declaring `scoreTotal` as a double type causes the score average to be calculated with decimal places. Declaring the `averageScore` variable as a double type allows it to store those decimal places.

To allow statements outside of the while loop to access the `scoreTotal` and `scoreCount` variables, this program declares these variables before the while loop. If these variables were declared inside the while loop, they would only be available within that block of code and couldn't be accessed by the statements that are executed after the while loop. In addition, the logic of the program wouldn't work because these variables would be reinitialized each time through the loop.

Here again, this program has some obvious shortcomings that will be addressed in later chapters. First, the data isn't formatted properly, but you'll learn how to fix that in the next chapter. Second, an exception will occur and the program will end prematurely if the user enters invalid data, but you'll learn how to fix that in chapter 5.

## The console input and output for a test run

```

C:\WINNT\system32\cmd.exe
Please enter test scores that range from 0 to 100.
To end the program enter 999.

Enter score: 90
Enter score: 80
Enter score: 75
Enter score: 999

Score count:    3
Score total:    245.0
Average score:  81.66666666666667

Press any key to continue . . . _

```

## The code for the application

```

import java.util.Scanner;

public class TestScoreApp
{
    public static void main(String[] args)
    {
        // display operational messages
        System.out.println(
            "Please enter test scores that range from 0 to 100.");
        System.out.println("To end the program enter 999.");
        System.out.println(); // print a blank line

        // initialize variables and create a Scanner object
        double scoreTotal = 0.0;
        int scoreCount = 0;
        int testScore = 0;
        Scanner sc = new Scanner(System.in);

        // get a series of test scores from the user
        while (testScore <= 100)
        {
            // get the input from the user
            System.out.print("Enter score: ");
            testScore = sc.nextInt();

            // accumulate score count and score total
            if (testScore <= 100)
            {
                scoreCount = scoreCount + 1;
                scoreTotal = scoreTotal + testScore;
            }
        }

        // display the score count, score total, and average score
        double averageScore = scoreTotal / scoreCount;
        String message = "\n"
            + "Score count:    " + scoreCount + "\n"
            + "Score total:    " + scoreTotal + "\n"
            + "Average score: " + averageScore + "\n";
        System.out.println(message);
    }
}

```

Figure 2-19 The Test Score application

## How to test and debug an application

---

In chapter 1, you were introduced to the compile-time errors that can occur when you compile an application (see figure 1-11). Once you've fixed those errors, you're ready to test and debug the application as described in this topic. Then, in the next two chapters, you'll learn several more debugging techniques. And when you do the exercises, you'll get lots of practice testing and debugging.

### How to test an application

---

When you *test* an application, you run it to make sure the application works correctly. As you test, you should try every possible combination of valid and invalid data to be certain that the application works correctly under every set of conditions. Remember that the goal of testing is to find errors, or *bugs*, not to show that an application program works correctly.

As you test, you will encounter two types of bugs. The first type of bug causes a *runtime error*. (In Java, this type of error is also known as a *runtime exception*.) A runtime error causes the application to end prematurely, which programmers often refer to as “crashing” or “blowing up.” In this case, an error message like the one in figure 2-20 is displayed, and this message shows the line number of the statement that was being executed when the crash occurred.

The second type of bug produces inaccurate results when the application runs. These bugs occur due to *logical errors* in the source code. For instance, the second example in this figure shows the output for the Test Score application. In this case, the final totals were displayed and the application ended before the user entered any input data. This type of bug can be more difficult to find and correct than a runtime error.

### How to debug an application

---

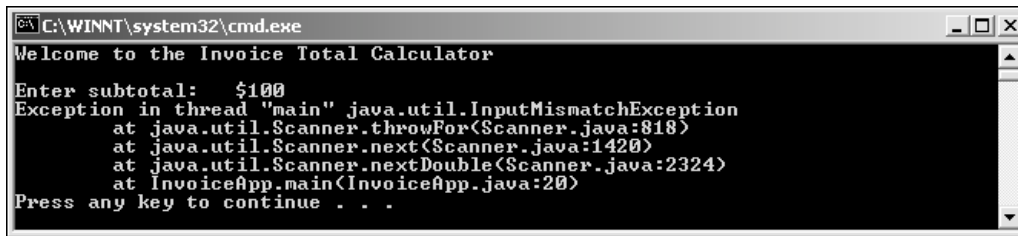
When you *debug* a program, you find the cause of the bugs, fix them, recompile, and test again. As you progress through this book and your programs become more complex, you'll see that debugging can be one of the most time-consuming aspects of programming.

To find the cause of runtime errors, you can start by finding the source statement that was running when the program crashed. You can usually do that by studying the error message that's displayed. In the first console in this figure, for example, you can see that the statement at line 20 was running when the program crashed. That's the statement that used the `nextDouble` method of the `Scanner` object, and that indicates that the problem is invalid input data. For now, you can ignore this bug. In chapter 5, you'll learn how to fix it.

To find the cause of incorrect output, you can start by figuring out why the application produced the output that it did. For instance, you can start by asking why the second application in this figure didn't prompt the user to enter any test scores. Once you figure that out, you're well on your way to fixing the bug.

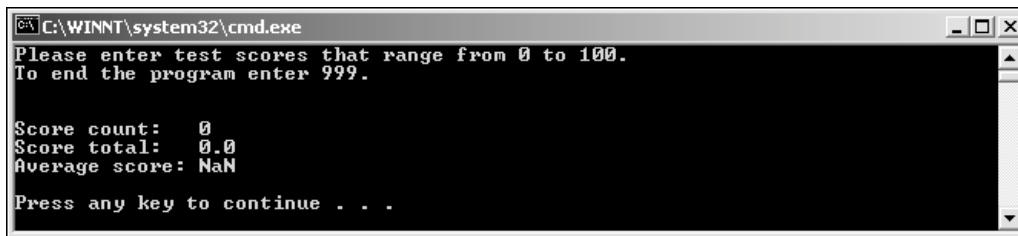


## A runtime error that occurred while testing the Invoice application



```
C:\WINNT\system32\cmd.exe
Welcome to the Invoice Total Calculator
Enter subtotal: $100
Exception in thread "main" java.util.InputMismatchException
    at java.util.Scanner.throwFor(Scanner.java:818)
    at java.util.Scanner.next(Scanner.java:1420)
    at java.util.Scanner.nextDouble(Scanner.java:2324)
    at InvoiceApp.main(InvoiceApp.java:20)
Press any key to continue . . .
```

## Incorrect output produced by the Test Score application



```
C:\WINNT\system32\cmd.exe
Please enter test scores that range from 0 to 100.
To end the program enter 999.

Score count: 0
Score total: 0.0
Average score: NaN
Press any key to continue . . .
```

## Debugging tips

- For a runtime error, go to the line in the source code that was running when the program crashed. That should give you a strong indication of what caused the error.
- For incorrect output, first figure out how the source code produced that output. Then, fix the code and test the application again.

## Description

- To *test* an application, you run it to make sure that it works properly no matter what combinations of valid and invalid data you enter. The goal of testing is to find the errors (or *bugs*) in the application.
- To *debug* an application, you find the causes of the bugs and fix them.
- One type of bug leads to a *runtime error* (also known as a *runtime exception*) that causes the program to end prematurely. This type of bug must be fixed before testing can continue.
- Even if an application runs to completion, the results may be incorrect due to *logical errors*. These bugs must also be fixed.

## Perspective

---

The goal of this chapter has been to get you started with Java programming and to get you started fast. Now, if you understand how the Invoice and Test Score applications in figures 2-18 and 2-19 work, you've come a long way. You should also be able to write comparable programs of your own.

Keep in mind, though, that this chapter is just an introduction to Java programming. So in the next chapter, you'll learn the details about working with data. In chapter 4, you'll learn the details about using control statements. And in chapter 5, you'll learn how to prevent and handle runtime exceptions.

## Summary

---

- The *statements* in a Java program direct the operation of the program. The *comments* document what the program does.
- You must code at least one public *class* for every Java program that you write. The *main method* of this class is executed when you run the class.
- *Variables* are used to store data that changes as a program runs, and you use *assignment statements* to assign values to variables. Two of the most common *data types* for numeric variables are the *int* and *double* types.
- A *string* is an object that's created from the *String* class, and it can contain any characters in the character set. You can use the plus sign to *join* a string with another string or a data type, and you can use assignment statements to *append* one string to another. To include special characters in strings, you can use *escape sequences*.
- Before you use many of the classes in the Java API, you should code an *import* statement for the class or for the *package* that contains it.
- When you use a *constructor* to create an *object* from a Java class, you are creating an *instance* of the class. There may be more than one constructor for a class, and a constructor may require one or more *arguments*.
- You *call* a *method* from an object and you call a *static method* from a class. A method may require one or more arguments.
- One of the most time-consuming aspects of Java programming is researching the classes and methods that your programs require.
- You can use the methods of a *Scanner* object to read data from the *console*, and you can use the *print* and *println* methods of the *System.out* object to print data to the console.
- You can code *if statements* to control the logic of a program based on the true or false values of *Boolean expressions*. You can code *while statements* to repeat a series of statements until a Boolean expression becomes false.
- *Testing* is the process of finding the errors or bugs in an application. *Debugging* is the process of fixing the bugs.

## Before you do the exercises for this chapter

If you didn't do it already, you should install and configure the JDK, the Java API documentation, and TextPad or an equivalent text editor as described in chapter 1. You also need to download and install the folders and files for this book from our web site ([www.murach.com](http://www.murach.com)) before you start the exercises for this chapter. For complete instructions, please refer to appendix A.

### Exercise 2-1 Test the Invoice application

In this exercise, you'll compile and test the Invoice application that's presented in figure 2-18. That will give you a better idea of how this program works.

1. Start your text editor and open the file named `InvoiceApp.java` that you should find in the `c:\java1.6\ch02` directory. Then, compile the application, which should compile with no errors.
2. Test this application with valid subtotal entries like 50, 150, 250, and 1000 so it's easy to see whether or not the calculations are correct.
3. Test the application with a subtotal value like 233.33. This will show that the application doesn't round the results to two decimal places. But in the next chapter, you'll learn how to do that.
4. Test the application with an invalid subtotal value like \$1000. This time, the application should crash. Study the error message that's displayed and determine which line of source code was running when the error occurred.
5. Restart the application, enter a valid subtotal, and enter 20 when the program asks you whether you want to continue. What happens and why?
6. Restart the application and enter two values separated by whitespace (like 1000 20) before pressing the Enter key. What happens and why?

### Exercise 2-2 Modify the Test Score application

In this exercise, you'll modify the Test Score application that's presented in figure 2-19. That will give you a chance to write some code of your own.

1. Open the file named `TestScoreApp.java` in the `c:\java1.6\ch02` directory, and save the program as `ModifiedTestScoreApp.java` in the same directory. Then, change the class name to `ModifiedTestScoreApp` and compile the class.
2. Test this application with valid data to see how it works. Then, test the application with invalid data to see what will cause exceptions. Note that if you enter a test score like 125, the program ends, even though the instructions say that the program ends when you enter 999.
3. Modify the while statement so the program only ends when you enter 999. Then, test the program to see how this works.
4. Modify the if statement so it displays an error message like "Invalid entry, not counted" if the user enters a score that's greater than 100 but isn't 999. Then, test this change.

## Exercise 2-3 Modify the Invoice application

In this exercise, you'll modify the Invoice application. When you're through with the modifications, a test run should look something like this:

```
C:\WINNT\system32\cmd.exe
Welcome to the Invoice Total Calculator

Enter subtotal: 100
Discount percent: 0.1
Discount amount: 10.0
Invoice total: 90.0

Continue? <y/n>: 0

Enter subtotal: 500
Discount percent: 0.25
Discount amount: 125.0
Invoice total: 375.0

Continue? <y/n>: n

Number of invoices: 2
Average invoice: 232.5
Average discount: 67.5

Press any key to continue . . .
```

1. Open the file named InvoiceApp.java that's in the c:\java1.6\ch02 directory, and save the program as ModifiedInvoiceApp.java in the same directory. Then, change the class name to ModifiedInvoiceApp.
2. Modify the code so the application ends only when the user enters "n" or "N". As it is now, the application ends when the user enters anything other than "y" or "Y". To do this, you need to use a not operator (!) with the equalsIgnoreCase method. This is illustrated by the third example in figure 2-15. Then, compile this class and test this change by entering 0 at the Continue? prompt.
3. Modify the code so it provides a discount of 25 percent when the subtotal is greater than or equal to \$500. Then, test this change.
4. Using the Test Score application as a model, modify the Invoice program so it displays the number of invoices, the average invoice amount, and the average discount amount when the user ends the program. Then, test this change.

## Exercise 2-4 Use the Java API documentation

This exercise steps you through the Java API documentation for the `Scanner`, `String`, and `Double` classes. That will give you a better idea of how extensive the Java API is.

1. Go to the index page of the Java API documentation as described in chapter 1. If you did the exercises for that chapter, you should have it bookmarked.
2. Click the `java.util` package in the upper left window and the `Scanner` class in the lower left window to display the documentation for the `Scanner` class. Then, scroll through this documentation to get an idea of its scope.
3. Review the constructors for the `Scanner` class. The constructor that's presented in this chapter has just an `InputStream` object as its argument. When you code that argument, remember that `System.in` represents the `InputStream` object for the console.
4. Review the methods of the `Scanner` class with special attention to the `next`, `nextInt`, and `nextDouble` methods. Note that there are three `next` methods and two `nextInt` methods. The ones used in this chapter have no arguments. Then, review the `has` methods in the `Scanner` class. You'll learn how to use some of these in chapter 5.
5. Go to the documentation for the `String` class, which is in the `java.lang` package, and note that it offers a number of constructors. In this chapter, though, you learned the shortcut for creating `String` objects because that's the best way to do that. Now, review the methods for this class with special attention to the `equals` and `equalsIgnoreCase` methods.
6. Go to the documentation for the `Double` class, which is also in the `java.lang` package. Then, review the static `parseDouble` and `toString` methods that you'll learn how to use in the next chapter.

If you find the documentation difficult to follow, rest assured that you'll become comfortable with it before you finish this book. Once you learn how to create your own classes, constructors, and methods, it will make more sense.



# 3

## How to work with data

In chapter 2, you learned how to use two of the eight primitive data types as you declared and initialized variables and coded assignment statements that used simple arithmetic expressions. Now, you'll learn all of the details that you need for working with variables and data types at a professional level.

<b>Basic skills for working with data .....</b>	<b>92</b>
The eight primitive data types .....	92
How to initialize variables .....	94
How to initialize constants .....	94
How to code assignment statements and arithmetic expressions .....	96
How to use the shortcut assignment operators .....	98
How to work with the order of precedence .....	100
How to work with casting .....	102
<b>How to use Java classes for working with data types ....</b>	<b>104</b>
How to use the NumberFormat class .....	104
How to use the Math class .....	106
How to use the Integer and Double classes .....	108
<b>The formatted Invoice application .....</b>	<b>110</b>
The code for the application .....	110
How to analyze the data problems in the Invoice application .....	112
<b>How to use the BigDecimal class for working with decimal data .....</b>	<b>114</b>
The constructors and methods of the BigDecimal class .....	114
How to use BigDecimal arithmetic in the Invoice application .....	116
<b>Perspective .....</b>	<b>118</b>

## Basic skills for working with data

---

In this topic, you'll learn about the six primitive data types that weren't presented in chapter 2. Then, you'll learn the fundamentals for working with all of the data types.

### The eight primitive data types

---

Figure 3-1 shows the eight *primitive data types* provided by Java. You can use these eight data types to store six types of numbers, characters, and true or false values.

In chapter 2, you learned how to use the `int` data type for storing *integers* (whole numbers). But as this figure shows, you can also use three other data types for integers. Most of the time, you can use the `int` type for working with integers, but you may need to use the *long* type if a value is too big for the `int` type. Although the use of the *short* and *byte* types is less common, you can use them when you're working with smaller integers and you need to save system resources.

In chapter 2, you also learned how to use the `double` data type for storing numbers with decimal places. But as this figure shows, you can also use the *float* data type for those numbers. The values in both of these data types are stored as *floating-point numbers* that can hold very large and very small values, but with a limited number of *significant digits*. For instance, the `double` type with its 16 significant digits provides for numbers like 12,345,678,901,234.56 or 12,345,678.90123456 or 12.34567890123456. Since the `double` type has more significant digits than the `float` type, you'll use the `double` type for most floating-point numbers.

To express the value of a floating-point number, you can use *scientific notation*. This lets you express very large and very small numbers in a sort of shorthand. To use this notation, you type the letter *e* or *E* followed by a power of 10. For instance, 3.65e+9 is equal to 3.65 times 10<sup>9</sup> (or 3,650,000,000), and 3.65e-9 is equal to 3.65 times 10<sup>-9</sup> (or .00000000365).

You can use the *char* type to store one character. Since Java uses the two-byte *Unicode character set*, it can store practically any character from any language around the world. As a result, you can use Java to create programs that read and print Greek or Chinese characters. In practice, though, you'll usually work with the characters that are stored in the older one-byte *ASCII character set*. These characters are the first 256 characters of the Unicode character set.

Last, you can use the *boolean* type to store a true value or a false value. This data type is typically used to represent a condition that can be true or false.



## The eight primitive data types

Type	Bytes	Use
byte	1	Very short integers from -128 to 127.
short	2	Short integers from -32,768 to 32,767.
int	4	Integers from -2,147,483,648 to 2,147,483,647.
long	8	Long integers from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807.
float	4	Single-precision, floating-point numbers from -3.4E38 to 3.4E38 with up to 7 significant digits.
double	8	Double-precision, floating-point numbers from -1.7E308 to 1.7E308 with up to 16 significant digits.
char	2	A single Unicode character that's stored in two bytes.
boolean	1	A <i>true</i> or <i>false</i> value.

### Description

- A *bit* is a binary digit that can have a value of one or zero. A *byte* is a group of eight bits. As a result, the number of bits for each data type is the number of bytes multiplied by 8.
- *Integers* are whole numbers, and the first four data types above provide for integers of various sizes.
- *Floating-point numbers* provide for very large and very small numbers that require decimal positions, but with a limited number of *significant digits*. A *single-precision number* provides for numbers with up to 7 significant digits. A *double-precision number* provides for numbers with up to 16 significant digits.
- The *double* data type is commonly used for business programs because it provides the precision (number of significant digits) that those programs require.
- The *Unicode character set* provides for over 65,000 characters with two bytes used for each character.
- The older *ASCII character set* that's used by most operating systems provides for 256 characters with one byte used for each character. In the Unicode character set, the first 256 characters correspond to the 256 ASCII characters.
- A *boolean* data type holds a *true* or *false* value.

### Technical notes

- To express the value of a floating-point number, you can use *scientific notation* like 2.382E+5, which means 2.382 times  $10^5$  (a value of 238,200), or 3.25E-8, which means 3.25 times  $10^{-8}$  (a value of .0000000325). Java will sometimes use this notation to display the value of a float or double data type.
- Because of the way floating-point numbers are stored internally, they can't represent the exact value of the decimal places in some numbers. This can cause a rounding problem in some business applications. Later in this chapter, you'll learn how to use the `BigDecimal` class to solve these rounding problems.

Figure 3-1 The eight primitive data types

## How to initialize variables

---

In chapter 2, you learned how to *declare* and *initialize* a *variable*. This information is repeated in figure 3-2, but with some new information. In particular, it shows how to use separate statements to declare and initialize the variable. It also shows how to declare and initialize some of the data types that weren't presented in chapter 2.

Although you usually declare and initialize a variable in one statement, it occasionally makes sense to do it in two. For instance, you may want to declare a variable at the start of a coding routine without giving it a starting value because its value won't be set until later on.

The one-statement examples in this figure show how to declare and initialize various types of variables. Here, the third and fourth examples show how to assign values to the float and long types. To do that, you need to add a letter after the value. For a float type, you add an *f* or *F* after the value. For a long type, you add an *L*. You can also use a lowercase *l*, but it's not a good coding practice since the lowercase *l* can easily be mistaken for the number 1. If you omit the letter in one of these assignments, you'll get a compile-time error.

The fifth statement shows how you can use scientific notation as you assign a value to a variable. Then, the sixth and seventh examples show that you can assign a character to the char type by enclosing a character in single quotes or by supplying the integer that corresponds to the character in the Unicode character set. And the eighth example shows how to initialize a variable named *valid* as a boolean type with a false value.

The last example shows that you can declare and initialize two or more variables in a single statement. Although you may occasionally want to do this, it's usually better to declare and initialize one variable per statement. That way, it's easier to read your code and to modify it later on.

## How to initialize constants

---

A *constant* is used in a Java program to store a value that can't be changed as the program executes, and many of the skills for initializing variables also apply to initializing constants. However, you begin the initialization statement for a constant with the *final* keyword. As a result, constants are sometimes called *final variables*. In addition, it's a common coding convention to use all uppercase letters for the name of a constant and to separate the words in the name with an underscore.

## How to initialize a variable in two statements

### Syntax

```
type variableName;
variableName = value;
```

### Example

```
int counter;           // declaration statement
counter = 1;           // assignment statement
```

## How to initialize a variable in one statement

### Syntax

```
type variableName = value;
```

### Examples

```
int counter = 1;           // initialize an int variable
double price = 14.95;      // initialize a double variable
float interestRate = 8.125F; // F indicates a floating-point value
long numberOfBytes = 20000L; // L indicates a long integer
double distance = 3.65e+9;  // scientific notation
char letter = 'A';         // stored as a two-digit Unicode character
char letter = 65;          // integer value for a Unicode character
boolean valid = false;     // where false is a keyword
int x = 0, y = 0;          // initialize 2 variables with 1 statement
```

## How to initialize a constant

### Syntax

```
final type CONSTANT_NAME = value;
```

### Examples

```
final int DAYS_IN_NOVEMBER = 30;
final double SALES_TAX = .075;
```

## Description

- A *variable* stores a value that can change as a program executes, while a *constant* stores a value that can't be changed.
- To *initialize* a variable or constant, you *declare* its data type and *assign* an initial value. As default values, it's common to initialize integer variables to 0, floating-point variables to 0.0, and boolean variables to false.
- To initialize more than one variable for a single data type in a single statement, use commas to separate the assignments.
- To identify float values, you must type an *f* or *F* after the number. To identify long values, you must type an *l* or *L* after the number.

## Naming conventions

- Start variable names with a lowercase letter and capitalize the first letter in all words after the first word.
- Capitalize all of the letters in constants and separate words with underscores.
- Try to use meaningful names that are easy to remember as you code.

---

Figure 3-2 How to initialize variables and constants

## How to code assignment statements and arithmetic expressions

---

In chapter 2, you learned how to code *assignment statements* that used simple *arithmetic expressions* to assign the value of the expression to a variable. These expressions used the first four *arithmetic operators* in figure 3-3. Now, this figure summarizes all of the other Java arithmetic operators. These operators indicate what operations are to be performed on the *operands* in the expression, which can be either *literals* or variables.

In this figure, the first five operators work on two operands. As a result, they're referred to as *binary operators*. For example, when you use the subtraction operator (-), you subtract one operand from another.

In contrast, the last four operators work on one operand. As a result, they're referred to as *unary operators*. For example, you can code the negative sign operator (-) in front of an operand to reverse the value of the operand. Although you can also code the positive sign operator (+) in front of an operand, it doesn't change the value of the operand so it's rarely used as a unary operator.

While the addition (+), subtraction (-), and multiplication (\*) operators are easy to understand, the division (/) and modulus (%) operators are more difficult. If you're working with integer data types, the division operator returns an integer value that represents the number of times the right operand will fit into the left operand. Then, the modulus operator returns an integer value that represents the remainder (which is the amount that's left over after dividing the right operand by the left operand). However, if you're working with non-integer data types, the division operator returns a value that uses decimal places to indicate the result of the division, and that's usually what you want.

When you code an increment (++) or decrement (--) operator, you can *prefix* the operand by coding the operator before the variable. Then, the increment or decrement operation is performed before the rest of the statement is executed. Conversely, you can *postfix* the operand by coding the operator after the variable. Then, the increment or decrement operation isn't performed until after the statement is executed.

Often, an entire statement does nothing more than increment a variable like this:

```
counter++;
```

Then, both the prefix and postfix forms will yield the same result. However, if you use the increment and decrement operators as part of a larger statement, you'll need to use the prefix and postfix forms of these operators to control when the operation is performed. More about that in a moment.

Since each char type is a Unicode character that has a numeric code that maps to an integer, you can perform some integer operations on char types. For instance, this figure shows an example of how you can use the increment operator to change the numeric value for a char variable from 67 to 68, which changes the character from *C* to *D*.

## Arithmetic operators

Operator	Name	Description
+	Addition	Adds two operands.
-	Subtraction	Subtracts the right operand from the left operand.
*	Multiplication	Multiplies the right operand and the left operand.
/	Division	Divides the right operand into the left operand. If both operands are integers, then the result is an integer.
%	Modulus	Returns the value that is left over after dividing the right operand into the left operand.
++	Increment	Adds 1 to the operand ( $x = x + 1$ ).
--	Decrement	Subtracts 1 from the operand ( $x = x - 1$ ).
+	Positive sign	Promotes byte, short, and char types to the int type.
-	Negative sign	Changes a positive value to negative, and vice versa.

## Examples of simple assignment statements

```

int x = 14;
int y = 8;
int result1 = x + y;           // result1 = 22
int result2 = x - y;           // result2 = 6
int result3 = x * y;           // result3 = 112
int result4 = x / y;           // result4 = 1
int result5 = x % y;           // result5 = 6
int result6 = -y + x;          // result6 = 6
int result7 = --y;             // result7 = 7
int result8 = ++x;             // result8 = 15, x = 15

double a = 8.5;
double b = 3.4;
double result9 = a + b;        // result9 = 11.9
double result10 = a - b;       // result10 = 5.1
double result11 = a * b;       // result11 = 28.90
double result12 = a / b;       // result12 = 2.5
double result13 = a % b;       // result13 = 1.7
double result14 = -a + b;      // result14 = -5.1
double result15 = --a;         // result15 = 7.5
double result16 = ++b;         // result16 = 4.4

// character arithmetic
char letter1 = 'C';            // letter1 = 'C'  Unicode integer is 67
char letter2 = ++letter1;      // letter2 = 'D'  Unicode integer is 68

```

## Description

- An *arithmetic expression* consists of *operands* and *arithmetic operators*. The first five operators above are called *binary operators* because they operate on two operands. The next four are called *unary operators* because they operate on just one operand.
- An *assignment statement* consists of a variable, an equals sign, and an expression. When the assignment statement is executed, the value of the expression is determined and the result is stored in the variable.

Figure 3-3 How to code arithmetic expressions and assignment statements

## How to use the shortcut assignment operators

---

When coding assignment statements, it's common to code the same variable on both sides of the equals sign. This is illustrated by the first group of statements in figure 3-4. That way, you can use the current value of the variable in an expression and update the variable by assigning the result of the expression to it. You saw this illustrated in chapter 2.

Since it's common to write statements like this, the Java language provides the five shorthand *assignment operators* shown in this figure. Although these operators don't provide any new functionality, you can use them to write shorter code.

If, for example, you need to increment or decrement a variable by a value of 1, you can use a shortcut operator. For example:

```
month = month + 1;
```

can be coded with a shortcut operator as

```
month += 1;
```

which is equivalent to

```
month++;
```

Similarly, if you want to add the value of a variable named `nextNumber` to a summary field named `sum`, you can do it like this:

```
sum += nextNumber;
```

which is equivalent to

```
sum = sum + nextNumber;
```

The techniques that you use are mostly a matter of preference because the code is easy to read and maintain either way you code it.

## Assignment operators

Operator	Name	Description
=	Assignment	Assigns a new value to the variable.
+=	Addition	Adds the operand to the starting value of the variable and assigns the result to the variable.
-=	Subtraction	Subtracts the operand from the starting value of the variable and assigns the result to the variable.
*=	Multiplication	Multiplies the operand by the starting value of the variable and assigns the result to the variable.
/=	Division	Divides the starting value of the variable by the operand and assigns the result to the variable. If the operand and the value of the variable are both integers, the result is an integer.
%=	Modulus	Derives the value that is left over after dividing the right operand by the value in the variable, and then assigns this value to the variable.

### Statements that use the same variable on both sides of the equals sign

```
count = count + 1;           // count is increased by 1
count = count - 1;          // count is decreased by 1
total = total + 100.0;       // total is increased by 100.0
total = total - 100.0;       // total is decreased by 100
price = price * .8;          // price is multiplied by .8
sum = sum + nextNumber;      // sum is increased by value of nextNumber
```

### Statements that use the shortcut operators to get the same results

```
count += 1;                  // count is increased by 1
count -= 1;                  // count is decreased by 1
total += 100.0;              // total is increased by 100.0
total -= 100.0;              // total is decreased by 100.0
price *= .8;                 // price is multiplied by .8
sum += nextNumber;           // sum is increased by the value of nextNumber
```

### Description

- Besides the equals sign, Java provides for the five other *assignment operators* shown above. These operators provide a shorthand way to code common assignment operations.

## How to work with the order of precedence

---

Figure 3-5 gives more information about coding arithmetic expressions. In particular, it gives the *order of precedence* of the arithmetic operations. This means that all of the prefixed increment and decrement operations in an expression are done first, followed by all of the positive and negative operations, and so on. If there are two or more operations at the same order of precedence, the operations are done from left to right.

Because this sequence of operations doesn't always work the way you want it to, you may need to override the sequence by using parentheses. Then, the expressions in the innermost sets of parentheses are done first, followed by the next sets of parentheses, and so on. Within the parentheses, though, the operations are done left to right by the order of precedence. In general, you should use parentheses to dictate the sequence of operations whenever there's any doubt about it.

The need for parentheses is illustrated by the first example in this figure. Because parentheses aren't used in the first expression that calculates the price, the multiplication operation is done before the subtraction operation, which gives an incorrect result. In contrast, because the subtraction operation is enclosed in parentheses in the second expression, this operation is performed before the multiplication operation, which gives a correct result.

The second example in this figure shows how parentheses can be used in a more complicated expression. Here, three sets of parentheses are used to calculate the current value of an investment account after a monthly investment amount is added to it, monthly interest is calculated, and the interest is added to it. If you have trouble following this, you can plug the initial values into the expression and evaluate it one set of parentheses at a time:

```
(5000 + 100) * (1 + (.12 / 12))
(5000 + 100) * (1 + .01)
(5100 * 1.01)
5151
```

If you have trouble creating an expression like this for a difficult calculation, you can often code it in a more direct way. To illustrate, this figure shows another way to calculate the current value. Here, the first statement adds the monthly investment amount to the current value. The second statement calculates the interest. And the third statement adds the interest to the current value. This not only takes away the need for parentheses, but also makes it easier to follow what's going on.

The third example in this figure shows the differences between the use of prefixed and postfix increment and decrement operators. With prefixed operators, the variable is incremented or decremented before the result is assigned. With postfix operators, the result is assigned before the operations are done. Because this can get confusing, it's best to limit these operators to simple expressions.



## The order of precedence for arithmetic operations

1. Increment and decrement
2. Positive and negative
3. Multiplication, division, and remainder
4. Addition and subtraction

### Example 1: A calculation that uses the default order of precedence

```
double discountPercent = .2;           // 20% discount
double price = 100;                   // $100 price
price = price * 1 - discountPercent;   // price = $99.8
```

### The same calculation with parentheses that specify the order of precedence

```
price = price * (1 - discountPercent); // price = $80
```

### Example 2: An investment calculation based on a monthly investment and yearly interest rate

```
double currentValue = 5000;           // current value of investment account
double monthlyInvestment = 100;       // amount added each month
double interestRate = .12;            // yearly interest rate
currentValue = (currentValue + monthlyInvestment) * (1 + (interestRate/12));
// currentValue = 5100 * 1.01 = 5151
```

### Another way to calculate the current value of the investment account

```
currentValue += monthlyInvestment;    // add investment
// calculate interest
double monthlyInterest = currentValue * interestRate / 12;
currentValue += monthlyInterest;      // add interest
```

### Example 3: Prefixed and postfix increment and decrement operators

```
int a = 5;
int b = 5;
int y = ++a;    // a = 6, y = 6
int z = b++;    // b = 6, z = 5
```

## Description

- Unless parentheses are used, the operations in an expression take place from left to right in the *order of precedence*.
- To specify the sequence of operations, you can use parentheses. Then, the operations in the innermost sets of parentheses are done first, followed by the operations in the next sets, and so on.
- When you use an increment or decrement operator as a *prefix* to a variable, the variable is incremented or decremented and then the result is assigned. But when you use an increment or decrement operator as a *postfix* to a variable, the result is assigned and then the variable is incremented or decremented.

## How to work with casting

---

As you develop Java programs, you'll frequently need to convert data from one data type to another. To do that, you use a technique called *casting*, which is summarized in figure 3-6.

As you can see, Java provides for two types of casting. *Implicit casts* are performed automatically and can be used to convert data with a less precise type to a more precise type. This is called a *widening conversion* because the new type is always wide enough to hold the original value. For instance, the first statement in this figure causes an integer value to be converted to a double value.

Java will also perform an implicit cast on the values in an arithmetic expression if some of the values have more precise data types than other values. This is illustrated by the next three statements in this figure. Here, the variables *d*, *i*, and *j* are used in an arithmetic expression. Notice that *d* is declared with the double data type, while *i* and *j* are declared with the int data type. Because of that, both *i* and *j* will be converted to double values when this expression is evaluated.

A *narrowing conversion* is one that casts data from a more precise data type to a less precise data type. With this type of conversion, the less precise data type may not be wide enough to hold the original value. In that case, you must use an *explicit cast*.

To perform an explicit cast, you code the data type in parentheses before the variable that you want to convert. When you do this, you should realize that you may lose some information. This is illustrated by the first example in this figure that performs an explicit cast. Here, a double value of 93.75 is cast to an int value of 93. An explicit cast is required in this example, however, because Java won't automatically cast a double value to an integer value since an integer value is less precise.

When you use explicit casting in an arithmetic expression, the casting is done before the arithmetic operations. This is illustrated by the last two examples of explicit casts. In the last example, two integer types are cast to double types before the division is done so the result will have decimal places if they are needed. Without explicit casting, the expression would return an integer value that would then be cast to a double.

When you code an explicit cast, an exception may occur at runtime if the JRE isn't able to perform the cast. As a result, you should use an explicit cast only when you're sure that the JRE will be able to perform the cast.

Although you typically cast between numeric data types, you can also cast between the int and char types. That's because every char value corresponds to an int value that identifies it in the Unicode character set. Since there's no possible loss of data, you can implicitly cast between these data types. However, if you prefer, you can also code these casts explicitly.

## How implicit casting works

### Casting from less precise to more precise data types

byte→short→int→long→float→double

#### Examples

```
double grade = 93;                // convert int to double

double d = 95.0;
int i = 86, j = 91;
double average = (d+i+j)/3;      // convert i and j to double values
                                // average = 90.666666...
```

## How you can code an explicit cast

### Syntax

(type) expression

#### Examples

```
int grade = (int) 93.75;          // convert double to int (grade = 93)

double d = 95.0;
int i = 86, j = 91;
double average = ((int)d+i+j)/3;  // convert d to int value (average = 90)

double result = (double) i / (double) j;    // result has decimal places
```

## How to cast between char and int types

```
char letterChar = 65;            // convert int to char (letterChar = 'A')
char letterChar2 = (char) 65;    // this works too
int letterInt = 'A';             // convert char to int (letterInt = 65)
int letterInt2 = (int) 'A';      // this works too
```

## Description

- If you assign a less precise data type to a more precise data type, Java automatically converts the less precise data type to the more precise data type. This can be referred to as an *implicit cast* or a *widening conversion*.
- When you code an arithmetic expression, Java implicitly casts the less precise data types to the most precise data type.
- To code an assignment statement that assigns a more precise data type to a less precise data type, you must use parentheses to specify the less precise data type. This can be referred to as an *explicit cast* or a *narrowing conversion*.
- You can also use an explicit cast in an arithmetic expression. Then, the casting is done before the arithmetic operations.
- Since each char value has a corresponding int value, you can implicitly or explicitly cast between these types.

## How to use Java classes for working with data types

---

As you learned in chapter 2, Java provides hundreds of classes that provide methods that you can use in your programs. Now, you'll learn about four of the classes that you'll use often when working with data types.

### How to use the `NumberFormat` class

---

When you use numeric values in a program, you often need to format them. For example, you may want to apply a standard currency format to a double value. To do that, you need to add a dollar sign and commas and to display just two decimal places. Similarly, you may want to display a double value in a standard percentage format. To do that, you need to add a percent sign and move the decimal point two digits to the right.

To do this type of formatting, Java provides the `NumberFormat` class, which is summarized in figure 3-7. Since this class is part of the `java.text` package, you'll usually want to include an import statement for this class before you begin working with it.

Once you import this class, you can call one of its static methods to return a `NumberFormat` object. As you learned in chapter 2, you can call static methods directly from a class. In other words, you code the name of the class, followed by the dot operator, followed by the method. For instance, the first example calls the static `getCurrencyInstance` method directly from the `NumberFormat` class.

Once you use a static method to return a `NumberFormat` object, you can call non-static methods from that object. To do that, you code the name of the object, followed by the dot operator, followed by the method. For instance, the first example calls the non-static `format` method from the `NumberFormat` object named `currency`. This returns a string that consists of a dollar sign plus the value of the `price` variable with two decimal places. In this format, negative numbers are enclosed in parentheses.

The second example shows how to format numbers with the percent format. The main difference between the first and second examples is that you use the `getPercentInstance` method to create a `NumberFormat` object that has the default percent format. Then, you can use the `format` method of this object to format a number as a percent. In this format, negative numbers have a leading minus sign.

The third example shows how to format numbers with the number format, and how to set the number of decimal places for a `NumberFormat` object. Here, the format is changed from the default of three decimal places to just one decimal place. In this format, negative numbers also have a leading minus sign.

The fourth example shows how you can use one statement to create a `NumberFormat` object and use its `format` method. Although this example accomplishes the same task as the second example, it doesn't create a variable for the `NumberFormat` object that you can use later in the program. As a result, you should only use code like this when you need to format just one number.

## The NumberFormat class

`java.text.NumberFormat`

### Three static methods of the NumberFormat class

Method	Returns a NumberFormat object that ...
<code>getCurrencyInstance()</code>	Has the default currency format (\$99,999.99).
<code>getPercentInstance()</code>	Has the default percent format (99%).
<code>getNumberInstance()</code>	Has the default number format (99,999.999).

### Three methods of a NumberFormat object

Method	Description
<code>format(anyNumberType)</code>	Returns a String object that has the format specified by the NumberFormat object.
<code>setMinimumFractionDigits(int)</code>	Sets the minimum number of decimal places.
<code>setMaximumFractionDigits(int)</code>	Sets the maximum number of decimal places.

#### Example 1: The currency format

```
double price = 11.575;
NumberFormat currency = NumberFormat.getCurrencyInstance();
String priceString = currency.format(price);           // returns $11.58
```

#### Example 2: The percent format

```
double majority = .505;
NumberFormat percent = NumberFormat.getPercentInstance();
String majorityString = percent.format(majority);     // returns 50%
```

#### Example 3: The number format with one decimal place

```
double miles = 15341.253;
NumberFormat number = NumberFormat.getNumberInstance();
number.setMaximumFractionDigits(1);
String milesString = number.format(miles);           // returns 15,341.3
```

#### Example 4: Two NumberFormat methods that are coded in one statement

```
String majorityString = NumberFormat.getPercentInstance().format(majority);
```

### Description

- You can use one of the three static methods to create a NumberFormat object. Then, you can use the methods of that object to format one or more numbers.
- When you use the format method, the result is automatically rounded by using a rounding technique called half-even. This means that the number is rounded up if the preceding digit is odd, but the extra decimal places are truncated if the preceding digit is even.
- Since the NumberFormat class is in the java.text package, you'll want to include an import statement when you use this class.

When you use the `format` method of a `NumberFormat` object, the numbers are automatically rounded by a technique called *half-even*, which rounds up if the preceding digit is odd, but rounds down if the preceding digit is even. If, for example, the currency format is used for a value of 123.455, the formatted result is \$123.46, which is what you would expect. But if the value is 123.445, the result is \$123.44. Although this is okay for many applications, it can cause problems in others. You'll learn more about this later in this chapter.

## How to use the Math class

---

The `Math` class provides a few dozen methods for working with numeric data types. Some of the most useful ones for business applications are presented in figure 3-8.

The first group of examples shows how to use the `round` method. Here, the first statement rounds a double type to a long type, and the second statement rounds a float type to an int type. Note, however, that this method only rounds to an integer value so it's not that useful.

The second group of examples shows how to use the `pow` method to raise the first argument to the power of the second argument. This method returns a double value and accepts two double arguments. However, since Java automatically converts any arguments of a less precise numeric type to a double, the `pow` method accepts all of the numeric types. In this example, the first statement is equal to  $2^2$ , the second statement is equal to  $2^3$ , and the third and fourth statements are equal to  $5^2$ .

In general, the methods of the `Math` class work the way you would expect. Sometimes, though, you may need to cast numeric types to get the methods to work the way you want them to. For example, the `pow` method returns a double type. So if you want to return an int type, you need to cast the double type to an int type as shown in the fourth `pow` example.

The third group of examples shows how to use the `sqrt` method to get the square root of a number, and the fourth group shows how to use the `max` and `min` methods to return the greater or lesser of two values. If you study these examples, you shouldn't have any trouble understanding how they work.

The fifth group of examples shows how to use the `random` method to generate random numbers. Since this method returns a random double value greater than or equal to 0.0 and less than 1.0, you can return any range of values by multiplying the random number by another number. In this example, the first statement returns a random double value greater than or equal to 0.0 and less than 100.0. Then, the second statement casts this double value to a long data type. A routine like this can be useful when you want to generate random values for testing a program.

If you have the right mathematical background, you shouldn't have any trouble using these or any of the other `Math` methods. If you've taken a course in trigonometry, for example, you should be able to understand the trigonometric methods that the `Math` class provides.

## The Math class

`java.lang.Math`

### Common static methods of the Math class

Method	Description
<code>round(float or double)</code>	Returns the closest long value to a double value or the closest int value to a float value. The result has no decimal places.
<code>pow(number, power)</code>	Returns a double value of a double argument (number) that is raised to the power of another double argument (power).
<code>sqrt(number)</code>	Returns a double value that's the square root of the double argument.
<code>max(a, b)</code>	Returns the greater of two float, double, int, or long arguments.
<code>min(a, b)</code>	Returns the lesser of two float, double, int, or long arguments.
<code>random()</code>	Returns a random double value greater than or equal to 0.0 and less than 1.0.

#### Example 1: The round method

```
long result = Math.round(1.667);    // result is 2
int result = Math.round(1.49F);     // result is 1
```

#### Example 2: The pow method

```
double result = Math.pow(2, 2);     // result is 4.0 (2*2)
double result = Math.pow(2, 3);     // result is 8.0 (2*2*2)
double result = Math.pow(5, 2);     // result is 25.0 (5 squared)
int result = (int) Math.pow(5, 2);  // result is 25 (5 squared)
```

#### Example 3: The sqrt method

```
double result = Math.sqrt(20.25);   // result is 4.5
```

#### Example 4: The max and min methods

```
int x = 67;
int y = 23;
int max = Math.max(x, y);           // max is 67
int min = Math.min(x, y);           // min is 23
```

#### Example 5: The random method

```
double x = Math.random() * 100;    // result is a value >= 0.0 and < 100.0
long result = (long) x;             // converts the result from double to long
```

### Description

- You can use the static methods of the Math class to perform common arithmetic operations. This figure summarizes the methods that are the most useful for business applications.
- When a method requires one or more arguments, you code them between the parentheses, separating multiple arguments with commas.
- In some cases, you need to cast the result to the data type that you want.

Figure 3-8 How to use the Math class

## How to use the Integer and Double classes

---

Figure 3-9 shows how to use a few of the constructors and static methods that are provided by the Integer and Double classes. Since these classes can be used to create objects that wrap around the primitive types, they are sometimes referred to as *wrapper classes*. Wrapper classes also exist for the other six primitive data types.

The first group of statements in this figure shows how to create Integer and Double objects that can store int and double data types. This is useful when you want to provide an int or double data type as an argument to a method, but the method requires that the argument be an object, not a data type. You'll see how this works in a later chapter. Once you create an Integer or Double object, you can use any of the methods of these classes to work with the data it contains.

Note, however, that these classes also provide static methods that you can use without creating objects. For instance, the second group of statements in this figure shows how to use the static toString method to convert a primitive type to a string. Here, the first statement converts the int variable named counter to a string and returns the value to a string variable named counterString. The second statement converts the double variable named price to a string and returns that value to the string variable named priceString.

Similarly, the third group of statements shows how to use the static parse methods to convert strings to primitive types. Here, the first statement uses the parseInt method of the Integer class to convert a string to an int data type. The second statement uses the parseDouble method of the Double class to convert a string to a double data type. Once these statements have been executed, you can use the quantity and price variables in arithmetic expressions.

But what happens if the string contains a non-numeric value like "ten" that can't be parsed to an int or double type? In that case, the parseInt or parseDouble method will cause a runtime error known as an exception. Using Java terminology, you can say that the method will *throw an exception*. In chapter 5, you'll learn how to *catch* the exceptions that are thrown by these methods.



## Constructors for the Integer and Double classes

Constructor	Description
<code>Integer(int)</code>	Constructs an Integer object from an int data type.
<code>Double(double)</code>	Constructs a Double object from a double data type.

## Two static methods of the Integer class

Method	Description
<code>parseInt(stringName)</code>	Attempts to convert the String object that's supplied as an argument to an int type. If successful, it returns the int value. If unsuccessful, it throws an exception.
<code>toString(intName)</code>	Converts the int value that's supplied as an argument to a String object and returns that String object.

## Two static methods of the Double class

Method	Description
<code>parseDouble(stringName)</code>	Attempts to convert the String object that's supplied as an argument to a double type. If successful, it returns the double value. If unsuccessful, it throws an exception.
<code>toString(doubleName)</code>	Converts the double value that's supplied as an argument to a String object and returns that String object

## How to create Integer and Double objects

```
Integer quantityIntegerObject = new Integer(quantity);
Double priceDoubleObject = new Double(price);
```

## How to use static methods to convert primitive types to String objects

```
String counterString = Integer.toString(counter);
String priceString = Double.toString(price);
```

## How to use static methods to convert String objects to primitive types

```
int quantity = Integer.parseInt(quantityString);
double price = Double.parseDouble(priceString);
```

## Description

- The Integer and Double classes are known as *wrapper classes* since they can be used to construct Integer and Double objects that contain (wrap around) int and double values. This can be useful when you need to pass an int or double value to a method that only accepts objects, not primitive data types.
- These classes also provide static methods that you can use for converting values from these data types to strings and vice versa. And every primitive type has a wrapper class that works like the Integer and Double classes.
- If the `parseInt` and `parseDouble` methods can't successfully parse the string, they will cause an error to occur. In Java terminology, this is known as *throwing an exception*. You'll learn how to handle or *catch* exceptions in chapter 5.

Figure 3-9 How to use the Integer and Double classes

## The formatted Invoice application

---

To illustrate some of the skills you've just learned, figure 3-10 shows the console and code for an enhanced version of the Invoice application that was presented in chapter 2. This time, the application does a few more calculations and formats the results before displaying them. You can see the results for one user entry in the console that's in this figure.

### The code for the application

---

In chapter 2, you saw how the console looks when displayed by Windows. In this figure, the console is displayed in a platform-neutral format that's easy to read. This is the format that will be used to display console output for the rest of this book.

The shaded code in this figure identifies the primary changes to the Invoice application of the last chapter. First, two new values are calculated. Sales tax is calculated by multiplying the total before tax by .05. And the invoice total is calculated by adding the sales tax to the total before tax.

Second, currency and percent objects are created by using the methods of the `NumberFormat` class. Then, the format methods of these objects are used to format the five values that have been calculated by this application. This shows how one currency object can be used to format two or more values. The result of each use of the format method is a string that is added to the message that eventually gets displayed.

Although this application is now taking on a more professional look, you should remember that it still has some shortcomings. First, it doesn't handle the exception that's thrown if the user doesn't enter a valid number at the console. You'll learn how to fix that problem in chapter 5. Second, because of the way rounding works with the `NumberFormat` methods, the results may not always come out the way you want them to. You'll learn more about that next.

## The console for the formatted Invoice application

```
Enter subtotal: 150.50
Discount percent: 10%
Discount amount: $15.05
Total before tax: $135.45
Sales tax: $6.77
Invoice total: $142.22

Continue? (y/n):
```

## The code for the formatted Invoice application

```
import java.util.Scanner;
import java.text.NumberFormat;

public class InvoiceApp
{
    public static void main(String[] args)
    {
        // create a Scanner object and start while loop
        Scanner sc = new Scanner(System.in);
        String choice = "y";
        while (choice.equalsIgnoreCase("y"))
        {
            // get the input from the user
            System.out.print("Enter subtotal:  ");
            double subtotal = sc.nextDouble();

            // calculate the results
            double discountPercent = 0.0;
            if (subtotal >= 100)
                discountPercent = .1;
            else
                discountPercent = 0.0;
            double discountAmount = subtotal * discountPercent;
            double totalBeforeTax = subtotal - discountAmount;
            double salesTax = totalBeforeTax * .05;
            double total = totalBeforeTax + salesTax;

            // format and display the results
            NumberFormat currency = NumberFormat.getCurrencyInstance();
            NumberFormat percent = NumberFormat.getPercentInstance();
            String message =
                "Discount percent: " + percent.format(discountPercent) + "\n"
                + "Discount amount: " + currency.format(discountAmount) + "\n"
                + "Total before tax: " + currency.format(totalBeforeTax) + "\n"
                + "Sales tax: " + currency.format(salesTax) + "\n"
                + "Invoice total: " + currency.format(total) + "\n";
            System.out.println(message);

            // see if the user wants to continue
            System.out.print("Continue? (y/n): ");
            choice = sc.next();
            System.out.println();
        }
    }
}
```

Figure 3-10 The formatted Invoice application

## How to analyze the data problems in the Invoice application

---

The console at the top of figure 3-11 shows more output from the Invoice application in figure 3-10. But wait! The results for a subtotal entry of 100.05 don't add up. If the discount amount is \$10.00, the total before tax should be \$90.05, but it's \$90.04. Similarly, the sales tax for a subtotal entry of .70 is shown as \$0.03, so the invoice total should be \$0.73, but it's shown as is \$0.74. What's going on?

To analyze data problems like this, you can add *debugging statements* to a program like the ones in this figure. These statements display the unformatted values of the result fields so you can see what they are before they're formatted. This is illustrated by the console at the bottom of this figure, which shows the results for the same entries as the ones in the console at the top of this figure.

If you look at the unformatted results for the first entry (100.05), you can easily see what's going on. Because of the way `NumberFormat` rounding works, the discount amount value of 10.005 and the total before tax value of 90.045 aren't rounded up. However, the invoice total value of 94.54725 is rounded up. With this extra information, you know that everything is working the way it's supposed to, even though you're not displaying the results you want.

Now, if you look at the unformatted results for the second entry (.70), you can see another type of data problem. In this case, the sales tax is shown as .034999999999999996 when it should be .035. This happens because floating-point numbers, which are binary, aren't able to exactly represent some decimal fractions. As a result, the formatted value is \$0.03 when it should be rounded up to \$0.04. However, the unformatted invoice total is correctly represented as 0.735, which is rounded to a formatted \$0.74. And here again, it looks like Java can't add.

Although trivial errors like these are acceptable in many applications, they are unacceptable in some business applications. And for those applications, you need to provide solutions that deliver the results that you want. (Imagine getting an invoice that didn't add up!)

One solution is to write your own code that does the rounding so you don't need to use the `NumberFormat` class to do the rounding for you. As you go through this book, you'll learn how to use classes and methods that will help you do that. However, that still doesn't deal with the fact that some decimal fractions can't be accurately represented by floating-point numbers. To solve that problem as well as the other data problems, the best solution is to use the `BigDecimal` class that you'll learn about next.

## Output data that illustrates a problem with the Invoice application

```

Enter subtotal: 100.05
Discount percent: 10%
Discount amount: $10.00
Total before tax: $90.04
Sales tax: $4.50
Invoice total: $94.55

Continue? (y/n): y

Enter subtotal: .70
Discount percent: 0%
Discount amount: $0.00
Total before tax: $0.70
Sales tax: $0.03
Invoice total: $0.74

Continue? (y/n):

```

## Statements that you can add to the program to help analyze this problem

```

// debugging statements that display the unformatted fields
// these are added before displaying the formatted results
String debugMessage = "\nUNFORMATTED RESULTS\n"
    + "Discount percent: " + discountPercent + "\n"
    + "Discount amount: " + discountAmount + "\n"
    + "Total before tax: " + totalBeforeTax + "\n"
    + "Sales tax: " + salesTax + "\n"
    + "Invoice total: " + total + "\n"
    + "\nFORMATTED RESULTS";

System.out.println(debugMessage);

```

## The unformatted and formatted output data

```

Enter subtotal: 100.05

UNFORMATTED RESULTS
Discount percent: 0.1
Discount amount: 10.005
Total before tax: 90.045
Sales tax: 4.50225
Invoice total: 94.54725

FORMATTED RESULTS
Discount percent: 10%
Discount amount: $10.00
Total before tax: $90.04
Sales tax: $4.50
Invoice total: $94.55

Continue? (y/n): y

Enter subtotal: .70

UNFORMATTED RESULTS
Discount percent: 0.0
Discount amount: 0.0
Total before tax: 0.7
Sales tax: 0.034999999999999996
Invoice total: 0.735

FORMATTED RESULTS
Discount percent: 0%
Discount amount: $0.00
Total before tax: $0.70
Sales tax: $0.03
Invoice total: $0.74

Continue? (y/n):

```

Figure 3-11 How to analyze the data problems in the Invoice application

## How to use the `BigDecimal` class for working with decimal data

---

The `BigDecimal` class is designed to solve two types of problems that are associated with floating-point numbers. First, the `BigDecimal` class can be used to exactly represent decimal numbers. Second, it can be used to work with numbers that have more than 16 significant digits.

### The constructors and methods of the `BigDecimal` class

---

Figure 3-12 summarizes a few of the constructors that you can use with the `BigDecimal` class. These constructors accept an `int`, `double`, `long`, or `string` argument and create a `BigDecimal` object from it. Because floating-point numbers are limited to 16 significant digits and because these numbers don't always represent decimal numbers exactly, it's often best to construct `BigDecimal` objects from strings rather than doubles.

Once you create a `BigDecimal` object, you can use its methods to work with the data. In this figure, for example, you can see some of the `BigDecimal` methods that are most useful in business applications. Here, the `add`, `subtract`, `multiply`, and `divide` methods let you perform those operations. The `compareTo` method lets you compare the values in two `BigDecimal` objects. And the `toString` method converts the value of a `BigDecimal` object to a string.

This figure also includes the `setScale` method, which lets you set the number of decimal places (*scale*) for the value in a `BigDecimal` object as well as the rounding mode. For example, you can use the `setScale` method to return a number that's rounded to two decimal places like this:

```
salesTax = salesTax.setScale(2, RoundingMode.HALF_UP);
```

In this example, `RoundingMode.HALF_UP` is a value in the `RoundingMode` enumeration that's summarized in this figure. The `scale` and `rounding mode` arguments work the same for the `divide` method.

Enumerations are similar to classes, and you'll learn more about them in chapter 9. For now, you can code the rounding mode as `HALF_UP` because it provides the type of rounding that is normal for business applications. However, you need to import the `RoundingMode` enumeration at the start of the application unless you want to qualify the rounding mode like this:

```
java.math.RoundingMode.HALF_UP
```

If you look at the API documentation for the `BigDecimal` class, you'll see that it provides several other methods that you may want to use. This class also provides many other features that you may want to become more familiar with.

## The BigDecimal class

`java.math.BigDecimal`

### Constructors of the BigDecimal class

Constructor	Description
<b>BigDecimal</b> (int)	Creates a new BigDecimal object with the specified int value.
<b>BigDecimal</b> (double)	Creates a new BigDecimal object with the specified double value.
<b>BigDecimal</b> (long)	Creates a new BigDecimal object with the specified long value.
<b>BigDecimal</b> (String)	Creates a new BigDecimal object with the specified String object. Because of the limitations of floating-point numbers, it's often best to create BigDecimal objects from strings.

### Methods of the BigDecimal class

Methods	Description
<b>add</b> (value)	Returns the value of this BigDecimal object after the specified BigDecimal value has been added to it.
<b>compareTo</b> (value)	Compares the value of the BigDecimal object with the value of the specified BigDecimal object and returns -1 if less, 0 if equal, and 1 if greater.
<b>divide</b> (value, scale, rounding-mode)	Returns the value of this BigDecimal object divided by the value of the specified BigDecimal object, sets the specified scale, and uses the specified rounding mode.
<b>multiply</b> (value)	Returns the value of this BigDecimal object multiplied by the specified BigDecimal value.
<b>setScale</b> (scale, rounding-mode)	Sets the scale and rounding mode for the BigDecimal object.
<b>subtract</b> (value)	Returns the value of this BigDecimal object after the specified BigDecimal value has been subtracted from it.
<b>toString</b> ()	Converts the BigDecimal value to a string.

## The RoundingMode enumeration

`java.math.RoundingMode`

### Two of the values in the RoundingMode enumeration

Values	Description
<b>HALF_UP</b>	Round towards the “nearest neighbor” unless both neighbors are equidistant, in which case round up.
<b>HALF_EVEN</b>	Round towards the “nearest neighbor” unless both neighbors are equidistant, in which case round toward the even neighbor.

### Description

- The BigDecimal class provides a way to perform accurate decimal calculations in Java. It also provides a way to store numbers with more than 16 significant digits.
- You can pass a BigDecimal object to the format method of a NumberFormat object, but NumberFormat objects limit the results to 16 significant digits.

Figure 3-12 The constructors and methods for the BigDecimal class

## How to use **BigDecimal** arithmetic in the Invoice application

---

Figure 3-13 shows how you can use **BigDecimal** arithmetic in the Invoice application. To start, look at the console output when **BigDecimal** is used. As you can see, this solves both the rounding problem and the floating-point problem so it now works the way you want it to.

To use **BigDecimal** arithmetic in the Invoice application, you start by coding an import statement that imports all of the classes and enumerations of the `java.math` package. This includes both the **BigDecimal** class and the **RoundingMode** enumeration. Then, you use the constructors and methods of the **BigDecimal** class to create the **BigDecimal** objects, do the calculations, and round the results when necessary.

In this figure, the code starts by constructing **BigDecimal** objects from the `subtotal` and `discountPercent` variables, which are `double` types. To avoid conversion problems, though, the `toString` method of the `Double` class is used to convert the `subtotal` and `discountPercent` values to strings that are used in the **BigDecimal** constructors.

Since the user may enter `subtotal` values that contain more than two decimal places, the `setScale` method is used to round the `subtotal` entry after it has been converted to a **BigDecimal** object. However, since the `discountPercent` variable only contains two decimal places, it isn't rounded. From this point on, all of the numbers are stored as **BigDecimal** objects and all of the calculations are done with **BigDecimal** methods.

In the statements that follow, only discount amount and sales tax need to be rounded. That's because they're calculated using multiplication, which can result in extra decimal places. In contrast, the other numbers (`total before tax` and `total`) don't need to be rounded because they're calculated using subtraction and addition. Once the calculations and rounding are done, you can safely use the `NumberFormat` objects and methods to format the **BigDecimal** objects for display.

When working with **BigDecimal** objects, you may sometimes need to create one **BigDecimal** object from another **BigDecimal** object. However, you can't supply a **BigDecimal** object to the constructor of the **BigDecimal** class. Instead, you need to call the `toString` method from the **BigDecimal** object to convert the **BigDecimal** object to a `String` object. Then, you can pass that `String` object as the argument of the constructor as illustrated by the last statement in this figure.

Is this a lot of work just to do simple business arithmetic? Relative to some other languages, you would have to say that it is. In fact, it's fair to say that this is a weakness of Java. In contrast, languages like Microsoft's `C#` and Visual Basic .NET provide a decimal data type that can have up to 28 significant digits along with a `Round` method in the `Math` class that's easy to use with decimal data. However, once you get the hang of working with the **BigDecimal** class, you should be able to solve floating-point and rounding problems with ease.



## The Invoice application output when BigDecimal arithmetic is used

```

Enter subtotal: 100.05
Subtotal: $100.05
Discount percent: 10%
Discount amount: $10.01
Total before tax: $90.04
Sales tax: $4.50
Invoice total: $94.54

Continue? (y/n): y

Enter subtotal: .70
Subtotal: $0.70
Discount percent: 0%
Discount amount: $0.00
Total before tax: $0.70
Sales tax: $0.04
Invoice total: $0.74

Continue? (y/n):

```

## The import statement that's required for BigDecimal arithmetic

```
import java.math.*; // imports all classes and enumerations in java.math
```

## The code for using BigDecimal arithmetic in the Invoice application

```

// convert subtotal and discount percent to BigDecimal
BigDecimal decimalSubtotal = new BigDecimal(Double.toString(subtotal));
decimalSubtotal = decimalSubtotal.setScale(2, RoundingMode.HALF_UP);
BigDecimal decimalDiscountPercent =
    new BigDecimal(Double.toString(discountPercent));

// calculate discount amount
BigDecimal discountAmount =
    decimalSubtotal.multiply(decimalDiscountPercent);
discountAmount = discountAmount.setScale(2, RoundingMode.HALF_UP);

// calculate total before tax, sales tax, and total
BigDecimal totalBeforeTax = decimalSubtotal.subtract(discountAmount);
BigDecimal salesTaxPercent = new BigDecimal(".05");
BigDecimal salesTax = salesTaxPercent.multiply(totalBeforeTax);
salesTax = salesTax.setScale(2, RoundingMode.HALF_UP);
BigDecimal total = totalBeforeTax.add(salesTax);

```

## How to create a BigDecimal object from another BigDecimal object

```
BigDecimal total2 = new BigDecimal(total.toString());
```

## Description

- With this code, all of the result values are stored in BigDecimal objects, and all of the results have two decimal places that have been rounded correctly when needed.
- Once the results have been calculated, you can use the NumberFormat methods to format the values in the BigDecimal objects without any fear of rounding problems. However, the methods of the NumberFormat object limits the results to 16 significant digits.

## Perspective

---

If this chapter has succeeded, you should now be able to work with whatever primitive data types you need in your applications. You should be able to use the `NumberFormat`, `Math`, `Double`, and `Integer` classes whenever you need them. And you should be able to use the `BigDecimal` class to solve the problems that are associated with floating-point numbers.

## Summary

---

- Java provides eight *primitive data types* to store *integer*, *floating-point*, *character*, and *boolean* values.
- *Variables* store data that changes as a program runs. *Constants* store data that doesn't change as a program runs. You use *assignment statements* to assign values to variables.
- You can use *arithmetic operators* to form *arithmetic expressions*, and you can use some *assignment operators* as a shorthand for some types of arithmetic expressions.
- Java can *implicitly cast* a less precise data type to a more precise data type. Java also lets you *explicitly cast* a more precise data type to a less precise data type.
- You can use the `NumberFormat` class to apply standard currency, percent, and number formats to any of the primitive numeric types.
- You can use the static methods of the `Math` class to perform mathematical operations such as rounding numbers and calculating square roots.
- You can use the constructors of the `Double` and `Integer` *wrapper classes* to create objects that wrap double and int values. You can also use the static methods of these classes to convert strings to numbers and vice versa.
- You can use the constructors of the `BigDecimal` class to create objects that store decimal values that aren't limited to 16 significant digits. Then, you can use the methods of these objects to do the calculations that your programs require.

### Exercise 3-1 Test the Invoice application

In this exercise, you'll compile and test the formatted Invoice application that's presented in figure 3-10.

1. Open the file named `FormattedInvoiceApp.java` that you should find in the `ch03` directory for your exercises. Then, compile and run the application. As you test the application, enter the three subtotal values that are shown in figures 3-10 and 3-11 to see how the program works and to see what the problems are.
2. To better understand what is happening, add debugging statements like those in figure 3-11 so the program displays two sets of data for each entry: first the unformatted output, then the formatted output. When you add debugging statements, you should try to do it in a way that makes them easy to remove when you're through debugging.
3. Test the application again with a range of entries so you clearly see what the data problems are when you study the unformatted and formatted results.

### Exercise 3-2 Modify the Test Score application

In this exercise, you'll use some of the skills that you learned in this chapter as you modify the Test Score application that you worked with in the last chapter, but you won't use `BigDecimal` arithmetic.

1. Open the file named `ModifiedTestScoreApp.java` that you should find in the `ch02` directory if you did exercise 2-2. If you didn't do that exercise, open `TestScoreApp` instead.
2. Save the file as `EnhancedTestScoreApp.java` in the `ch03` directory, and change the class name in the file to `EnhancedTestScoreApp`. Then, compile and run the program to refresh your memory about how it works.
3. Use the `+=` operator to increase the `scoreCount` and `scoreTotal` fields. Then, test this to make sure that it works.
4. As the user enters test scores, use the methods of the `Math` class to keep track of the minimum and maximum scores. When the user enters 999 to end the program, display these scores at the end of the other output data. Now, test these changes to make sure that they work. (This step can be challenging if you're new to programming, but you'll learn a lot by doing it.)
5. Change the variable that you use to total the scores from a `double` to an `int` data type. Then, use casting to cast the score count and score total to `doubles` as you calculate the average score and save that average as a `double`. Now, test that change.
6. Use the `NumberFormat` class to round the average score to one decimal place before displaying it at the end of the program. Then, test this change. Note that the rounding method that's used doesn't matter in a program like this.

### Exercise 3-3    Create a new application

In this exercise, you'll develop an application that will give you a chance to use your new skills. This application asks the user to enter a file size in megabytes (MB) and then calculates how long it takes to download that file with a 56K analog modem (you won't need to use `BigDecimal` arithmetic). The output from this application should look something like this:

```
Welcome to the Download Time Estimator

This program calculates how long it will take to
download a file with a 56K analog modem.

Enter file size (MB): 50

A "56K" modem will take 2 hours 44 minutes 6 seconds

Continue? (y/n):
```

1. Instead of starting this application from scratch, open the file named `FormattedInvoiceApp.java` in the `ch03` directory. Then, save it with the name `DownloadTimeApp.java`, and change its class name to `DownloadTimeApp`.
2. Delete the code that you won't need for this application, and modify the code that remains so it provides for the basic operation of the program without the calculations. These first two steps are an efficient way to start any new application because you don't have to re-enter the routine code.
3. Add the code that calculates the hours, minutes, and seconds needed to download this file with a 56K analog modem. To do the calculations, assume that a 56K modem can transfer data at the rate of 5.2 kilobytes (KB) per second. Then, add the code for displaying the results. (You also need to know that 1 MB is equal to 1,024 KB).
4. Compile and run the application. Enter a value of 50 for the file size to be sure that the calculated value is the same as shown above. Then, enter other values to see how they work.

### Exercise 3-4    Use `BigDecimal` arithmetic

To get some practice with `BigDecimal` arithmetic, this exercise has you modify the Test Score application so it uses `BigDecimal` arithmetic.

1. Open `EnhancedTestScoreApp` in the `ch03` directory or your last version of the Test Score application in the `ch02` directory. Then, save the file as `BDTestScoreApp` in `ch03`, and change the class name to `BDTestScoreApp`.
2. Modify the program so it uses `BigDecimal` arithmetic to calculate the average test score with the result rounded to one decimal place. Be sure to use the appropriate `toString` methods of either the `Double` or `Integer` classes so the `BigDecimal` objects are constructed from string values. Then, test this change with a range of values.

# 4

## How to code control statements

In chapter 2, you learned how to code simple if and while statements to control the execution of your applications. Now, you'll learn more about coding these statements. You'll learn how to code the other control statements that Java offers. And you'll learn how to code your own static methods, which will help you divide your applications into manageable parts.

<b>How to code Boolean expressions .....</b>	<b>122</b>
How to compare primitive data types .....	122
How to compare strings .....	124
How to use the logical operators .....	126
<b>How to code if/else and switch statements .....</b>	<b>128</b>
How to code if/else statements .....	128
How to code switch statements .....	130
An enhanced version of the Invoice application .....	132
<b>How to code loops .....</b>	<b>134</b>
How to code while and do-while loops .....	134
How to code for loops .....	136
The Future Value application .....	138
How to code nested for loops .....	140
<b>How to code break and continue statements .....</b>	<b>142</b>
How to code break statements .....	142
How to code continue statements .....	144
<b>How to code and call static methods .....</b>	<b>146</b>
How to code static methods .....	146
How to call static methods .....	146
The Future Value application with a static method .....	148
<b>Perspective .....</b>	<b>150</b>

## How to code Boolean expressions

---

In chapter 2, you learned how to code the *Boolean expressions* that control the operation of your control statements. These are expressions that evaluate to either true or false. To start, this topic repeats some of the information that you learned before, but in a larger context.

### How to compare primitive data types

---

Figure 4-1 shows how to use the six *relational operators* to code a Boolean expression that compares *operands* that are primitive data types. In a Boolean expression, an operand can be a literal, a variable, an arithmetic expression, or a keyword such as true or false.

The first three expressions in this figure use the equality operator (==) to test if the two operands are equal. To use this operator, you must code two equals signs instead of one. That's because a single equals sign is used for assignment statements. As a result, if you try to code a Boolean expression with a single equals sign, your code won't compile.

The next expression uses the inequality operator (!=) to test if a variable is not equal to a numeric literal. The two expressions after that use the greater than operator (>) to test if a variable is greater than a numeric literal and the less than operator (<) to test if one variable is less than another. And the two expressions after that use the greater than or equal operator (>=) and less than or equal operator (<=) to compare operands.

The last two expressions in this figure illustrate that you don't need the == or != operator when you use a boolean variable in an expression. That's because, by definition, a boolean variable evaluates to a boolean value. As a result,

```
isValid == true
```

is the same as

```
isValid
```

and

```
!isValid
```

is the same as

```
isValid == false
```

Although the first and last expressions may be easier for a beginning programmer to understand, the second and third expressions are commonly used by professional programmers.

When comparing numeric values, you usually compare values of the same data type. However, if you compare different types of numeric values, Java will automatically cast the less precise numeric type to the more precise type. For example, if you compare an int type to a double type, the int type will be cast to the double type before the comparison is made.

## Relational operators

Operator	Name	Description
<code>==</code>	Equality	Returns a true value if both operands are equal.
<code>!=</code>	Inequality	Returns a true value if the left and right operands are not equal.
<code>&gt;</code>	Greater Than	Returns a true value if the left operand is greater than the right operand.
<code>&lt;</code>	Less Than	Returns a true value if the left operand is less than the right operand.
<code>&gt;=</code>	Greater Than Or Equal	Returns a true value if the left operand is greater than or equal to the right operand.
<code>&lt;=</code>	Less Than Or Equal	Returns a true value if the left operand is less than or equal to the right operand.

## Examples of Boolean expressions

```

discountPercent == 2.3    // equal to a numeric literal
letter == 'y'             // equal to a char literal
isValid == false          // equal to the false value

subtotal != 0             // not equal to a numeric literal

years > 0                 // greater than a numeric literal
i < months                // less than a variable

subtotal >= 500           // greater than or equal to a numeric literal
quantity <= reorderPoint // less than or equal to a variable

isValid                   // isValid is equal to true
!isValid                  // isValid is equal to false

```

## Description

- You can use the relational operators to create a Boolean expression that compares two operands and returns a boolean value that is either true or false.
- If you compare two numeric operands that are not of the same type, Java will convert the less precise operand to the type of the more precise operand before doing the comparison.
- By definition, a boolean variable evaluates to a boolean value of true or false.

Figure 4-1 How to compare primitive data types

## How to compare strings

---

As you learned in chapter 2, a string is an object, not a primitive data type, so you can't use the relational operators to compare strings. Instead, you must use the `equals` or `equalsIgnoreCase` method of the `String` class as shown by the expressions at the start of figure 4-2.

Both of these methods require an argument that provides the `String` object or literal that you want to compare with the current object. The difference between the two is that the `equals` method is case-sensitive while the `equalsIgnoreCase` method is not.

If you call the `equals` or `equalsIgnoreCase` method from a string that contains a null, however, Java will throw an exception. To avoid that, you can use the equality operator (`==`) or the inequality operator (`!=`) to check whether a string contains a null before you use the `equals` or `equalsIgnoreCase` method. This is illustrated by the last two expressions at the start of this figure.

The next block of code shows what happens when you test two strings for equality with the `==` operator. Here, the code asks you to enter values for two different strings. No matter what values you enter, though, the `equals` comparison that follows will be false. If, for example, you enter "abc" for both strings, the `equals` test will be false.

That's because all object variables are *reference types*, which means that they don't actually contain the data like primitive types do. Instead, reference types refer to (or point to) the data, which is held in another area of internal storage. For these types, the equality and inequality operators test to see whether the variables refer to the same object. If they do, they're considered equal. But if they refer to two different objects, they're considered unequal, even if the objects contain the same values.

What happens if you issue this statement?

```
string1 = string2;
```

The variable named `string1` now refers to the same data that `string2` refers to. As a result, the Boolean expression

```
(string1 == string2)
```

will be true because both variables will refer to the same object.

This just makes the point that you shouldn't use the equality and inequality operators to test whether two strings have the same values because these operators don't work that way. Since all objects are reference types, this holds true for other types of objects too. As a result, you'll learn other ways to test objects for equality as you progress through this book.



## Two methods of the String class

Method	Description
<code>equals (String)</code>	Compares the current String object with the String object specified as the argument and returns a true value if they are equal. This method makes a case-sensitive comparison.
<code>equalsIgnoreCase (String)</code>	Works like the equals method but is not case-sensitive.

## Expressions that compare two string values

```

firstName.equals("Frank")           // equal to a string literal
firstName.equalsIgnoreCase("Frank") // equal to a string literal
firstName.equals("")                // equal to an empty string

!lastName.equals("Jones")           // not equal to a string literal
!code.equalsIgnoreCase(productCode) // not equal to another string variable

firstName == null                    // equal to a null value
firstName != null                    // not equal to a null value

```

## Code that tests whether two strings refer to the same object

```

Scanner sc = new Scanner(System.in);
System.out.print("Enter string1: ");
String string1 = sc.next();
System.out.print("Enter string2: ");
String string2 = sc.next();

if (string1 == string2)           // this will be false no matter what you enter
    System.out.println("string1 = string2");
else
    System.out.println("string1 not = string2");

```

## Description

- To test two strings to see whether they contain the same string values, you must call one of the methods of the String object.
- To test whether a string is null, you can use the equality operator (==) or the inequality operator (!=) with the null keyword.
- A string object is a *reference type*, not a primitive data type. That means that a string variable doesn't contain the data like a primitive type does. Instead, a string variable refers to (or points to) the data, which is in another location of computer memory.
- If you use the equality or inequality operator to compare two string variables, Java tests to see whether the two strings refer to the same String object. If they do, the expression is true. If they don't, it's false.

## Technical note

- Because Java stores string literals in pools to reduce duplication, the equality and inequality tests for strings may not work as shown above when two String objects are assigned the same literal value.

## How to use the logical operators

---

Figure 4-3 shows how to use the *logical operators* to code a Boolean expression that consists of two or more Boolean expressions. For example, the first expression uses the `&&` operator. As a result, it evaluates to true if both the first expression *and* the second expression evaluate to true. Conversely, the second expression uses the `||` operator. As a result, it evaluates to a true value if either the first expression *or* the second expression evaluate to true.

When you use the `&&` and `||` operators, the second expression is only evaluated if necessary. Because of that, these operators are sometimes referred to as the *short-circuit operators*. To illustrate, suppose the value of `subtotal` in the first example is less than 250. Then, the first expression evaluates to false. That means that the entire expression will return a false value. As a result, the second expression is not evaluated. Since this is more efficient than always evaluating both expressions, you'll want to use these operators most of the time.

However, there may be times when you want to evaluate both expressions regardless of the value that's returned by the first expression. For example, there may be times when the second expression performs an operation such as incrementing a variable or calling a method. In that case, you can use the `&` and `|` operators to make sure that the second expression is evaluated.

You can also use multiple logical operators in the same expression as illustrated by the fifth example. Here, the `&&` and `||` operators connect three expressions. As a result, the entire expression is true if the first *and* second expressions are true *or* the third expression is true.

When you code this type of expression, the expression is evaluated from left to right based on this order of precedence: arithmetic operations first, followed by relational operations, followed by logical operations. For logical operations, And operations are performed before Or operations. If you need to change this sequence or if there's any doubt about the order of precedence, you can use parentheses to clarify or control this evaluation sequence.

If necessary, you can use the `!` operator to reverse the value of an expression. However, this can create code that's difficult to read. As a result, you should avoid using the `!` operators whenever possible. For example, instead of coding

```
!(subtotal < 100)
```

you can code

```
subtotal >= 100
```

Both expressions perform the same task, but the second expression is easier to read.

## Logical operators

Operator	Name	Description
&&	And	Returns a true value if both expressions are true. This operator only evaluates the second expression if necessary.
	Or	Returns a true value if either expression is true. This operator only evaluates the second expression if necessary.
&	And	Returns a true value if both expressions are true. This operator always evaluates both expressions.
	Or	Returns a true value if either expression is true. This operator always evaluates both expressions.
!	Not	Reverses the value of the expression.

## Examples

```

subtotal >= 250 && subtotal < 500
timeInService <=4 || timeInService >= 12

isValid == true & counter++ < years
isValid == true | counter++ < years

(subtotal >= 250 && subtotal < 500) || isValid == true

!(counter++ >= years)

```

## Description

- You can use the *logical operators* to create a Boolean expression that combines two or more Boolean expressions.
- Since the && and || operators only evaluate the second expression if necessary, they're sometimes referred to as *short-circuit operators* and are slightly more efficient than the & and | operators.
- By default, Not operations are performed first, followed by And operations, and then Or operations. These operations are performed after arithmetic operations and relational operations.
- You can use parentheses to change the sequence in which the operations will be performed or to clarify the sequence of operations.

Figure 4-3 How to use the logical operators

## How to code if/else and switch statements

---

In chapter 2, you were introduced to the if/else statement, but this topic will expand on that. This topic will also present the switch statement.

### How to code if/else statements

---

Figure 4-4 reviews the use of the *if/else statement* (or just *if statement*). This is Java's implementation of the *selection structure*.

When an if statement is executed, Java begins by evaluating the Boolean expression in the if clause. If it's true, the statements within this clause are executed and the rest of the if/else statement is skipped. If it's false, Java evaluates the first else if clause (if there is one). Then, if its Boolean expression is true, the statements within this else if clause are executed, and the rest of the if/else statement is skipped. Otherwise, Java evaluates the next else if clause.

This continues with any remaining else if clauses. Finally, if none of the clauses contains a Boolean expression that evaluates to true, Java executes the statements in the else clause (if there is one). If none of the Boolean expressions are true and there is no else clause, Java doesn't execute any statements.

As the syntax shows, you code the statements for an if or else clause in braces. The braces are optional if a clause contains only one statement, as illustrated by the first example in this figure. They're required if the clause contains two or more statements, as illustrated by the second example.

Whenever you code a set of braces in Java, you are explicitly defining a *block* of code that may contain one or more statements. Then, any variables that are declared within those braces have *block scope*. In other words, they can't be accessed outside of that block. As a result, if you want to access the variable outside of the block, you must declare it before the block. This is illustrated by the second example.

However, in if/else statements, each clause automatically has block scope even if you don't code the braces. As a result, if you want to access the variable outside of the if/else statement, you must declare it before the if/else statement, as shown in the first example. In addition, if you try to declare a variable in an if/else clause without using braces, you'll get a compile-time error. That makes sense because the variable can't be used outside of the clause, and there aren't any additional statements within the clause to use it.

When coding if statements, it's a common practice to code one if statement within another if statement. This is known as *nesting* if statements, and it's illustrated by the third example in this figure. When you nest if statements, it's a good practice to indent the nested statements and their clauses since this allows the programmer to easily identify where each nested statement begins and ends.

Another good coding practice is to code the conditions with a logical structure and in a logical sequence. If necessary, you can also add comments to your code so it's easier to follow. As always, the easier your code is to read and understand, the easier it is to test, debug, and maintain.

## The syntax of the if/else statement

```
if (booleanExpression) {statements}
[else if (booleanExpression) {statements}] ...
[else {statements}]
```

### Example 1: An if statement with else if and else clauses

```
double discountPercent = 0.0;
if (subtotal >= 100 && subtotal < 200)
    discountPercent = .1;
else if (subtotal >= 200 && subtotal < 300)
    discountPercent = .2;
else if (subtotal >= 300)
    discountPercent = .3;
else
    discountPercent = 0.05;
```

### Example 2: An if statement that contains two blocks of code

```
double discountPercent = 0.0;
if (customerType.equals("R"))
{
    discountPercent = .1;
    shippingMethod = "UPS";
}
else if (customerType.equals("C"))
{
    discountPercent = .2;
    shippingMethod = "Bulk";
}
else
    shippingMethod = "USPS";
```

// start block  
// end block  
// start block  
// end block

### Example 3: Nested if statements

```
if (customerType.equals("R"))
{
    if (subtotal >= 100)
        discountPercent = .2;
    else
        discountPercent = .1;
}
else
    discountPercent = .4;
```

// begin nested if  
// end nested if

## Description

- If a clause in an if/else statement contains just one statement, you don't have to enclose the statement in braces. You can just end the clause with a semicolon. However, this statement can't declare a variable or it won't compile.
- If a clause requires more than one statement, you must enclose the *block* of statements in braces. Then, any variable that is declared within the block has *block scope* so it can only be used within that block.

## How to code switch statements

---

Figure 4-5 shows how to work with the *switch statement*. This is the Java implementation of a control structure known as the *case structure*, which lets you code different actions for different cases. The switch statement can sometimes be used in place of an if statement with else if clauses. However, since the switch statement can only be used with expressions that evaluate to an integer, this statement has limited use.

To code a switch statement, you start by coding the switch keyword followed by a switch expression that evaluates to one of the integer types. After the switch expression, you can code one or more *case labels* that represent the possible values of the switch expression. Then, when the switch expression matches the value specified by the case label, the statements after the label are executed.

You can code the case labels in any sequence, but you should be sure to follow each label with a colon. Then, if the label contains one or more statements, you can code a *break statement* after them to jump to the end of the switch statement. Otherwise, the execution of the program *falls through* to the next case label and executes the statements in that label. The *default label* is an optional label that identifies the statements to execute if none of the case labels are executed.

The first example in this figure shows how to code a switch statement that sets the description for a product based on the value of an int variable named productID. Here, the first case label assigns a value of “Hammer” to the productDescription variable if productID is equal to 1. Then, the break statement exits the switch statement. Similarly, the second case label sets the product description to “Box of Nails” if productID is equal to 2 and then exits the switch statement. If productID is equal to something other than 1 or 2, the default case label is executed. Like the other two case labels, this one sets the value of the productDescription variable and then exits the switch statement.

The second example shows how to code a switch statement that sets a day variable to “weekday” or “weekend” depending on the value of the integer in the variable named dayOfWeek. Here, the first break statement is coded after the case labels for 2, 3, 4, 5, and 6. As a result, day is set to “weekday” for any of those values. Similarly, whenever dayOfWeek equals 1 or 7, day is set to “weekend”.

Although the last case label in both of these examples includes a break statement, that isn’t necessary. If you omit it, program execution falls through to the statement that follows the switch statement. Even so, it’s a good programming practice to code a break statement after the last case label.

When you code switch statements, you can nest one statement within another. You can also nest if/else statements within switch statements and switch statements within if/else statements. Here again, you should try to code the statements with a logical structure that is relatively easy to understand. If necessary, you can also add comments that clarify the logic of your code.

## The syntax of the switch statement

```
switch (integerExpression)
{
    case label1:
        statements
        break;
    case label2:
        statements
        break;
    any other case statements
    default: (optional)
        statements
        break;
}
```

### Example 1: A switch statement with a default label

```
switch (productID)
{
    case 1:
        productDescription = "Hammer";
        break;
    case 2:
        productDescription = "Box of Nails";
        break;
    default:
        productDescription = "Product not found";
        break;
}
```

### Example 2: A switch statement that falls through case labels

```
switch (dayOfWeek)
{
    case 2:
    case 3:
    case 4:
    case 5:
    case 6:
        day = "weekday";
        break;
    case 1:
    case 7:
        day = "weekend";
        break;
}
```

## Description

- The switch statement can only be used with an expression that evaluates to one of these integer types: char, byte, short, or int. The case labels represent the integer values of that expression, and these labels can be coded in any sequence.
- The switch statement transfers control to the appropriate *case label*. If control isn't transferred to one of the case labels, the optional *default label* is executed.
- If a case label doesn't contain a break statement, code execution will *fall through* to the next label. Otherwise, the break statement ends the switch statement.

## **An enhanced version of the Invoice application**

---

To give you a better idea of how if/else statements can be used, figure 4-6 presents another enhanced version of the Invoice application. This time, the console prompts the user for two entries: customer type and subtotal.

In this application, if the user enters “R” or “C” for the customer type, the discount percent changes depending on the value of the subtotal. If, for example, the customer type is “R” and the subtotal is greater than or equal to 250, the discount percent is .2. Or, if the customer type is “C” and the subtotal is less than 250, the discount percent is .2.

Here, you can see that the conditions are coded in a logical order. For instance, the expressions in the nested if statement for customer type “R” go from a subtotal that’s less than 100, to a subtotal that’s greater than or equal to 100, to a subtotal that’s greater than or equal to 250. That covers all of the possible subtotals from the smallest to the largest. Although you could code these conditions in other sequences, this sequence makes it easy to tell that all possibilities have been covered.



## The console

```
Enter customer type (r/c): r
Enter subtotal: 100
Discount percent: 10%
Discount amount: $10.00
Total: $90.00

Continue? (y/n):
```

## The code

```
import java.text.NumberFormat;
import java.util.Scanner;

public class InvoiceApp
{
    public static void main(String[] args)
    {
        Scanner sc = new Scanner(System.in);
        String choice = "y";

        while (!choice.equalsIgnoreCase("n"))
        {
            // get the input from the user
            System.out.print("Enter customer type (r/c): ");
            String customerType = sc.next();
            System.out.print("Enter subtotal:   ");
            double subtotal = sc.nextDouble();

            // get the discount percent
            double discountPercent = 0.0;
            if (customerType.equalsIgnoreCase("R"))
            {
                if (subtotal < 100)
                    discountPercent = 0;
                else if (subtotal >= 100 && subtotal < 250)
                    discountPercent = .1;
                else if (subtotal >= 250)
                    discountPercent = .2;
            }
            else if (customerType.equalsIgnoreCase("C"))
            {
                if (subtotal < 250)
                    discountPercent = .2;
                else
                    discountPercent = .3;
            }
            else
                discountPercent = .1;

            // the code to calculate, format, and display results goes here

            // the code to see if the user wants to continue goes here

        }
    }
}
```

Figure 4-6 The enhanced Invoice application

## How to code loops

---

In chapter 2, you learned how to code while statements and while loops. Now, you'll review the coding for those loops and learn how to code two other Java statements that implement the *iteration structure*.

### How to code while and do-while loops

---

Figure 4-7 shows how to use the *while statement* to code a *while loop*. Then, it shows how to code a *do-while loop*. The difference between these types of loops is that the Boolean expression is evaluated at the beginning of a while loop and at the end of a do-while loop. As a result, the statements in a while loop are executed zero or more times while the statements in a do-while loop are always executed at least once.

When coding while loops, it's common to use a *counter variable* to execute the statements in a loop a certain number of times. The first loop in this figure, for example, uses an int counter variable named *i* that's initialized to 1. Then, the last statement in the loop increments the counter variable with each iteration of the loop. As a result, the first statement in this loop will be executed as long as the counter variable is less than or equal to 36. As I've mentioned earlier, it is a common coding practice to name counter variables with single letters like *i*, *j*, and *k*.

Most of the time, you can use either of these two types of loops to accomplish the same task. For instance, the first example in this figure uses a while loop to calculate the future value of a series of monthly payments at a specified interest rate, and the second example uses a do-while loop to perform the same calculation.

When coding loops, it's important to remember that the code within a loop has block scope. As a result, any variables that are declared within the loop can't be used outside of the loop. That's why the variables that are needed outside of the loop have been declared outside of the loop. That way, you can use these variables after the loop has finished executing.

When you code loops, you usually want to avoid *infinite loops*. If, for example, you forget to code a statement that increments the counter variable, the loop will never end. Then, you have to press Ctrl+C or close the console to cancel the application so you can debug your code.

## The syntax of the while loop

```
while (booleanExpression)
{
    statements
}
```

### A while loop that calculates a future value

```
int i = 1;
int months = 36;
while (i <= months)
{
    futureValue = (futureValue + monthlyPayment) *
        (1 + monthlyInterestRate);
    i++;
}
```

## The syntax of the do-while loop

```
do
{
    statements
}
while (booleanExpression);
```

### A do-while loop that calculates a future value

```
int i = 1;
int months = 36;
do
{
    futureValue = (futureValue + monthlyPayment) *
        (1 + monthlyInterestRate);
    i++;
}
while (i <= months);
```

## Description

- In a *while loop*, the condition is tested before the loop is executed. In a *do-while loop*, the condition is tested after the loop is executed.
- A while or do-while loop executes the block of statements within the loop as long as its Boolean expression is true.
- If a loop requires more than one statement, you must enclose the statements in braces. This identifies the block of statements that are executed by the loop, and any variables or constants that are declared in that block have block scope.
- If a loop requires just one statement, you don't have to enclose the statement in braces. However, that statement can't declare a variable or it won't compile.
- If the condition at the start of a while statement never becomes false, the statement never ends. Then, the program goes into an *infinite loop*. You can cancel an infinite loop by closing the console window or pressing Ctrl+C.

## How to code for loops

---

Figure 4-8 shows how to use the `for` statement to code *for loops*. This type of loop is useful when you need to increment or decrement a counter that determines how many times the loop is going to be executed.

To code a `for` loop, you start by coding the `for` keyword followed by three expressions enclosed in parentheses and separated by semicolons. The first expression is an initialization expression that gives the starting value for the counter variable. This expression can also declare the counter variable, if necessary. The second expression is a Boolean expression that determines when the loop will end. And the third expression is an increment expression that determines how the counter is incremented or decremented each time the loop is executed.

The first example in this figure shows how to use these expressions. First, the initialization expression declares the counter variable that's used to determine the number of loops and assigns an initial value to it. In this example, the counter variable is an `int` type named `i`, and it's initialized to 0. Next, a Boolean expression specifies that the loop will be repeated as long as the counter is less than 5. Then, the increment expression increments the counter by 1 at the end of each repetition of the loop.

Since the two loops in this example store the counter variable followed by a space in a string, this code stores the numbers 0 to 4 in a string variable like this:

```
0 1 2 3 4
```

Notice that you can code this loop using a single statement or using multiple statements. If you use more than one statement, though, you must enclose those statements in braces.

The second example calculates the sum of 8, 6, 4, and 2. Here, the `sum` variable is declared before the loop so it will be available outside of the loop. Within the parentheses of the `for` loop, the initialization expression initializes the counter variable to 8, the Boolean expression indicates that the loop will end when the counter variable is no longer greater than zero, and the increment expression uses an assignment operator to subtract 2 from the counter variable with each repetition of the loop. Within the loop, the value of the counter variable is added to the value that's already stored in the `sum` variable. As a result, the final value for the `sum` variable is 20.

The third example shows how to code a loop that calculates the future value for a series of monthly payments. Here, the loop executes one time for each month. If you compare this example with the examples in the previous figure, you can see how a `for` loop improves upon a `while` or `do-while` loop when a counter variable is required.

## The syntax of the for loop

```
for(initializationExpression; booleanExpression; incrementExpression)
{
    statements
}
```

### Example 1: A for loop that stores the numbers 0 through 4 in a string

#### With a single statement

```
String numbers = "";
for (int i = 0; i < 5; i++)
    numbers += i + " ";
```

#### With a block of statements

```
String numbers = "";
for (int i = 0; i < 5; i++)
{
    numbers += i;
    numbers += " ";
}
```

### Example 2: A for loop that adds the numbers 8, 6, 4, and 2

```
int sum = 0;
for (int j = 8; j > 0; j -= 2)
{
    sum += j;
}
```

### Example 3: A for loop that calculates a future value

```
for (int i = 1; i <= months; i++)
{
    futureValue = (futureValue + monthlyPayment) *
        (1 + monthlyInterestRate);
}
```

## Description

- A *for loop* is useful when you need to increment or decrement a counter that determines how many times the loop is executed.
- Within the parentheses of a for loop, you code an initialization expression that gives the starting value for the counter, a Boolean expression that determines when the loop ends, and an increment expression that increments or decrements the counter.
- The loop ends when the Boolean expression is false.
- If necessary, you can declare the counter variable before the for loop. Then, this variable will be in scope after the loop finishes executing.

## The Future Value application

---

Now that you've learned the statements for coding loops, figure 4-9 presents an application that uses a for loop within a while loop. As the console for this application shows, the user starts by entering the values for the monthly payment that will be made, the yearly interest rate, and the number of years the payment will be made. Then, for each group of entries, the application calculates and displays the future value.

If you look at the code for this application, you can see that it uses a while loop to determine when the program will end. Within this loop, the program first gets the three entries from the user. Next, it converts these entries to the same time unit, which is months. To do that, the number of years is multiplied by 12, and the yearly interest rate is divided by 12. Besides that, the yearly interest rate is divided by 100 so it will work correctly in the future value calculation.

Once those variables are prepared, the program enters a for loop that calculates the future value. When the loop finishes, the program displays the result and asks whether the user wants to continue.

Because this application doesn't validate the user's entries, it will crash if the user enters invalid data. But you'll learn how to fix that in the next chapter. Otherwise, this application works the way you would want it to. In this case, rounding isn't an issue because the result is rounded just one time after the future value loop has finished.

Because it can be hard to tell whether an application with a loop is producing the right results, it often makes sense to add debugging statements within the loop while you're testing it. For instance, you could add this statement to the Future Value application as the last statement in the loop:

```
System.out.println("Debug: " + i + "      " + futureValue);
```

Then, one line will be displayed on the console each time through the loop so you can check to make sure that the calculations for the first few months are accurate. You will also be able to tell at a glance whether the loop was executed the right number of times.

## The console

```
Enter monthly investment: 100
Enter yearly interest rate: 3
Enter number of years: 3
Future value: $3,771.46

Continue? (y/n): y
```

## The code

```
import java.util.Scanner;
import java.text.NumberFormat;

public class FutureValueApp
{
    public static void main(String[] args)
    {
        Scanner sc = new Scanner(System.in);
        String choice = "y";
        while (!choice.equalsIgnoreCase("n"))
        {
            // get the input from the user
            System.out.print("Enter monthly investment: ");
            double monthlyInvestment = sc.nextDouble();
            System.out.print("Enter yearly interest rate: ");
            double interestRate = sc.nextDouble();
            System.out.print("Enter number of years: ");
            int years = sc.nextInt();

            // convert yearly to monthly values and initialize future value
            double monthlyInterestRate = interestRate/12/100;
            int months = years * 12;
            double futureValue = 0.0;

            // use a for loop to calculate the future value
            for (int i = 1; i <= months; i++)
            {
                futureValue =
                    (futureValue + monthlyInvestment) *
                    (1 + monthlyInterestRate);
            }

            // format and display the result
            NumberFormat currency = NumberFormat.getCurrencyInstance();
            System.out.println("Future value: "
                + currency.format(futureValue));
            System.out.println();

            // see if the user wants to continue
            System.out.print("Continue? (y/n): ");
            choice = sc.next();
            System.out.println();
        }
    }
}
```

Figure 4-9 The code for the Future Value application

## How to code nested for loops

---

Like if and switch statements, you can also nest for loops, and that's illustrated by the code in figure 4-10. As with all nested statements, you should use indentation to clearly show the relationships between the statements.

The example in this figure shows how to use three nested for loops to display a table of future value calculations on the console. Here, the amount of the monthly investment is set to \$100, the interest rates vary from 5.0% to 6.5%, and the number of years varies from 2 years to 4 years. Before the nested for loops are executed, another for loop adds the headings to the table string.

After that, the three nested loops add one row for each year to the table string. To do that, the outer loop iterates through the years (4, 3, and 2). Then, the code adds the year to the start of the row string. After that, the next loop iterates through the four interest rates (5%, 5.5%, 6%, and 6.5%) using the innermost loop to calculate the future value for each interest rate and to append these calculations to the row string. When this loop is finished, the top-level loop appends the row to the table string and clears the row string so it can be used again in the next iteration of the loop.

When all of the loops have been completed, the `println` method prints the table string to the console. This shows the lowest future value amount in the bottom left corner and the highest future value amount in the top right corner. To align these interest rates, this code uses spaces. Although it would be possible to use tab characters to align each column, tabs don't always align columns correctly.



## The console

Monthly Payment: 100.0

	5.0%	5.5%	6.0%	6.5%
4	\$5,323.58	\$5,379.83	\$5,436.83	\$5,494.59
3	\$3,891.48	\$3,922.23	\$3,953.28	\$3,984.64
2	\$2,529.09	\$2,542.46	\$2,555.91	\$2,569.45

## Nested for loops that print a table of future values

```
// get the currency and percent formatters
NumberFormat currency = NumberFormat.getCurrencyInstance();
NumberFormat percent = NumberFormat.getPercentInstance();
percent.setMinimumFractionDigits(1);

// set the monthly payment to 100 and display it to the user
double monthlyPayment = 100.0;
System.out.println("Monthly Payment: " + monthlyPayment);
System.out.println();

// declare a variable to store the table
String table = " ";

// fill the first row of the table
for (double rate = 5.0; rate < 7.0; rate += .5)
{
    table += percent.format(rate/100) + " ";
}
table += "\n";

// loop through each row
for (int years = 4; years > 1; years--)
{
    // append the years variable to the start of the row
    String row = years + " ";
    // loop through each column
    for (double rate = 5.0; rate < 7.0; rate += .5)
    {
        // calculate the future value for each rate
        int months = years * 12;
        double monthlyInterestRate = rate/12/100;
        double futureValue = 0.0;
        for (int i = 1; i <= months; i++)
        {
            futureValue =
                (futureValue + monthlyPayment) *
                (1 + monthlyInterestRate);
        }
        // add the calculation to the row
        row += currency.format(futureValue) + " ";
    }
    table += row + "\n";
    row = "";
}

// display the table to the user
System.out.println(table);
```

Figure 4-10 How to code nested for loops that calculate future values

## How to code break and continue statements

---

When you code loops, you usually want them to run to completion. Occasionally, though, an application may require that you jump out of a loop. To do that, you can use the `break` or `continue` statement.

### How to code break statements

---

Figure 4-11 shows how to use the `break` statement and the labeled `break` statement to exit loops. If you need to exit the current loop, you can code a `break` statement. If you need to exit another loop in a set of nested loops, you can use the labeled `break` statement.

The first example shows how you can use the `break` statement to exit from an inner loop. Here, a `while` loop that generates random numbers is nested within a `for` loop. Notice that the Boolean expression for the `while` loop has been set to `true`. Because of that, this loop would execute indefinitely without a statement that explicitly jumps out of the loop. In this case, a `break` statement is used to exit from the loop when the random number that's generated is greater than 7. Then, control is returned to the `for` loop, which is executed until its Boolean expression is satisfied.

The second example shows how you can use the labeled `break` statement to exit an outer loop from an inner loop. To use a labeled `break` statement, you code a *label* for the loop that you want to exit. Then, to break out of the outer loop, you just type the `break` statement followed by the name of the label. This will transfer control to the statement that follows the outer loop.

## The syntax of the break statement

```
break;
```

### Example 1: A break statement that exits the inner loop

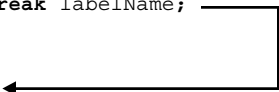
```
for (int i = 1; i < 4; i++)
{
    System.out.println("Outer " + i);
    while (true)
    {
        int number = (int) (Math.random() * 10);
        System.out.println("    Inner " + number);
        if (number > 7)
            break;
    }
}
```

## The syntax of the labeled break statement

```
break labelName;
```

### The structure of the labeled break statement

```
labelName:
loop declaration
{
    statements
    another loop declaration
    {
        statements
        if (conditionalExpression)
        {
            statements
            break labelName;
        }
    }
}
```



### Example 2: A labeled break statement that exits the outer loop

```
outerLoop:
for (int i = 1; i < 4; i++)
{
    System.out.println("Outer " + i);
    while (true)
    {
        int number = (int) (Math.random() * 10);
        System.out.println("    Inner " + number);
        if (number > 7)
            break outerLoop;
    }
}
```

## Description

- To jump to the end of the current loop, you can use the break statement.
- To jump to the end of an outer loop from an inner loop, you can label the outer loop and use the labeled break statement. To code a *label*, type the name of the label and a colon before a while, do-while, or for loop.

Figure 4-11 How to code break statements

## How to code continue statements

---

Figure 4-12 shows how to use the continue statement and labeled continue statement to jump to the beginning of a loop. These statements work similarly to the break statements, but they jump to the beginning of a loop instead of the end of a loop. Like the break statements, you can use the unlabeled version of the statement to work with the current loop and you can use the labeled version of the statement to work with nested loops.

The first example shows how to use the continue statement to print 9 random numbers. In this example, the loop generates random numbers from 0 through 10 and prints them to the console. If the random number is less than or equal to 7, though, the continue statement jumps to the beginning of the loop. As a result, the println method that comes after the continue statement is only executed when the random number is greater than 7.

The second example shows how to use the labeled continue statement to print the prime numbers from 1 through 19. In this example, the outer loop loops through the numbers 1 through 19, while the inner loop loops through all numbers from 2 through the outer number minus 1. Then, the remainder variable is set equal to the remainder of the outer loop counter divided by the inner loop counter. If the remainder equals 0, the continue statement causes control of the program to jump to the top of the outer loop. As a result, the outer loop continues with the next number. But if the remainder doesn't equal 0 at any point in the inner loop, which means the number is a prime number, the program finishes the inner loop and the println method prints the number to the console.

## The syntax of the continue statement

```
continue;
```

### Example 1: A continue statement that jumps to the beginning of a loop

```
for (int j = 1; j < 10; j++)
{
    int number = (int) (Math.random() * 10);
    System.out.println(number);
    if (number <= 7)
        continue;
    System.out.println("This number is greater than 7");
}
```

## The syntax of the labeled continue statement

```
continue labelName;
```

## The structure of the labeled continue statement

```
labelName: ←
loop declaration
{
    statements
    another loop declaration
    {
        statements
        if (conditionalExpression)
        {
            statements
            continue labelName;
        }
    }
}
```

### Example 2: A labeled continue statement that jumps to the beginning of the outer loop

```
outerLoop:
for(int i = 1; i < 20; i++)
{
    for(int j = 2; j < i-1; j++)
    {
        int remainder = i%j;
        if (remainder == 0)
            continue outerLoop;
    }
    System.out.println(i);
}
```

## Description

- To skip the rest of the statements in the current loop and jump to the top of the current loop, you can use the continue statement.
- To skip the rest of the statements in the current loop and jump to the top of a labeled loop, you can add a label to the loop and use the labeled continue statement.
- To code a label, type the name of the label and a colon before a while, do-while, or for loop.

Figure 4-12 How to code continue statements

## How to code and call static methods

---

So far, you've learned how to code applications that consist of a single method, the static main method that's executed automatically when you run a class. Now, you'll learn how to code and call other static methods. That's one way to divide the code for an application into manageable parts.

### How to code static methods

---

Figure 4-13 shows how to code a *static method*. To start, you code an *access modifier* that indicates whether the method can be called from other classes (public) or just the class that it's coded in (private). Next, you code the static keyword to identify the method as a static method.

After the static keyword, you code a return type that identifies the type of data that the method will return. That return type can be either a primitive data type or a class like the String class. If the method doesn't return any data, you code the void keyword.

After the return type, you code a method name that indicates what the method does. A common coding convention is to use camel notation and to start each method name with a verb followed by a noun or by an adjective and a noun, as in calculateFutureValue.

After the method name, you code a set of parentheses. Within the parentheses, you declare the *parameters* that are required by the method. If a method doesn't require any parameters, you can code an empty set of parentheses. And if a method requires more than one parameter, you separate them with commas as shown by the second example. Later on, when you call the method, you pass arguments that correspond to these parameters.

At this point, you code a set of braces that contains the statements that the method will execute. If the method is going to return a value, these statements must include a *return statement* that identifies the variable or object to be returned. This is illustrated by the calculateFutureValue method in this figure.

When you code the method name and parameter list of a method, you form the *signature* of the method. As you might expect, each method must have a unique signature. However, you can code two or more methods with the same name but with different parameters. This is known as *overloading* a method, and you'll learn more about that in chapter 6.

### How to call static methods

---

Figure 4-13 also shows how to *call* a static method that's coded within the same class. This is just like calling a static method from a Java class, but you don't need to code the class name. Then, if the method requires *arguments*, you code the arguments within parentheses, separating each argument with a comma. Otherwise, you code an empty set of parentheses.

## The basic syntax for coding a static method

```
{public|private} static returnType methodName([parameterList])
{
    statements
}
```

### A static method with no parameters and no return type

```
private static void printWelcomeMessage()
{
    System.out.println("Hello New User"); // This could be a lengthy message
}
```

### A static method with three parameters that returns a double value

```
public static double calculateFutureValue(double monthlyInvestment,
double monthlyInterestRate, int months)
{
    double futureValue = 0.0;
    for (int i = 1; i <= months; i++)
    {
        futureValue = (futureValue + monthlyInvestment)
            * (1 + monthlyInterestRate);
    }
    return futureValue;
}
```

## The syntax for calling a static method that's in the same class

```
methodName([argumentList])
```

### A call statement with no arguments

```
printWelcomeMessage();
```

### A call statement that passes three arguments

```
double futureValue = calculateFutureValue(investment, rate, months);
```

## Description

- To allow other classes to access a method, use the public *access modifier*. To prevent other classes from accessing a method, use the private modifier.
- To code a method that returns data, code a return type in the method declaration and code a *return statement* in the body of the method. The return statement ends the execution of the method and returns the specified value to the calling method.
- Within the parentheses of a method, you can code an optional *parameter list* that contains one or more *parameters* that consist of a data type and name. These are the values that must be passed to the method when it is called.
- The name of a method along with its parameter list form the *signature* of the method, which must be unique.
- When you call a method, the *arguments* in the *argument list* must be in the same order as the parameters in the parameter list defined by the method, and they must have compatible data types. However, the names of the arguments and the parameters don't need to be the same.

In practice, the terms *parameter* and *argument* are often used interchangeably. In this book, however, we'll use the term *parameter* to refer to the variables of a method declaration, and we'll use the term *argument* to refer to the variables that are passed to a method.

## **The Future Value application with a static method**

---

To illustrate the use of static methods, figure 4-14 presents another version of the Future Value application. This time it uses a static method to calculate the future value. This method requires three arguments, and it includes the for loop that processes those arguments. When the loop finishes, the return statement returns the future value to the main method.

To use this statement, the body of the program prepares the three arguments in month units. Then, it calls the method and passes the three arguments to it. This simplifies the body of the program, and this illustrates how static methods can be used to divide a program into manageable components.

In this case, the call statement passes arguments that have the same variable names as the parameters of the method. Although this isn't necessary, it makes the code easier to follow. What is necessary, though, is that the arguments be passed in the same sequence as the parameters and with the same data types.

In the next chapter, you'll see other ways that static methods can be used. Then, in chapter 6, you'll see how static methods can be coded with the `public` keyword so they can be accessed by other classes. For now, though, you can code all of your static methods with the `private` access modifier.



## The code

```
import java.util.Scanner;
import java.text.NumberFormat;

public class FutureValueApp
{
    public static void main(String[] args)
    {
        Scanner sc = new Scanner(System.in);
        String choice = "y";
        while (!choice.equalsIgnoreCase("n"))
        {
            // get the input from the user
            System.out.print("Enter monthly investment:  ");
            double monthlyInvestment = sc.nextDouble();
            System.out.print("Enter yearly interest rate: ");
            double interestRate = sc.nextDouble();
            System.out.print("Enter number of years:      ");
            int years = sc.nextInt();

            // convert yearly values to monthly values
            double monthlyInterestRate = interestRate/12/100;
            int months = years * 12;

            // call the future value method
            double futureValue = calculateFutureValue(
                monthlyInvestment, monthlyInterestRate, months);

            // format and display the result
            NumberFormat currency = NumberFormat.getCurrencyInstance();
            System.out.println("Future value:          "
                + currency.format(futureValue));
            System.out.println();

            // see if the user wants to continue
            System.out.print("Continue? (y/n): ");
            choice = sc.next();
            System.out.println();
        }
    }

    // a static method that requires three arguments and returns a double
    private static double calculateFutureValue(double monthlyInvestment,
        double monthlyInterestRate, int months)
    {
        double futureValue = 0.0;
        for (int i = 1; i <= months; i++)
        {
            futureValue =
                (futureValue + monthlyInvestment) *
                (1 + monthlyInterestRate);
        }
        return futureValue;
    }
}
```

Figure 4-14 The Future Value application with a static method

## Perspective

---

If this chapter has succeeded, you should now be able to use *if*, *switch*, *while*, *do-while*, and *for* statements. These are the Java statements that implement the selection, case, and iteration structures, and they provide the logic of an application. You should also be able to code and call your own static methods, which will help you divide your programs into manageable parts.

## Summary

---

- You can use the *relational operators* to create *Boolean expressions* that compare primitive data types and return true or false values, and you can use the *logical operators* to connect two or more Boolean expressions.
- To determine whether two strings are equal, you can call the `equals` and `equalsIgnoreCase` methods from a `String` object.
- You can use *if/else statements* and *switch statements* to control the logic of an application, and you can *nest* these statements whenever necessary.
- You can use *while*, *do-while*, and *for loops* to repeatedly execute one or more statements until a Boolean expression evaluates to false, and you can nest these statements whenever necessary.
- You can use *break statements* to jump to the end of the current loop or a labeled loop, and you can use *continue statements* to jump to the start of the current loop or a labeled loop.
- To code a *static method*, you code an access modifier, the static keyword, its return type, its name, and a *parameter* list. Then, to return a value, you code a *return statement* within the method.
- To call a static method that's in the same class as the main method, you code the method name followed by an *argument* list.

## Exercise 4-1 Test the Future Value application

In this exercise, you'll test the Future Value application that's presented in figure 4-9 in this chapter.

1. Open the FutureValueApp class stored in the ch04 directory. Then, compile and test it with valid data to see how it works.
2. To make sure that the results are correct, add a debugging statement within the for loop that calculates the future value. This statement should display the month and future value each time through the loop. Then, test the program with simple entries like 100 for monthly investment, 12 for yearly interest (because that's 1 percent each month), and 1 for year. When the debugging data is displayed, check the results manually to make sure they're correct.

## Exercise 4-2 Enhance the Invoice application

In this exercise, you'll modify the nested if/else statements that are used to determine the discount percent for the Invoice application in figure 4-6. Then, you'll code and call a static method that determines the discount percent.

### Open the application and change the if/else statement

1. Open the application named CodedInvoiceApp that's in the ch04 directory, save it as EnhancedInvoiceApp, and change the class name. Then, compile and run the application to see how it works.
2. Change the if/else statement so customers of type "R" with a subtotal that is greater than or equal to \$250 but less than \$500 get a 25% discount and those with a subtotal of \$500 or more get a 30% discount. Next, change the if/else statement so customers of type "C" always get a 20% discount. Then, test the application to make sure this works.
3. Add another customer type to the if/else statement so customers of type "T" get a 40% discount for subtotals of less than \$500, and a 50% discount for subtotals of \$500 or more. Then, test the application.
4. Check your code to make sure that no discount is provided for a customer type code that isn't "R", "C", or "T". Then, fix this if necessary.

### Code and call a static method that determines the discount percent

5. Code a static method named getDiscountPercent that has two parameters: customer type and subtotal. To do that efficiently, you can move the appropriate code from the main method of the application into the static method and make the required modifications.
6. Add code that calls the static method from the body of the application. Then, test to make sure that it works.

## Exercise 4-3    Enhance the Test Score application

In this exercise, you'll enhance the Test Score application so it uses a while or a do-while loop plus a for loop. After the enhancements, the console for a user's session should look something like this:

```
Enter the number of test scores to be entered: 5
Enter score 1: 75
Enter score 2: 80
Enter score 3: 75
Enter score 4: 880
Invalid entry, not counted
Enter score 4: 80
Enter score 5: 95

Score count:    5
Score total:    405
Average score:  81
Minimum score:  75
Maximum score:  95

Enter more test scores? (y/n): y
Enter the number of test scores to be entered: 3
Enter score 1: 85
Enter score 2: 95
Enter score 3: 100

Score count:    3
Score total:    280
Average score:  93.3
Minimum score:  85
Maximum score:  100

Enter more test scores? (y/n):
```

1. Open the `EnhancedTestScoreApp` in `ch03` that you developed for exercise 3-2. If you didn't do that exercise, you can work from an earlier version like the `ModifiedTestScoreApp` or the `TestScoreApp` that's in `ch02`. Then, save the application in the `ch04` directory as `EnhancedTestScoreApp`, change the class name if necessary, and compile and test the application.
2. Change the while statement to a do-while statement, and test this change. Does this work any better than the while loop?
3. Enhance the program so it uses a while or do-while loop that controls whether the user enters more than one set of test scores. The statements in this loop should first ask the user how many test scores are going to be entered. Then, this number should be used in a for loop that gets that many test score entries from the user. When the loop ends, the program should display the summary data for the test scores, determine whether the user wants to enter another set of scores, and repeat the while loop if necessary. After you make these enhancements, test them to make sure they work.
4. If you didn't already do it, make sure that the code in the for loop doesn't count an invalid entry. In that case, an error message should be displayed and the counter decremented by one. Now, test to make sure this works.

# Appendix A

## **How to use the downloadable files for this book**

The single topic in this appendix describes the files for this book that are available for download from our web site and tells you how you can use them.

## How to use the downloadable files

---

Throughout this book, complete applications have been used to illustrate the material presented in each chapter. To help you understand these applications, you can download the source code and data for some of these applications from our web site. Then, you can view the source code and run them.

These files come in a single download that also includes the source code and data you'll need for the exercises at the end of each chapter. Figure A-1 describes how you download, install, and use these files.

When you download the single install file and execute it, it will install all of the files for this book in the `murach\java6` directory on your C drive. Within this directory, you'll find a directory named `Exercises` that contains a directory named `java1.6` that contains all of the files and directories you'll need for the exercises presented in this book. Before you use these files and directories, though, you'll want to copy (not move) the `java1.6` directory that contains them to the root directory of your C drive. That way, it will be easy to locate the files and directories you need as you're working on the exercises. In addition, if you make a mistake and want to restore a file to its original state, you can do that by copying it from the directory where it was originally installed.

You'll also find a directory named `Applications` within the `c:\murach\java6` directory. This directory contains the source code for the applications presented in this book. Within this directory, you'll find chapter directories, and within the chapter directories you'll find a directory that contains the source files for each application. You can view the source code for these applications as you read each chapter, and you can compile and run these applications to experiment with them.

If your computer doesn't allow you to run the self-extracting zip file (which is an exe file), you can download a zip file from our web site that contains the same files as the self-extracting zip file. Then, you can extract the files that are contained in this zip file to the `c:\murach` directory. If this directory doesn't already exist, you can create it yourself.

## What the downloadable files for this book contain

- The source code and data for selected applications presented in the book
- The starting source code and data for all of the exercises included in the book

## How to download and install the files for this book

- Go to [www.murach.com](http://www.murach.com), and go to the page for *Murach's Java SE 6*.
- Click the link for “FREE download of the book applications.” Then, click the “All book files” link for the self-extracting zip file. This will download one file named `jse6_allfiles.exe` to the root directory of your C drive.
- Use the Windows Explorer to find the exe file on your C drive. Then, double-click this file and respond to the dialog boxes that follow. This installs the files in directories that start with `c:\murach\java6`.

## How to prepare your system for doing the exercises

- Some of the exercises have you start from existing applications. The source code for these applications is in the `c:\murach\java6\exercises\java1.6` directory. Before you do the exercises, you'll want to copy (not move) the `java1.6` subdirectory and all of its subdirectories to the root directory of the C drive. From that point on, you can find the programs and files that you need in directories like `c:\java1.6\ch01` and `c:\java1.6\ch02`. When an application requires a file or database, you'll find it within the chapter or project directory.

## How to use the source code for the applications presented in this book

- The source code for the applications presented in this book can be found in the subdirectories of the `c:\murach\java6\applications` directory. Then, you can view, compile, and run the source code for each application in each chapter of this book.

## How to use a zip file instead of a self-extracting zip file

- Although we recommend using the self-extracting zip file (`jse6_allfiles.exe`) to install the downloadable files as described above, some systems won't allow self-extracting zip files to run. In that case, you can download a regular zip file (`jse6_allfiles.zip`) from our web site. Then, you can extract the files stored in this zip file into the `c:\murach` directory. If the `c:\murach` directory doesn't already exist, you will need to create it.

## Notes for other versions of the JDK

- If you're using JDK 1.6 (Java SE 6) or later, you should be able to compile and run all of these applications.
- If you're using an earlier version of the JDK, you can still view the source code, but you won't be able to compile and run applications that use the features of Java introduced with later versions of the JDK. To solve this problem, you can download and install the JDK 1.6 as described in chapter 1.

# Index

- operator, 96, 97
- operator, 96, 97
- ! operator, 126, 127
- != operator, 75, 122-125
- % operator, 96, 97
- %= operator, 99
- & operator, 126, 127
- && operator, 126, 127
- \* operator, 96, 97
- \*= operator, 99
- / operator, 96, 97
- /= operator, 99
- @author tag (javadoc), 304, 305
- @param tag (javadoc), 304, 305
- @return tag (javadoc), 304, 305
- @see tag (javadoc), 304, 305
- @version tag (javadoc), 304, 305
- | operator, 126, 127
- || operator, 126, 127
- + operator
  - addition, 96, 97
  - concatenation, 58, 59
  - positive sign, 96, 97
- ++ operator, 96, 97
- += operator
  - assignment, 99
  - concatenation, 58, 59
- < operator, 75, 122, 123
- <= operator, 75, 122, 123
- = operator, 99
- = operator, 99
- == operator, 75, 122-125
- > operator, 75, 122, 123
- >= operator, 75, 122, 123

## A

- absolute method (ResultSet interface), 736, 737
- absolute path, 626, 627
- Abstract class, 252, 253
  - compared to interface, 264, 265
- abstract keyword, 252, 253
  - with interfaces, 268, 269
- Abstract method, 252, 253
- AbstractButton class (javax.swing), 516, 518
- Access modes (random-access file), 660, 661
- Access modifiers, 50, 51, 146, 147, 232, 233
  - instance variable, 186
  - method, 190, 191
  - static field, 206, 207
- Access database, 714, 715

- Action event
  - combo box, 524
  - handling, 492, 493
- Action listener
  - check box, 516, 517
  - combo box, 524, 525
  - radio button, 518, 519
- Action query, 724
- ActionEvent object, 548, 549
- ActionListener interface (java.awt.event), 266, 267, 492, 493, 524, 525
- ActionListener object, 548, 549
- actionPerformed method (ActionListener interface), 492, 493
  - check box, 516, 517
  - combo box, 524, 525
- Adapter classes, 570, 571
- add method (ArrayList class), 352-355
- add method (BigDecimal class), 114, 115
- add method (BorderLayout class), 496, 497
- add method (ButtonGroup class), 518, 519
- add method (Calendar class), 390, 391
- add method (Container class), 488, 489
- add method (Container class), 534, 535
- add method (GregorianCalendar class), 390, 391
- add method (JFrame class), 488, 489
- add method (LinkedList class), 362-365
- addActionListener method (JButton class), 492, 493
- addActionListener method (JCheckBox class), 516, 517
- addActionListener method (JComboBox class), 522, 523
- addActionListener method (JRadioButton class), 518, 519
- addComponentListener method (Component class), 564, 565
- addElement method (DefaultListModel class), 530, 531
- addFirst method (LinkedList class), 362-365
- addFocusListener method (Component class), 564, 565
- Adding records to a database, 724, 725, 740, 741,
- addItem method (JComboBox class), 522, 523
- addItemListener method (JComboBox class), 522, 523
- Addition assignment operator, 99
- Addition operator, 96, 97
- addKeyListener method (Component class), 564, 565
- addLast method (LinkedList class), 362-365
- addMouseListener method (Component class), 564, 565
- addMouseMotionListener method (Component class), 564, 565
- addWindowListener method (Window class), 564, 565
- afterLast method (ResultSet interface), 736, 737
- Altova XMLSpy, 688, 689
- anchor field (GridBagConstraints class), 536, 537
- And operator, 126, 127
- Anonymous inner class, 562, 563
- ANSI format, 20
- Apache Derby, see Derby database
- API (Application Programming Interface), 32, 33



- API documentation
  - installing, 32, 33
  - navigating, 34, 35
  - using, 66, 67
- append method (JTextArea class), 512, 513
- append method (StringBuilder class), 406-409
- Appending data to a file, 636, 637
- Appending strings, 58, 59
- Applet, 6, 7, 596, 597
  - deploying, 612
  - inheritance hierarchy, 600, 601
  - security issues, 598, 599
  - signed, 598, 599
  - testing with Applet Viewer, 606, 607
  - testing with web browser, 612, 613
- Applet class (java.applet), 600, 601
- APPLET tag (HTML), 604, 605, 610, 611, 616, 617
- Applet Viewer, 606, 607
- Application, 6, 7
  - debugging, 84, 85
  - running from command prompt, 26, 27
  - running from TextPad, 22, 23
  - structuring with classes, 178, 179
  - testing, 84, 85
- APRIL field (Calendar class), 391
- ARCHIVE attribute (APPLET tag), 616, 617
- Argument, 52, 64, 65, 146, 147
  - passing to constructor, 196, 197
  - passing to method, 198, 199
- Argument list (of a method), 146, 147
- Arithmetic operators, 56, 57, 96, 97
- ArithmeticException (java.lang), 155
- Array
  - assigning values to, 324, 325
  - compared to a collection, 346, 347
  - copying, 336, 337
  - creating, 322, 323
  - creating references to, 336, 337
  - referring to, 324, 325
  - variable, 322, 323
- Array list, 348, 349, 352-355
  - untyped, 380, 381
- Array of arrays, 338, 339
- arraycopy method (System class), 336, 337
- ArrayIndexOutOfBoundsException (java.lang), 324, 325, 330, 331
- ArrayList class (java.util), 348, 349, 352-355
- Arrays class (java.util), 330, 331
- ASCII character set, 92, 93
- ASCII format, 20
- assert statement, 432, 433
- AssertionError, 432, 433
- Assertions, 432, 433
- Assigning values to array elements, 324, 325
- Assignment operator, 98, 99
- Assignment statement, 54-57
- Assignment statement, 96, 97

- Asynchronous thread, 456
- ATTLIST declaration (DTD), 686, 687
- Attribute (HTML), 604, 605
- Attribute (XML), 680, 681
  - vs. child element, 684, 685
- ATTRIBUTE constant (XMLStreamConstants interface), 698, 699
- AUGUST field (Calendar class), 391
- Autoboxing, 354, 355
- Autoflush feature, 634, 635
- Automatic driver loading, 730, 731
- AutoSelect class with FocusAdapter, 570, 571
- AutoSelect class, 566, 567
- available method (DataInputStream class), 656, 657
- AWT (Abstract Windows Toolkit), 62, 63, 226, 227, 478, 479

## B

---

- B tag (HTML), 304, 305
- Base class, 224, 225
- beforeFirst method (ResultSet interface), 736, 737
- BigDecimal class, 114, 115
- BIGINT data type (SQL), 754, 755
- Binary file, 628, 629, 650-659
  - reading, 656, 657
  - writing, 652, 653
- Binary input stream, 654, 655
- Binary operator, 96, 97
- Binary output stream, 650, 651
- Binary stream, 628, 629
- Binary strings, 658, 659
- binarySearch method (Arrays class), 330-333
- BIT data type (SQL), 754, 755
- Bit, 93
- BLOB objects (Binary Large Objects), 738, 739
- Block comment, 46, 47
- Block of code, 46, 47, 128, 129
- Block scope, 76, 77, 128, 129
- Blocked state (thread), 442, 443, 450
- BlueJ IDE, 38
- BODY tag (HTML), 604, 605
- Book class, 242, 244
- Boolean expression, 74, 75, 122-127
- boolean type, 92, 93
- Boolean value, 74, 75
- Boolean variable, 122, 123
- Border interface (javax.swing.border), 520
- Border layout manager, 496, 497, 532, 533
- Border, 510, 511, 520, 521
- BorderFactory class (javax.swing), 520, 521
- BorderLayout class (java.awt), 496, 497
- BOTH field (GridBagConstraints class), 536, 537
- Box layout manager, 532, 533
- break statement, 130, 131
  - with loops, 142, 143
- Browser, 6
  - displaying XML files, 688, 689

- Buffer, 630, 631
- Buffered input stream
  - binary, 654, 655
  - character, 638, 639
- Buffered output stream
  - character, 634, 635
  - binary, 650, 651
- Buffered stream, 630, 631
- BufferedInputStream class (java.io), 654, 655
- BufferedOutputStream class (java.io), 650, 651
- BufferedReader class (java.io), 638-641
- BufferedWriter class (java.io), 632, 633
- Business classes, 178, 179
- Business objects, 178, 179
- Business rules layer, 178, 179
- Button events (handling), 492, 493
- Button group, 518, 519
- Button, 476, 477
  - adding to panel, 490, 491
- ButtonGroup class (javax.swing), 518, 519
- byte type, 92, 93
- Byte, 93
- Bytecodes, 8, 9

## C

- C# (compared to Java), 4, 5
- C++ (compared to Java), 4, 5
- Calculated field, 722, 723
- Calendar class (java.util), 390, 391
- Call stack, 416, 417
- Calling
  - methods, 64, 65, 198, 199
  - static fields, 208, 209
  - static methods, 146, 147, 208, 209
- Camel notation, 54
- canRead method (File class), 624, 625
- canWrite method (File class), 624-627
- capacity method (StringBuilder class), 406-409
- Card layout manager, 532, 533
- Case label (switch statement), 130, 131
- Case structure, 130
- Case-sensitive, 20
- Casting, 56, 57, 102, 103
  - objects, 248, 249
- Catch an exception, 154
- Catch block, 156, 157
- catch clause (try statement), 418, 419
- Catching exceptions, 156, 157, 418, 419
- cause field (exception), 430, 431
- cd command (DOS), 30, 31
- CENTER field (BorderLayout class), 496, 497
- CENTER field (FlowLayout class), 494, 495
- CENTER field (GridBagConstraints class), 536, 537
- Centering frames, 486, 487
- char type, 92, 93
- Character input stream, 638, 639
- Character output stream, 634, 635
- Character stream, 628, 629
- CHARACTERS constant (XMLStreamConstants interface), 698, 699
- charAt method (String class), 402-405
- charAt method (StringBuilder class), 406-409
- Check box, 510, 511, 516, 517
- Checked exception, 414, 415
  - throwing to calling method, 422, 423
- Child class, 224, 225
- Child element (XML), 682, 683, 684
  - vs. attributes, 684, 685
- Class, 8, 9, 50, 51
  - declaring, 196, 197
  - event listener, 556-559
  - for working with random-access files, 666-673
  - for working with text files, 644-649
  - instance of, 64, 65
  - modifying for applet, 602, 603
- Class class (java.lang), 246, 247, 736, 737
- Class declaration, 50, 51
- Class diagram, 180, 181
  - Line Item application, 212, 213
  - Product Maintenance application, 278, 279
- Class documentation, 302, 303
- Class field, 206, 207
- Class method, 206, 207
- Class name, 20, 21
- Class path, 16, 17, 204
- ClassCastException (java.lang), 248, 249
- Classes
  - coding in same file, 308, 309
  - importing, 62, 63
  - legacy collections, 378, 379
  - making available, 300, 301
  - nested, 310, 311
  - storing in packages, 296, 297
  - structuring, 178, 179
- ClassNotFoundException, 730, 731
- classpath command (DOS), 16, 17
- clear method (LinkedList class), 362, 363
- clear method (ArrayList class), 352, 353
- clear method (DefaultListModel class), 530, 531
- clear method (HashMap class), 374, 375
- clear method (TreeMap class), 374, 375
- CLOB objects (Character Large Objects), 738, 739
- clone method (Object class), 228, 229
- clone method (Object class), 288-291
- Cloneable interface (java.lang), 266-269
  - implementing, 288-291
- CloneNotSupportedException (java.lang), 288, 289
- Cloning an object, 288, 289
- close method (BufferedReader class), 640, 641
- close method (DataInputStream class), 656, 657
- close method (DataOutputStream class), 652, 653
- close method (PrintWriter class), 636, 637
- close method (RandomAccessFile class), 662, 663

- close method (ResultSet interface), 736, 737
- close method (XMLStreamWriter class), 694, 695
- Closeable interface (java.io), 632, 633
- Closing frames, 484, 485
- Cloudscape, 760, 761
- CODE attribute (APPLET tag), 604, 605
- Code tag (HTML), 304, 305
- Collection framework, 348, 349
- Collection interface (java.util), 348, 349
- Collections, 346, 347, 348, 349
  - legacy, 378, 379
  - untyped, 380, 381
  - compared to arrays, 346, 347
- Column
  - database, 714, 715
  - file, 636, 637
- Combo box, 510, 511, 522, 523
  - event listener, 524, 525
- Command path, 14, 15
- Command prompt, 26, 27
- Command Results window (TextPad), 22, 23
- COMMENT constant (XMLStreamConstants interface), 698, 699
- Comment (HTML), 604, 605
- Comment (Java), 46, 47
- Comment (XML), 682, 683
- Communicating among threads, 458, 459
- Comparable interface (java.lang), 266, 267, 330, 331, 334, 335
- compareTo method (BigDecimal class), 114, 115
- compareTo method (Comparable interface), 334, 335
- compareTo method (String class), 402, 403
- compareToIgnoreCase method (String class), 402-405
- Comparing objects, 250, 251
- Comparing primitive data types, 122, 123
- Comparing strings, 124, 125, 402, 403
- Comparing variables, 74, 75
- Compiler (Java), 8, 9
- Compile-time error, 24, 25
- Compiling source code
  - classes in a package, 298, 299
  - from the command prompt, 26, 27
  - from TextPad, 22, 23
- Component class (java.awt), 226, 227, 478-481
- Component hierarchy, 478, 479
- Components, 476, 478
- Concatenating strings, 58, 59
- CONCUR\_READ\_ONLY field (ResultSet interface), 734, 735
- CONCUR\_UPDATABLE field (ResultSet interface), 734, 735
- Concurrency, 456, 457
- Conditional expression, 74, 75
- Connecting to a database, 730, 731
  - embedded Derby database, 772, 773
  - networked Derby database, 784, 785
- Connection object, 730, 731, 772, 773
  - closing, 750
- Console applications, 475
- Console, 22, 23
  - input and output, 68-73
- Console applications
  - Invoice application with array list, 356, 357
  - Invoice application with linked list, 368, 369
  - Line Item application, 212, 213
  - Order Queue application, 460, 461
  - Product application with inheritance, 238, 239
  - Product application, 204, 205
  - Product Maintenance application, 280, 281
- Constant, 94, 95
  - in an enumeration, 312, 313
  - in an interface, 262, 268-271
- Constraints (Grid Bag layout), 534-537
- Constructor, 64, 65
  - coding, 188, 189
- consume method (InputEvent class), 568, 569
- Consumers, 460, 461
- Container, 478
- Container class (java.awt), 226, 227, 478, 479, 495
- Containing class, 558, 559
- contains method (ArrayList class), 352, 353
- contains method (DefaultListModel class), 530, 531
- contains method (LinkedList class), 362, 363
- containsKey method (HashMap class), 374, 375
- containsKey method (TreeMap class), 374, 375
- containsValue (TreeMap class), 374, 375
- containsValue method (HashMap class), 374, 375
- Content pane, 488, 489
- Content element (XML), 682, 683
- continue statement, 144, 145
- Control statements, 74-79
- Converting HTML pages for applets, 608-611
- Copying arrays, 336, 337
- copyOf method (Arrays class), 330, 331, 336, 337
- copyOfRange method (Arrays class), 330, 331, 336, 337
- Counter variable (while loop), 78, 134
- CPU (central processing unit), 438, 439
- createEtchedBorder method (BorderFactory class), 520, 521
- createLineBorder method (BorderFactory class), 520, 521
- createLoweredBevelBorder method (BorderFactory class), 520, 521
- createNewFile method (File class), 624-627
- createRaisedBevelBorder method (BorderFactory class), 520, 521
- createStatement method (Connection interface), 734, 735
- createTitledBorder method (BorderFactory class), 520, 521
- createXMLStreamReader method (XMLInputFactory class), 696, 697
- createXMLStreamWriter method (XMLOutputFactory class), 692, 693
- Current row pointer, 720, 721
- currentThread method (Thread class), 445, 448, 449
- Cursor
  - random-access file, 660
  - result set, 720, 721, 736, 737
- Custom exception classes, 428-431

## D

---

Daemon thread, 444, 445  
 DAOFactory class, 278, 279, 282, 283  
 Data hiding, 180  
 Data types  
     Java mapped to SQL, 754, 755  
     primitive, 54, 55  
     SQL mapped to Java, 754, 755  
 Data validation, 160-165, 572-579  
 Database, 714, 715  
     connecting to, 730, 731  
     modifying data, 740, 741  
 Database drivers, 726, 727  
 Database layer, 178, 179  
 DataInput interface (java.io), 656, 657, 662, 663  
 DataInputStream class (java.io), 654-657  
 DataOutput interface (java.io), 652, 653, 662, 663  
 DataOutputStream class (java.io), 650-653  
 Date class (java.util), 392, 393  
 DATE data type (SQL), 754, 755  
 DATE field (Calendar class), 391  
 Date object, 390-395  
 DateFormat class (java.text), 394, 395  
 DateUtils class, 396, 397  
 DAY\_OF\_MONTH field (Calendar class), 391  
 DAY\_OF\_WEEK field (Calendar class), 391  
 DAY\_OF\_YEAR field (Calendar class), 391  
 DB2, 714, 715  
 DBMS (database management system), 714, 715  
 DBStringUtil class, 744, 745  
 DDL (Data Definition Language), 720  
 Debugging applications, 84, 85  
 Debugging statements, 112, 113  
 DECEMBER field (Calendar class), 391  
 Declaration (XML), 682, 683  
 Declaring  
     arrays, 322, 323  
     classes, 50, 51, 196, 197  
     enumerations, 312, 313  
     interfaces, 268, 269  
     main methods, 52, 53  
     static methods, 146, 147  
     typed collections, 350, 351  
     variables, 54, 55  
 Decrement operator, 96, 97  
 Default constructor, 188-190  
 Default label (for a switch statement), 130, 131  
 Default value (of a field), 718, 719  
 DefaultListModel class (javax.swing), 530, 531  
 delete method (File class), 624, 625  
 delete method (StringBuilder class), 406, 407  
 DELETE statement (SQL), 724, 725  
 deleteCharAt method (StringBuilder class), 406-409  
 deleteRow method (ResultSet interface), 740, 741  
 Deleting records from a database, 740, 741, 724, 725  
 Delimited text file, 636, 637

DepartmentConstants interface, 268, 269  
 Deploying applets, 612  
 Deprecated features of Java, 28, 29  
 deprecation switch (javac command), 28, 29  
 Derby database, 760-787  
     configuring your system for, 762, 763  
     connecting to embedded db, 772, 773  
     connecting to networked db, 784, 785  
     documentation, 786, 787  
     interactive JDBC tool, 764-771  
     starting networked server, 782, 783  
 Derived class, 224, 225  
 destroy method (Applet class), 600, 601  
 Diagram (class), 180, 181  
 Dialog box (displaying), 572, 573  
 Dimension class (java.awt), 486, 487  
 dir command (DOS), 30, 31  
 Directory structure for classes in a package, 296, 297  
 Disconnecting from a Derby database, 774, 775  
 DISPOSE\_ON\_CLOSE constant (WindowConstants interface), 484, 485  
 divide method (BigDecimal class), 114, 115  
 Division assignment operator, 99  
 Division operator, 96, 97  
 DML (Data Manipulation Language), 720  
 DO\_NOTHING\_ON\_CLOSE constant (WindowConstants interface), 484, 485  
 DOCTYPE declaration (XML document), 686, 687  
 Document Selector pane (TextPad), 22, 23  
 Documentation  
     installing, 32, 33  
     Derby database, 786, 787  
     class, 302, 303  
     Java API, 66, 67  
     generating for a class, 306, 307  
     navigating, 34, 35  
     viewing for a class, 306, 307  
 DOM (Document Object Model), 690, 691  
 DOS commands, 30, 31  
 DOS prompt, 26, 27  
 DOS window, 26  
 DOSKey program, 30, 31  
 Double class, 108, 109  
 DOUBLE data type (SQL), 754, 755  
 double data type, 54, 55, 92, 93  
 Double-precision number, 93  
 Do-while loop, 134, 135  
 DriverManager class (java.sql), 730, 731, 772, 773  
 Drivers (database), 726, 727  
 DTD (Document Type Definition), 686, 687  
 DTD constant (XMLStreamConstants interface), 698, 699

## E

---

EAST field (BorderLayout class), 496, 497  
 EAST field (GridBagConstraints class), 536, 537  
 Eclipse IDE, 36, 37

- Editing XML files, 688, 689
- EE (Enterprise Edition), 4, 5
- ELEMENT declaration (DTD), 686, 687
- Elements
  - XML, 680-683
  - array, 322-325
- EMBED tag (HTML), 604, 605, 610, 611
- Embedded Derby database, 772-781
- Empty string, 58, 59
- Encapsulation, 180, 181
  - Product class, 186
- End tag (XML), 682, 683
- END\_ELEMENT constant (XMLStreamConstants interface), 698-701
- End-of-line comment, 46, 47
- endsWith method (String class), 402, 403
- Enhanced for loop
  - arrays, 328, 329
  - linked list, 364, 365
  - map, 376, 377
- Enhanced Invoice application, 132, 133
- Entry (validating), 162-165
- entrySet method (HashMap class), 374-377
- entrySet method (TreeMap class), 374-377
- Enum class (java.lang), 314, 315
- enum keyword, 312, 313
- Enumeration, 312-315
- EOFException (java.io), 632, 633
- Equality operator, 75, 122-125
- equals method
  - Arrays class, 330-333
  - Object class, 228, 229, 250, 251
  - String class, 74, 75, 124, 125, 402, 403
  - overriding, 250, 251
- equalsIgnoreCase method (String class), 74, 75, 124, 125, 402, 403
- Equi-join, 722, 723
- Error, 24, 25
- Error class (java.lang), 414, 415
- Error message (displaying), 572, 573
- ERROR\_MESSAGE field (JOptionPane class), 572, 573
- Escape sequences, 60, 61
- Event, 548, 549
- Event listener, 492, 548, 549
  - as anonymous inner class, 562, 563
  - as inner class, 558, 559
  - as separate class, 556, 557
  - combo box, 524, 525
  - for single event, 560, 561
  - in panel class, 554, 555
- Event listener interface, 548, 549
  - implementing, 552, 553
- Event object, 548-551
- Event source, 548, 549
  - ways to handle, 552, 553
- EventListener interface (java.util), 266, 267
- EventObject class (java.util), 548, 549

- Events,
  - button, 492, 493
  - handling, 548-563
  - types, 550, 551
- Exception chaining, 430, 431
- Exception class (java.lang), 154, 155, 414, 415
- Exception classes (custom), 428-431
- Exception handler, 156, 416, 417
  - testing, 424, 425
- Exception handling, 154, 414, 415
- Exception hierarchy, 154, 155
- Exception propagation, 416, 417
- Exceptions, 70, 71, 154, 155, 414, 415
  - catching, 156, 157, 418, 419
  - custom, 428-431
  - swallowing, 418, 419
  - throwing, 416, 417, 422, 423
  - when to throw, 424, 425
- executeQuery method (PreparedStatement interface), 742, 743
- executeQuery method (Statement interface), 734, 735
- executeUpdate method (PreparedStatement interface), 742, 743
- executeUpdate method (Statement interface), 740, 741
- exists method (File class), 624-627
- EXIT\_ON\_CLOSE field (JFrame class), 484, 485
- Explicit cast, 102, 103
- Extending a superclass, 224, 225, 234, 235
- Extending the JFrame class, 482, 483
- Extending the Thread class, 446, 447
- extends keyword, 234, 235, 272, 273

## F

---

- Factory pattern, 278
- FEBRUARY field (Calendar class), 390, 391
- Field
  - class 180, 181
  - database, 714, 715, 718, 719
  - file, 636, 637
- File class (java.io), 624, 625
- File I/O, 628, 629
- File name, 20, 21
- FileInputStream class (java.io), 654, 655
- FileNotFoundException (java.io), 632, 633
- FileOutputStream class (java.io), 650, 651
- FileReader class (java.io), 638, 639
- FileWriter class (java.io), 632, 633
- fill field (GridBagConstraints class), 536, 537
- fill method (Arrays class), 330-333
- FilterInputStream class (java.io), 654, 655
- FilterOutputStream class (java.io), 650, 651
- Final class, 254, 255
- final keyword, 94, 95, 254, 255, 268, 269
- Final method, 254, 255
- Final parameter, 254, 255
- Final variable, 94, 95
- finalize method (Object class), 228, 229
- finally clause (try statement), 418-421

first method (ResultSet interface), 736, 737  
 Fixed-length strings, 664, 665  
 float type, 92, 93  
 Floating-point number, 92, 93  
 Flow layout manager, 494, 495, 532, 533  
 FlowLayout class (java.awt), 494, 495  
 flush method (DataOutputStream class), 652, 653  
 flush method (PrintWriter class), 636, 637  
 flush method (XMLStreamWriter class), 694, 695  
 flushing the buffer, 630, 631  
 Focus, 476  
 Focus event, 566, 567  
 Focus listener, 566, 567  
 FocusAdapter class (java.awt.event), 570, 571  
 FocusEvent class (java.awt.event), 566, 567  
 focusGained method (FocusListener interface), 566, 567  
 FocusListener interface (java.awt.event), 566, 567  
 focusLost method (FocusListener interface), 566, 567  
 For loop, 136, 137  
     nested, 140, 141  
     using with arrays, 326, 327  
 for statement, 136, 137  
 Foreach loop, 328, 329  
 Foreign key, 716, 717  
 format method (DateFormat class), 394, 395  
 format method (NumberFormat object), 104, 105  
 Formatted Invoice application, 110, 111  
 Formatting dates and times, 394, 395  
 forName method (Class class), 730, 731  
 Forward-only result set, 736, 737  
 Forward-only, read-only result set, 734, 735  
 Frame, 476, 477  
     closing, 484, 485  
     centering, 486, 487  
     displaying, 482, 483  
 Frame class (java.awt), 478, 479, 482, 483  
 FRIDAY field (Calendar class), 391  
 FROM clause (SELECT statement), 720, 721  
 FULL field (DateFormat class), 394, 395  
 Future Value application, 138, 139  
     data validation version, 166-171  
     exception handling version, 158, 159  
     nested for loops version, 138, 139  
     static method version, 148, 149  
 Future Value Calculator applet  
     code, 602, 603  
     HTML page, 604, 605  
     interface, 596, 597  
 Future Value Calculator application (GUI), 502-505

## G

Garbage collector, 228, 229  
 Generic class, 350, 351  
 Generic methods for validating an entry, 164, 165  
 Generic queue, 366, 367  
 GenericQueue class, 366, 367

Generics, 350, 351  
 get method (ArrayList class), 352-355  
 get method (Calendar class), 390, 391  
 get method (DefaultListModel class), 530, 531  
 get method (HashMap class), 374, 375  
 get method (LinkedList class), 362, 363  
 get method (TreeMap class), 374, 375  
 getAbsolutePath method (File class), 624-627  
 getAttributeCount method (XMLStreamReader class), 698-701  
 getAttributeLocalName method (XMLStreamReader class), 698-701  
 getAttributeValue method (XMLStreamReader class), 698-701  
 getCause method (Throwable class), 430, 431  
 getClass method (Object class), 228, 229, 246, 247  
 getColumnCount method (ResultSetMetaData interface), 752, 753  
 getColumnLabel method (ResultSetMetaData interface), 752, 753  
 getColumnName method (ResultSetMetaData interface), 752, 753  
 getColumnType method (ResultSetMetaData interface), 752-755  
 getColumnName method (ResultSetMetaData interface), 752-755  
 getComponent method (FocusEvent class), 566, 567  
 getConnection method (DriverManager class), 730, 731, 772, 773  
 getContentPane method (JFrame class), 488, 489  
 getCurrencyInstance method (NumberFormat class), 104, 105  
 getDateInstance method (DateFormat class), 394, 395  
 getDateTimeInstance method (DateFormat class), 394, 395  
 getDefaultToolkit method (Toolkit class), 486, 487  
 getDouble method (ResultSet interface), 738, 739  
 getElementText method (XMLStreamReader class), 698-701  
 getEventType method (XMLStreamReader class), 698-701  
 getFirst method (LinkedList class), 362, 363  
 getHeight method (Component class), 480, 481  
 getItem method (ItemEvent class), 524, 525  
 getItemCount method (JComboBox class), 522, 523  
 getKey method (Map.Entry interface), 374-377  
 getKeyChar method (KeyEvent class), 568, 569  
 getKeyCode method (KeyEvent class), 568, 569  
 getLast method (LinkedList class), 362, 363  
 getLocalName method (XMLStreamReader class), 698-701  
 getMessage method (Throwable class), 426, 427  
 getMetaData method (ResultSet interface), 752, 753  
 getName method (Class class), 246, 247  
 getName method (Component class), 480, 481  
 getName method (File class), 624-627  
 getName method (Thread class), 445  
 getNumberInstance method (NumberFormat class), 104, 105  
 getOppositeComponent method (FocusEvent class), 566, 567  
 getPath method (File class), 624, 625  
 getPercentInstance method (NumberFormat class), 104, 105  
 getRow method (ResultSet interface), 736, 737  
 getScreenSize method (Toolkit class), 486, 487  
 getSelectedIndex method (JComboBox class), 522, 523  
 getSelectedIndex method (JList class), 526, 527

getSelectedIndices method (JList class), 528, 529  
 getSelectedItem method (JComboBox class), 522, 523  
 getSelectedValue method (JList class), 526, 527  
 getSelectedValues method (JList class), 528, 529  
 getSource method (ActionEvent class), 492, 493  
 getSource method (ItemEvent class), 524, 525  
 getStateChanged method (ItemEvent class), 524, 525  
 getString method (ResultSet interface), 738, 739  
 getText (JButton class), 490, 491  
 getText method (JLabel class), 498, 499  
 getText method (JTextArea class), 512, 513  
 getText method (JTextField class), 500, 501  
 getText method (XMLStreamReader class), 698, 699  
 getTime method (Calendar class), 390, 391, 392, 393  
 getTime method (Date class), 392, 393  
 getTimeInstance method (DateFormat class), 394, 395  
 getValue method (Map.Entry interface), 374-377  
 getWidth method (Component class), 480, 481  
 getX method (Component class), 480, 481  
 getY method (Component class), 480, 481  
 Glass pane, 488, 489  
 Greater Than operator, 75, 122, 123  
 Greater Than Or Equal operator, 75, 122, 123  
 GregorianCalendar class (java.util), 388, 389  
 Grid Bag layout manager, 532, 533, 534-537  
 Grid layout manager, 532, 533  
 GridBagConstraints class (java.awt), 534-537  
 GridBagLayout class (java.awt), 534, 535  
 gridheight field (GridBagConstraints class), 536, 537  
 gridwidth field (GridBagConstraints class), 536, 537  
 gridx field (GridBagConstraints class), 536, 537  
 gridy field (GridBagConstraints class), 536, 537  
 GUI (graphical user interface), 6, 7, 475

## H

H1 tag (HTML), 604, 605  
 HALF\_EVEN field (RoundingMode enumeration), 114, 115  
 HALF\_UP field (RoundingMode enumeration), 114, 115  
 Half-even rounding technique, 105, 106  
 has map, 374, 375  
 Hash code, 228, 229  
 Hash table, 374  
 hashCode method (Object class), 228, 229  
 HashMap class (java.util), 348, 349, 374, 375  
 HashSet class (java.util), 348, 349  
 HashTable class (java.util), 378, 379  
 hasNext method (Scanner class), 160, 161  
 hasNext method (XMLStreamReader class), 698-701  
 hasNextDouble method (Scanner class), 160, 161  
 hasNextInt method (Scanner class), 160, 161  
 HEAD tag (HTML), 604, 605  
 Heavyweight components, 478  
 HEIGHT attribute (APPLET tag), 604, 605  
 height field (Dimension class), 486, 487  
 HIDE\_ON\_CLOSE constant (WindowConstants interface), 484, 485

HORIZONTAL field (GridBagConstraints class), 536, 537  
 HORIZONTAL\_SCROLLBAR\_ALWAYS field  
   (ScrollPaneConstants interface), 515  
 HORIZONTAL\_SCROLLBAR\_AS\_NEEDED field  
   (ScrollPaneConstants interface), 515  
 HORIZONTAL\_SCROLLBAR\_NEVER field  
   (ScrollPaneConstants interface), 515  
 HOUR field (Calendar class), 391  
 HOUR\_OF\_DAY field (Calendar class), 391  
 HTML (Hypertext Markup Language), 604, 605  
 HTML Converter, 608, 609  
 HTML documentation, 306, 307  
 HTML page, 6, 604, 605  
   converted, 610, 611  
   converting for an applet, 608, 609  
 HTML tags, 604, 605  
   in javadoc comments, 304, 305

## I

I tag (HTML), 304, 305  
 I/O exceptions, 632, 633  
 I/O operations, 628, 629  
 IBM Cloudscape database, 760, 761  
 IDE (Integrated Development Environment), 36-39  
 Identifier, 48, 49  
 Identity of an object, 182, 183  
 if/else statement, 76, 77, 128, 129  
 ij tool (Derby database), 764-771  
 IllegalArgumentException (java.lang), 154, 155, 424, 425  
 IllegalMonitorStateException (java.lang), 459  
 Immutable object, 290  
 Immutable strings, 406  
 Implementing interfaces, 262, 263, 270, 271  
 Implementing the Runnable interface, 448, 449  
 implements keyword, 270-273  
 Implicit cast, 102, 103  
 import statement, 62, 63, 296-301  
 Import (static), 314, 315  
 Increment operator, 96, 97  
 Index of an array, 324, 325  
 index.html file, 306, 307  
 indexOf method (ArrayList class), 352, 353  
 indexOf method (LinkedList class), 362, 363  
 indexOf method (String class), 402-405  
 IndexOutOfBoundsException (java.lang), 401  
 Inequality operator, 75, 122-125  
 Infinite loop, 78, 79, 134, 135  
 INFORMATION\_MESSAGE field (JOptionPane class), 572, 573  
 Inheritance, 224, 225  
   Java API, 226, 227  
   using in applications, 230, 231  
   using with interfaces, 276, 277  
 Inheritance hierarchy, 226, 227  
   applets, 600, 601  
   Swing components, 478, 479

- inherited keyword, 276, 277
- init method (Applet class), 600, 601, 602, 603
- initCause method (Throwable class), 430, 431
- Initialization block, 210, 211
- Initializing constants, 94, 95
- Initializing variables, 54, 55, 94, 95
- Inner class, 310, 311
  - anonymous, 562, 563
  - event listener, 558, 559
- Inner join, 722, 723
- Input data (validating), 572-579
- Input file, 628, 629
- Input stream, 628, 629
  - binary, 654, 655
  - character, 638, 639
- InputEvent class (java.awt.event), 568, 569
- InputMismatchException (java.util), 154, 155, 156
  - preventing, 160, 161
- InputStream class (java.io), 654, 655
- InputStream hierarchy, 654, 655
- InputStreamReader class (java.io), 638, 639
- insert method (StringBuilder class), 406-409
- INSERT statement (SQL), 724, 725
- insertRow method (ResultSet interface), 740, 741
- Insets class (java.awt), 536, 537
- insets field (GridBagConstraints class), 536, 537
- Installing
  - Java, 10, 11
  - TextPad, 18, 19
- Instance of a class, 64, 65, 182, 183
- Instance variable
  - coding, 186, 187
  - initializing, 196, 197
- instanceof keyword, 247, 248
- Instantiating arrays, 322, 323
- Instantiation, 64, 65
- int data type, 54, 55, 92, 93
- Integer class, 108, 109
- INTEGER data type (SQL), 754, 755
- Integer, 92, 93
- Interactive JDBC tool (Derby database), 764-771
- Interface, 262, 263
  - advantages, 264, 265
  - coding, 268, 269
  - compared to abstract class, 264, 265
  - implementing, 270, 271
  - inheriting, 276, 277
  - using as parameter, 274, 275
- interface keyword, 268, 269
- Interface type, 262-264, 274, 275
- Interfaces,
  - Java API, 266, 267
  - Product Maintenance application, 278, 279
- Internet Explorer, 596
- Interpreter, 8, 9
- interrupt method (Thread class), 445, 454, 455
- InterruptedException (java.lang), 444, 445, 450, 451, 454, 455
- Interrupting threads, 454, 455
- Invoice application, 80, 81
  - array list version, 356-361
  - BigDecimal version, 114, 115
  - date version, 398, 399
  - enhanced version, 132, 133
  - linked list version, 368-373
- Invoice class, 356-359
- InvoiceApp class, 360, 361
  - linked list version, 370-373
- IOException (java.io), 632, 633
- IOException hierarchy, 632, 633
- IOStringUtils class, 664, 665
- ipadx field (GridBagConstraints class), 536, 537
- ipady field (GridBagConstraints class), 536, 537
- isAfterLast method (ResultSet interface), 736, 737
- isAltDown method (KeyEvent class), 568, 569
- isBeforeFirst method (ResultSet interface), 736, 737
- isControlDown method (KeyEvent class), 568, 569
- isDirectory method (File class), 624-627
- isEmpty method (ArrayList class), 352, 353
- isEmpty method (String class), 402-405
- isEnabled method (Component class), 480, 481
- isFile method (File class), 624, 625
- isFirst method (ResultSet interface), 736, 737
- isInterrupted method (Thread class), 445, 454, 455
- isLast method (ResultSet interface), 736, 737
- isSelected method (JCheckBox class), 516, 517
- isSelected method (JRadioButton class), 518, 519
- isSelectedIndex method (JList class), 526, 527
- isShiftDown method (KeyEvent class), 568, 569
- isTemporary method (FocusEvent class), 566, 567
- isVisible method (Component class), 480, 481
- Item event (combo box), 524
- ItemEvent class (java.awt.event), 524, 525
- itemStateChanged method (ItemListener interface), 524, 525
- Iteration structure, 78

## J

---

- J2EE (Java 2 Platform, Enterprise Edition), 4
- J2SE (Java 2 Platform, Standard Edition), 4
- Jagged arrays, 340, 341
- JANUARY field (Calendar class), 390, 391
- JApplet class (javax.swing), 600, 601
- jar command, 300, 301
- JAR file, 300, 301
  - including in HTML page, 616, 617
  - using with applets, 614-617
- JAR tool, 614, 615
- Java API
  - inheritance, 226, 227
  - interfaces, 266, 267
- Java classes (importing), 62, 63
- java command, 26, 27
- Java compiler, 8, 9
- Java Console, 612, 613



Java data types, 754, 755  
 Java DB, 760, 761  
 Java EE (Enterprise Edition), 4, 5  
 Java interpreter, 8, 9  
 Java Plug-in, 8, 9, 596, 597  
 Java SE (Standard Edition), 4, 5  
 java.awt package, 226, 227, 478, 479  
 java.awt.event package, 550, 551  
 java.io package, 630, 631  
 java.lang package, 62, 63, 226, 227, 246, 247  
 java.lang.Math class, 106, 107  
 java.math package, 114, 115  
 java.math.BigDecimal class, 114, 115  
 java.nio package, 630, 631  
 java.sql package, 730, 731  
 java.text.NumberFormat class, 104, 105  
 java.util package, 71  
 java.util.Map class, 374, 375  
 java.util.zip package, 651, 654, 655  
 JavaBean, 186  
 javac command, 26, 27, 298, 299  
     switches, 28, 29  
 javadoc command, 306, 307  
 javadoc comment, 302, 303  
 javadoc tags, 304, 305  
 javadoc utility, 302-307  
 javax.swing package, 226, 227, 478, 479  
 javax.swing.border package, 520, 521  
 javax.swing.event package, 550, 551  
 JButton class (javax.swing), 478, 479  
 JCheckBox class, 516, 517  
 JComboBox class, 522, 523  
 JComponent class (javax.swing), 478, 479, 521  
 JDataConnect driver, 732, 733  
 JDBC (Java Database Connectivity), 726  
 JDBC driver manager, 726, 727  
 JDBC-ODBC bridge driver, 726, 727, 730, 731  
 JDK (Java Development Kit), 4, 5  
     directories and files, 12, 13  
     installing, 10, 11  
 JFrame class (javax.swing), 224-227, 478, 479  
 JLabel class (javax.swing), 478, 479, 498, 499  
 JList class, 526-529  
 JNetDirect Inc., 732, 733  
 Join, 722, 723  
 Joining strings, 58, 59  
 JOptionPane class (javax.swing), 572, 573  
 JPanel class (javax.swing), 478, 479  
 JRadioButton class, 518, 519  
 JRE (Java Runtime Environment), 12, 13  
 JScrollPane class (javax.swing), 514, 515  
 JSP (Java Server Page), 6  
 JTabbedPane class (javax.swing), 532, 533  
 JTextArea class (javax.swing), 512, 513  
 JTextComponent class, 512, 566, 567  
 JTextField class (javax.swing), 478, 479, 500, 501  
 JULY field (Calendar class), 391

JUNE field (Calendar class), 391  
 JVM (Java virtual machine), 8, 9

## K

KeyAdapter class (java.awt.event), 570, 571  
 Keyboard event, 568, 569  
 KeyEvent class (java.awt.event), 568, 569  
 KeyListener interface (java.awt.event), 568, 569  
 keyPressed method (KeyListener interface), 568, 569  
 keyReleased method (KeyListener interface), 568, 569  
 keyTyped method (KeyListener interface), 568, 569  
 Key-value pair, 348, 349  
 Keywords, 48, 49

## L

Label, 476, 477, 498, 499  
 Labeled break statement, 142, 143  
 Labeled continue statement, 144, 145  
 last method (ResultSet interface), 736, 737  
 lastIndexOf method (String class), 402, 403  
 lastModified method (File class), 624, 625  
 Late binding, 236  
 Layered pane, 488, 489  
 Layering input streams  
     binary, 654, 655  
     character, 638, 639  
 layering output streams  
     binary, 650, 651  
     character, 635, 636  
 layering streams, 630, 631  
 Layout managers, 480, 494-501, 531-537  
 LEFT field (FlowLayout class), 494, 495  
 Left outer join, 722  
 Legacy collection classes, 378, 379  
 length field (array), 326, 327  
 length method (File class), 624, 625  
 length method (RandomAccessFile class), 662, 663  
 length method (String class), 402-405  
 length method (StringBuilder class), 406-409  
 Length of an array, 322, 323  
 Less Than operator, 75, 122, 123  
 Less Than Or Equal operator, 75, 122, 123  
 Life cycle of a thread, 442, 443  
 Lightweight components, 478  
 Line Item application, 212-219  
 Line numbers (displaying in TextPad), 20, 21  
 LineItem class with Cloneable interface, 290, 291  
 LineItem class, 218, 219  
 LineItemApp class, 214, 215  
 Linked list, 348, 349, 362-367  
     using to create a queue, 366, 367  
 LinkedList class (java.util), 348, 349, 362-367  
 List, 348, 349, 510, 511, 526-531  
 List interface (java.util), 348, 349

- list method (File class), 624-627
- List model, 530, 531
- Listener interfaces, 550, 551
  - low-level, 564, 565
- listFiles method (File class), 624, 625
- ListModel interface (javax.swing), 530, 531
- listRoots method (File class), 624, 625
- ListSelectionModel interface (javax.swing), 528, 529
- Literal value, 54, 55
- Literal, 96
- Local class, 310, 311
- Locking objects, 456, 457
- Logical error, 84, 85
- Logical operators, 126, 127
- LONG field (DateFormat class), 394, 395
- long data type, 92, 93
- LONGVARBINARY data type (SQL), 754, 755
- LONGVARCHAR data type (SQL), 754, 755
- Loops, 134-141
- Low-level events, 550, 551, 564-571

## M

---

- Main method, 52, 53
- Main thread, 438, 439
- Manipulating strings, 402, 403
- Many-to-many relationship, 716, 717
- Map, 348, 349, 374, 375
- Map interface (java.util), 348, 349
- Map.Entry interface (java.util.Map), 374, 375
- MARCH field (Calendar class), 391
- Math class, 106, 107
- max method (Math class), 106, 107
- MAX\_PRIORITY field (Thread class), 452, 453
- MAY field (Calendar class), 391
- MEDIUM field (DateFormat class), 395
- Member class, 310, 311
- Metadata
  - file, 660, 661
  - result set, 752-755
- Metal look and feel, 476
- Method, 180, 181
- Method, 52, 53
  - calling, 64, 65, 198, 199
  - coding, 190, 191
  - naming, 190, 191
  - overloading, 146, 192, 193
  - overriding, 234, 235
  - signature, 146, 147, 192, 193
  - static, 64, 65, 146, 147
- Methods
  - for handling dates, 396, 397
  - in an interface, 262
- Middle layer, 178, 179
- min method (Math class), 106, 107
- MIN\_PRIORITY field (Thread class), 452, 453
- MINUTE field (Calendar class), 391

- mkdirs method (File class), 624-627
- Modulus assignment operator, 99
- Modulus operator, 96, 97
- MONDAY field (Calendar class), 391
- MONTH field (Calendar class), 391
- moveToCurrentRow method (ResultSet interface), 740, 741
- moveToInsertRow method (ResultSet interface), 740, 741
- Multi-layered architecture, 178
- Multiple inheritance, 262
- MULTIPLE\_INTERVAL\_SELECTION field (ListSelectionModel interface), 528, 529
- Multiplication assignment operator, 99
- Multiplication operator, 96, 97
- multiply method (BigDecimal class), 114, 115
- Multithreading, 438, 439
- Multi-tiered architecture, 178
- MurachDB class, 776-781
- Mutable object, 290, 291
- Mutable strings, 406, 407

## N

---

- name method (enumeration constant), 314, 315
- Naming
  - conventions, 94, 95
  - recommendations, 50, 51, 54, 55
  - rules, 50, 51
- Narrowing conversion, 102, 103
- Native protocol all Java driver, 726, 727
- Native protocol partly Java driver, 726, 727
- Negative sign operator, 96, 97
- Nested classes, 310, 311
- Nested exceptions, 730, 731
- Nested for loops, 140, 141
- Nested foreach loops, 340
- Nesting if statements, 76, 77, 128, 129
- Net protocol all Java driver, 726, 727
- NetBeans IDE, 38, 39
- Netscape, 596
- Networked Derby database, 782-785
- NetworkServerControl class, 782, 783
- new keyword, 196, 197, 322, 323
- New state (thread), 442, 443
- newInstance method (XMLInputFactory class), 696, 697
- newInstance method (XMLOutputFactory class), 692, 693
- next method (ResultSet interface), 736, 737
- next method (Scanner class), 70, 71
- next method (XMLStreamReader class), 698-701
- nextDouble method (Scanner class), 70, 71
- nextInt method (Scanner class), 70, 71
- nextLine method (Scanner class), 70, 71, 160, 161
- NONE field (GridBagConstraints class), 536, 537
- NORM\_PRIORITY field (Thread class), 452, 453
- NORTH field (BorderLayout class), 496, 497
- NORTH field (GridBagConstraints class), 536, 537
- NORTHEAST field (GridBagConstraints class), 536, 537
- NoSuchElementException class (java.util), 154, 155

Not operator, 126, 127  
 notify method (Object class), 440-443, 458, 459  
 notifyAll method (Object class), 440-443, 458, 459  
 NOVEMBER field (Calendar class), 391  
 null keyword, 124, 125  
 Null value, 58, 59  
 NullPointerException (java.lang), 155  
   preventing, 160, 161  
 NumberFormat class, 104, 105  
 NumberFormatException (java.lang), 154, 155, 574, 575  
 NUMERIC data type (SQL), 754, 755  
 Numeric data, 92-103  
 Numeric variables, 54, 57  
   comparing, 74, 75  
 NumFilter class with KeyAdapter, 570, 571  
 NumFilter class, 568, 569

## O

Object, 182, 183  
   casting, 248, 249  
   cloning, 288-291  
   comparing, 250, 251  
   creating, 64, 65, 196, 197  
   identity, 182, 183  
   state, 182, 183  
 Object class (java.lang), 226-229, 440, 441  
   enumerations, 314, 315  
 Object diagram, 182, 183  
 Object reference, 196, 197  
 OBJECT tag (HTML), 604, 605, 610, 611  
 OCTOBER field (Calendar class), 391  
 ODBC (Open Database Connectivity), 726, 727  
 ODBC data source, 728, 729  
 offer method (LinkedList class), 362, 363  
 One-dimensional array, 322-329  
 One-to-many relationship, 716, 717  
 One-to-one relationship, 716, 717  
 Operand, 56, 57, 96, 97  
   in a Boolean expression, 122, 123  
 Options (setting in TextPad), 22, 23  
 Or operator, 126, 127  
 Oracle, 714, 715  
 ORDER BY clause (SELECT statement), 720, 721  
 Order class (Order Queue application), 464, 465  
 Order of precedence, 100, 101  
 Order Queue application, 460-469  
 OrderHandler class (Order Queue application), 468, 469  
 OrderQueue class (Order Queue application), 468, 469  
 OrderQueueApp class, 464, 465  
 OrderTaker class (Order Queue application), 466, 467  
 ordinal method (enumeration constant), 314, 315  
 Outer class, 310, 311  
 Outer join, 722, 723  
 Output file, 628, 629  
 Output stream, 628, 629  
   binary, 650, 651  
   character, 634, 635  
 OutputStream class (java.io), 650, 651  
 OutputStream hierarchy, 650, 651  
 OutputStreamWriter class (java.io), 633  
 Overloading a method, 146  
 Overloading constructors, 188, 189  
 Overloading methods, 192, 193  
 Overriding equals method, 250, 251  
 Overriding methods, 224, 225  
   of a superclass, 234, 235  
   of an enumeration, 314, 315

## P

P tag (HTML), 604, 605  
 pack method (Window class), 534, 535  
 Package, 34, 35  
   compiling, 298, 299  
   Java, 62, 63  
   making available, 300, 301  
   name, 296, 297  
   storing classes in, 296, 297  
 package statement, 296, 297  
 Panel, 476, 477  
   adding to frame, 488, 489, 490, 491  
   modifying for applet, 602, 603  
 Parameter list of a method, 190, 146, 147  
 Parameters, 146, 147  
   interface type, 274, 275  
   prepared statement, 742, 743  
 Parent class, 224, 225  
 Parent element (XML), 682-684  
 parseDouble method (Double class), 108, 109  
 parseInt method (Integer class), 108, 109  
 parsing strings, 404, 405  
 Passing arguments by reference, 200, 201  
 Passing arguments by value, 200, 201  
 path, 626, 627  
 path command (DOS), 14, 15  
 Payment application, 538-543  
 peek method (LinkedList class), 362, 363  
 Pixels, 480, 481  
 PLAIN\_MESSAGE field (JOptionPane class), 572, 573  
 Platform independence, 8, 9  
 Plug-in (Java), 8, 9  
 Pointer (random-access file), 660, 661  
 poll method (LinkedList class), 362, 363  
 Polymorphism, 236, 237  
 Positive sign operator, 96, 97  
 Postfix an operand, 96, 97  
 pow method (Math class), 106, 107  
 Prefix an operand, 96, 97  
 Prepared statement, 742, 743  
 PreparedStatement interface (java.sql), 742, 743  
 prepareStatement method (Connection interface), 742, 743  
 Presentation layer, 178, 179  
 previous method (ResultSet interface), 736, 737  
 Primary key, 714, 715

Primitive data types, 54, 55, 92, 93  
     comparing, 122, 123  
     passing to method, 200, 201  
 print method (PrintWriter class), 636, 637  
 print method (System.out object), 68, 69  
 Printable interface, 268, 269  
 printf method (System.out object), 68, 70  
 Printing output to the console, 68, 69  
 println method (PrintWriter class), 636, 637  
 println method (System.out object), 68, 69  
 printStackTrace method (Throwable class), 426, 427  
 PrintStream class, 68  
 PrintWriter class (java.io), 632, 633, 636, 637  
 Priority of a thread, 452, 453  
 private keyword, 146, 147, 206, 207  
     abstract method, 253  
     class declaration, 50, 51  
     instance variable, 186, 187  
     method, 190, 191  
     superclass, 232, 233  
 Producer/consumer pattern, 460, 461  
 Producers, 460, 461  
 Product application with inheritance, 238-245  
 Product class, 242, 243  
     diagram, 180, 181  
     with Cloneable interface, 288, 289  
 Product Maintenance application, 278-287, 580-591  
 ProductApp class, 204, 205  
     with inheritance, 240, 241  
 ProductConstants interface, 276, 277, 642, 643  
 ProductDAO interface, 276-279  
     file I/O version, 642, 643  
 ProductDB class, 202, 203, 746, 747  
     with inheritance, 242, 245  
 ProductMaintApp class, 284-287  
 ProductRandomFile class, 666-673  
 ProductReader interface, 276, 277, 642, 643  
 ProductTextFile class, 282, 283, 644-649  
 ProductWriter interface, 268, 269, 276, 277, 642, 643  
 ProductXMLFile class, 702-709  
 Properties class (java.util), 772, 773  
 protected keyword  
     abstract method, 253  
     superclass, 232, 233  
 Protected members (superclass), 232, 233  
 public keyword, 146, 147, 206, 207  
     class declaration, 50, 51  
     enumeration, 312, 313  
     instance variable, 186  
     interface, 262, 268, 269  
     main method declaration, 52  
     method, 190, 191  
     outer class, 310, 311  
     superclass, 232, 233  
 Pull operation, 366, 367  
 Push operation, 366, 367  
 put method (HashMap class), 374-377

put method (Properties class), 772, 773  
 put method (TreeMap class), 374-377

## Q

Query, 720, 721  
 Querying tables, 720, 721  
 QUESTION\_MESSAGE field (JOptionPane class), 572, 573  
 Queue, 366, 367

## R

Radio buttons, 510, 511, 518, 519  
 random method (Math class), 106, 107  
 Random-access files, 660-665  
 RandomAccessFile class (java.io), 660-663  
 Range checking, 162, 163  
 RDBMS (relational database management system), 714  
 read method (BufferedReader class), 640, 641  
 readBoolean method (DataInput interface), 656, 657  
 readChar method (DataInput interface), 656, 657, 658, 659  
 readDouble method (DataInput interface), 656, 657  
 Reader class (java.io), 638, 639  
 Reader hierarchy, 638, 639  
 Reading  
     binary files, 656, 657  
     fixed-length strings, 664, 665  
     from text files, 640, 641  
     input from the console, 70, 71  
     random-access files, 662, 663  
 readInt method (DataInput interface), 656, 657  
 readLine method (BufferedReader class), 640, 641  
 readUTF method (DataInput interface), 656, 657, 658, 659  
 REAL data type (SQL), 754, 755  
 Record  
     database, 714, 715  
     file, 636, 637  
 Rectangular arrays, 338, 339  
 Red-black tree, 374  
 Reference to an array, 336, 337  
 Reference type, 124, 125  
     passing to method, 200, 201  
 Regions of containers, 496, 497  
 regular expression, 402, 403  
 Relational database, 714, 715  
 Relational operators, 74, 75, 122, 123  
 Relationships between tables, 716, 717  
 RELATIVE field (GridBagConstraints class), 536, 537  
 relative method (ResultSet interface), 736, 737  
 relative path, 626, 627  
 REMAINDER field (GridBagConstraints class), 536, 537  
 remove method (ArrayList class), 352-355  
 remove method (HashMap class), 374, 375  
 remove method (LinkedList class), 362, 363  
 remove method (TreeMap class), 374, 375  
 removeElementAt method (DefaultListModel class), 530, 531

removeFirst method (LinkedList class), 362-365  
 removeItem method (JComboBox class), 522, 523  
 removeItemAt method (JComboBox class), 522, 523  
 removeLast method (LinkedList class), 362-365  
 replace method (String class), 402, 403  
 replace method (StringBuilder class), 406, 407  
 requestFocusInWindow method (Component class), 480, 481  
 Result set, 720, 721
 

- moving through, 736, 737
- retrieving, 734, 735
- returning data, 738, 739

 ResultSetMetaData interface (java.sql), 752, 753  
 return statement, 146, 147, 190, 191  
 Return type, 146, 147, 190, 191  
 RIGHT field (FlowLayout class), 494, 495  
 Right outer join, 722  
 roll method (Calendar class), 390, 391  
 roll method (GregorianCalendar class), 390, 391  
 Root element (XML), 683, 684  
 Root pane, 488, 489  
 round method (Math class), 106, 107  
 Rounding, 105, 106  
 RoundingMode enumeration, 114, 115  
 Row
 

- database, 714, 715
- file, 636, 637

 RTTI (runtime type identification), 246  
 run method (Runnable interface), 440, 441  
 run method (Thread class), 440, 441, 445  
 Runnable interface (java.lang), 440, 441
 

- implementing, 448, 449

 Runnable state (thread), 442, 443, 450, 458, 459  
 Running an application
 

- from command prompt, 26, 27
- from TextPad, 22, 23

 Running SQL scripts (Derby database), 768-771  
 Runtime error, 24, 25, 84, 85  
 Runtime exception, 84, 85  
 RuntimeException (java.lang), 154, 155, 414, 415

## S

SATURDAY field (Calendar class), 391  
 SAX (Simple API for XML), 690, 691  
 Scale, 114, 115  
 Scanner class, 70, 71, 160, 161  
 Schema
 

- database, 774
- XML, 686, 687

 Scientific notation, 92, 93  
 Scroll pane, 510, 511, 514, 515
 

- with lists, 526, 527

 Scrollable result set, 734-737  
 SDK (Software Development Kit), 4, 5  
 SE (Standard Edition), 4, 5  
 SECOND field (Calendar class), 391  
 Security issues for applets, 598, 599

seek method (RandomAccessFile class), 662, 663  
 SELECT statement (SQL), 720, 721
 

- for joining tables, 722, 723
- for adding records, 724, 725

 selectAll method (JTextComponent class), 566, 567  
 Selection mode (list), 526  
 Selection structure, 76, 128, 129  
 Semantic events, 550, 551  
 SEPTEMBER field (Calendar class), 391  
 Sequential-access file, 660, 661  
 Servlet, 6, 7  
 Set command (DOS), 16, 17  
 Set interface (java.util), 348, 349  
 set method (ArrayList class), 352-355  
 set method (Calendar class), 390, 391  
 set method (LinkedList class), 362, 363  
 setBorder method (JComponent class), 520, 521  
 setBounds method (Component class), 480, 481  
 setCharAt method (StringBuilder class), 406, 407  
 setColumns method (JTextField class), 500, 501  
 setDaemon method (Thread class), 444, 445  
 setDefaultCloseOperation method (JFrame class), 484, 485  
 setDouble method (PreparedStatement interface), 742, 743  
 setEditable method (JComboBox class), 522, 523  
 setEditable method (JTextField class), 500, 501  
 setEnabled method (Component class), 480, 481, 490, 491  
 setFixedCellWidth method (JList class), 526, 527  
 setFocusable method (Component class), 480, 481  
 setFocusable method (JTextField class), 501  
 setLayout method (Container class), 494, 495, 534, 535  
 setLength method (RandomAccessFile class), 662, 663  
 setLength method (StringBuilder class), 406, 407  
 setLineWrap method (JTextArea class), 512, 513  
 setLocation method (Component class), 480, 481  
 setMaximumFractionDigits method (NumberFormat object), 104, 105  
 setMinimumFractionDigits method (NumberFormat object), 104, 105  
 setName method (Component class), 480, 481  
 setPriority method (Thread class), 444, 445, 452, 453  
 setProperty method (System class), 772, 773  
 setReadOnly method (File class), 624, 625  
 setResizable method (Frame class), 482, 483  
 setScale method (BigDecimal class), 114, 115  
 setSelected method (JCheckBox class), 516, 517  
 setSelectedIndex method (JComboBox class), 522, 523  
 setSelectedIndex method (JList class), 526, 527  
 setSelectionMode method (JList class), 526, 527  
 setSize method (Component class), 480, 481  
 setString method (PreparedStatement interface), 742, 743  
 setText (JButton class), 490, 491  
 setText method (JLabel class), 498, 499  
 setText method (JTextArea class), 512, 513  
 setText method (JTextField class), 500, 501  
 setTime method (Calendar class), 390, 391  
 Setting dates and times (GregorianCalendar class), 388, 389  
 setTitle method (Frame class), 482, 483

- setVisible method (Component class), 480, 481, 482, 483
- setVisibleRowCount method (JList class), 526, 527
- setWrapStyleWord method (JTextArea class), 512, 513
- SHORT field (DateFormat class), 394, 395
- short data type, 92, 93
- Short-circuit operators, 126, 127
- showMessageDialog method (JOptionPane class), 572, 573
- Signature
  - of a constructor, 188, 189
  - of a method, 146, 147, 192, 193
- Signed applet, 598, 599
- Significant digits, 92, 93
- SINGLE\_INTERVAL\_SELECTION field (ListSelectionModel interface), 528, 529
- SINGLE\_SELECTION field (ListSelectionModel interface), 528, 529
- Single-line comment, 46, 47
- Single-precision number, 93
- size method (ArrayList class), 352-355
- size method (DataOutputStream class), 652, 653
- size method (DefaultListModel class), 530, 531
- size method (HashMap class), 374, 375
- size method (LinkedList class), 362, 363
- size method (TreeMap class), 374, 375
- Size of an array, 322, 323
- skip method (BufferedReader class), 640, 641
- skipBytes method (DataInput interface), 656, 657
- sleep method (Thread class), 440, 441, 445, 450, 451
- SMALLINT data type (SQL), 754, 755
- Software class, 242, 244
- sort method (Arrays class), 330-333
- Source code, 8, 9
  - compiling from command prompt, 26, 27
  - compiling with TextPad, 22, 23
- source switch (javac command), 28, 29
- SOUTH field (BorderLayout class), 496, 497
- SOUTH field (GridBagConstraints class), 536, 537
- SOUTHEAST field (GridBagConstraints class), 536, 537
- SPACE constant (XMLStreamConstants interface), 698, 699
- Special characters, 60, 61
- split method (String class), 402-405
- SQL (Structured Query Language), 720
- SQL data types, 754, 755
- SQL scripts, 768-771
- SQL Server, 714, 715
- SQLException (java.sql), 730, 731, 772, 773
- sqrt method (Math class), 106, 107
- Stack class (java.util), 378, 379
- Stack trace, 154, 155, 416, 417
- start method (Applet class), 600, 601
- start method (Thread class), 440-447
- Start tag (XML), 682, 683
- START\_ELEMENT constant (XMLStreamConstants interface), 698-701
- Starting the Derby server, 782, 783
- startsWith method (String class), 402, 403
- State of an object, 182, 183
- Statement object, 734, 735
- Statements (Java), 46, 47
- Static fields
  - calling, 208, 209
  - coding, 206, 207
  - importing, 314, 315
  - when to use, 210
- Static import, 314, 315
- static initialization block, 210, 211
- Static inner class, 310, 311
- static keyword, 52, 146, 206, 207, 314, 315, 268, 269
- Static methods, 64, 65, 146-149
  - calling, 146, 147, 208, 209
  - coding, 146-147, 206, 207
  - importing, 314, 315
  - when to use, 210
- StAX (Streaming API for XML), 690-709
- stop method (Applet class), 600, 601
- Storing classes in packages, 296, 297
- Stream, 628, 629
- stream, 630, 631
- String class (java.lang), 58, 59, 124, 125, 400-405
- StringBuffer class (java.lang), 406, 407
- StringBuilder class (java.lang), 406-409
- Strings, 58-61, 400, 401
  - appending, 58, 59
  - binary, 658, 659
  - comparing, 74, 75, 124, 125, 402, 403
  - concatenating, 58, 59
  - database, 744, 745
  - joining, 58, 59
  - manipulating, 402, 403
  - parsing, 404, 405
- Subclass, 154, 155, 224, 225
  - casting to superclass, 248, 249
  - creating, 234, 235
- substring method (String class), 402-405
- substring method (StringBuilder class), 406-409
- subtract method (BigDecimal class), 114, 115
- Subtraction assignment operator, 99
- Subtraction operator, 96, 97
- Sun Java DB, 760, 761
- SUNDAY field (Calendar class), 391
- super keyword, 234, 235
- Superclass, 224, 225
  - casting to subclass, 248, 249
  - creating, 232, 233
- Swallowing exceptions, 418, 419, 450, 451
- Swing, 62, 63, 478, 479
  - classes, 226, 227
  - package, 475
- SwingValidator class, 576, 577
- switch statement, 130, 131
- Switches (javac command), 28, 29
- synchronized keyword, 456, 457
- Synchronizing threads, 456-459
- System class (java.lang), 772, 773

System.exit method, 492, 493

System.in object, 70, 71, 628

System.out object, 68, 69

## T

Tables (database), 714, 715

adding records, 724, 725

deleting records, 724, 725

querying, 720, 721

relationships, 716, 717

updating records, 724, 725

Tag (XML), 682, 683

Tagging interface, 266, 267

Terminated state (thread), 442, 443

Test Score application, 82, 83

Testing

applets from web browser, 612, 613

applets with Applet Viewer, 606, 607

applications, 84, 85

exception handlers, 424, 425

Text area, 510-513

adding to scroll pane, 514, 515

Text editor, 18

editing XML files, 688, 689

Text fields, 476, 477, 500, 501

validating, 574, 575

Text files, 628, 629, 634-641

delimited, 636, 637

reading, 640, 641

writing, 636, 637

TextPad

compiling source code, 22, 23

displaying line numbers, 20, 21

installing, 18, 19

running an application, 22, 23

setting options, 20, 21

working with source code, 20, 21

this keyword, 188, 189, 194, 195

Thread class (java.lang), 440, 441, 444, 445

extending, 446, 447

Thread scheduler, 438, 439, 442, 443

Thread states, 442, 443

Threads, 438, 439, 484, 485

communicating among, 458, 459

creating, 444-449

interrupting, 454, 455

life cycle, 442, 443

manipulating, 450-455

Order Queue application, 460, 461

putting to sleep, 450, 451

setting priority, 452, 453

synchronizing, 456-459

uses for, 438-440

ways to create, 440, 441

Three-tiered architecture, 178, 179

Throw an exception, 108, 109, 154, 155, 416, 417

throw statement, 424, 425

Throwable class (java.lang), 414, 415, 426, 427

Throwable hierarchy, 414, 415

Throwing a checked exception, 422, 423

throws clause, 422, 423

THURSDAY field (Calendar class), 391

TIME data type (SQL), 754, 755

Times (formatting), 394, 395

TIMESTAMP data type (SQL), 754, 755

TINYBIT data type (SQL), 754, 755

Title bar, 476, 477

TITLE tag (HTML), 604, 605

toArray method (ArrayList class), 352, 353

toArray method (LinkedList class), 362, 363

Token, 70, 71

Toolkit class (java.awt), 486, 487

toString method (BigDecimal class), 114, 115

toString method (Date class), 392, 393

toString method (Double class), 108, 109

toString method (Integer class), 108, 109

toString method (Object class), 228, 229

toString method (StringBuilder class), 406, 407

toString method (Throwable class), 426, 427

Tree map, 374, 375

TreeMap class (java.util), 348, 349, 374, 375

trim method (String class), 402-405

try statement, 156, 157, 418, 419

TUESDAY field (Calendar class), 391

Two-dimensional arrays, 338-341

Type variable, 350, 351

TYPE\_FORWARD\_ONLY field (ResultSet interface), 734, 735

TYPE\_SCROLL\_INSENSITIVE field (ResultSet interface),  
734, 735

TYPE\_SCROLL\_SENSITIVE field (ResultSet interface), 734,  
735

Typed collection, 350, 351

Types class (java.sql), 752, 753

Type-safe enumerations, 312, 313

## U

UML (Unified Modeling Language), 180, 181

Unary operator, 96, 97

UNC (Universal Naming Convention), 626, 627

Unchecked exception, 414, 415

Unicode character set, 92, 93

Unicode format, 20

Untyped collection, 380, 381, 382, 383

UPDATE statement (SQL), 724, 725

Updateable result set, 734, 735

updateDouble method (ResultSet interface), 740, 741

updateRow method (ResultSet interface), 740, 741

updateString method (ResultSet interface), 740, 741

Updating records in a database, 724, 725, 740, 741

URL (Uniform Resource Locator), 730, 731

## User interface

- Future Value Calculator application, 502, 503
- Product Maintenance application, 580, 581

User thread, 444, 445

UTF (Universal Text Format), 652, 653

**V**

## Validating entries

- console 162, 163
- Swing, 572-579

Validator class, 214, 216-217

VARBINARY data type (SQL), 754, 755

VARCHAR data type (SQL), 754, 755

Variable, 54, 55

- array, 322, 323
- initializing, 54, 55, 94, 95
- numeric, 54-57
- string, 58-61

Vector class (java.util), 378, 379

VERTICAL field (GridBagConstraints class), 536, 537

VERTICAL\_SCROLL\_AS\_NEEDED field  
(ScrollPaneConstants interface), 515

VERTICAL\_SCROLLBAR\_ALWAYS field  
(ScrollPaneConstants interface), 515

VERTICAL\_SCROLLBAR\_NEVER field  
(ScrollPaneConstants interface), 515

Viewing XML files, 688, 689

void keyword, 52, 190, 191

**W**

wait method (Object class), 440-443, 458, 459

Waiting state (thread), 442, 443, 458, 459

WARNING\_MESSAGE field (JOptionPane class), 572, 573

Web browser, 6

- testing applet, 612, 613
- viewing javadoc documentation, 306, 307

WEDNESDAY field (Calendar class), 391

weightx field (GridBagConstraints class), 536, 537

weighty field (GridBagConstraints class), 536, 537

WEST field (BorderLayout class), 496, 497

WEST field (GridBagConstraints class), 536, 537

WHERE clause (SELECT statement), 720, 721

while loop, 78, 79, 134, 135

Whitespace, 70, 71

Widening conversion, 102, 103

WIDTH attribute (APPLET tag), 604, 605

width field (Dimension class), 486, 487

Window class (java.awt), 226, 227, 478, 479, 534, 535

WindowConstants interface (javax.swing), 484, 485

WindowListener interface (java.awt.event), 266, 267

Wrapper class, 108, 109, 346, 350, 351

- untyped collections, 382, 383

writeAttribute method (XMLStreamWriter class), 694, 695

writeBoolean method (DataOutput interface), 652, 653

writeChar method (DataOutput interface), 652, 653

writeCharacters method (XMLStreamWriter class), 694, 695

writeChars method (DataOutput interface), 652, 653, 658, 659

writeComment method (XMLStreamWriter class), 694, 695

writeDouble method (DataOutput interface), 652, 653

writeDTD method (XMLStreamWriter class), 694, 695

writeEndElement method (XMLStreamWriter class), 694, 695

writeInt method (DataOutput interface), 652, 653

Writer class (java.io), 633

Writer hierarchy, 634, 635

writeStartDocument method (XMLStreamWriter class), 694, 695

writeStartElement method (XMLStreamWriter class), 694, 695

writeUTF method (DataOutput interface), 652, 653, 658, 659

## Writing

- binary files, 652, 653
- fixed-length strings, 664, 665
- random-access files, 662, 663
- text files, 636, 637

**X**

XML (Extensible Markup Language), 680, 681

XML APIs, 690, 691

XML attributes, 684, 685

XML declaration, 682, 683

XML document, 680, 681

XML file, 688, 689

XML tag, 682, 683

XMLInputFactory class (javax.xml.stream), 696, 697

XMLOutputFactory class (javax.xml.stream), 692, 693

XMLStreamException class (javax.xml.stream), 692, 693

XMLStreamException class (javax.xml.stream), 696, 697

XMLStreamReader class (javax.xml.stream), 696, 697

XMLStreamWriter class (javax.xml.stream), 692-695

**Y**

YEAR field (Calendar class), 391

yield method (Thread class), 445, 446, 447



**For more on Murach products, visit us at**  
**www.murach.com**

## **For professional developers**

Murach's Java SE 6	\$52.50
Murach's Java Servlets and JSP (Second Edition)	52.50
Murach's Oracle SQL and PL/SQL	52.50
Murach's JavaScript and DOM Scripting	\$54.50
Murach's C# 2008	\$52.50
Murach's ASP.NET 3.5 Web Programming with C# 2008	52.50
Murach's ADO.NET 3.5, LINQ, and the Entity Framework with C#	52.50
Murach's Visual Basic 2008	52.50
Murach's ASP.NET 3.5 Web Programming with VB 2008	52.50
Murach's ADO.NET 3.5, LINQ, and the Entity Framework with VB	52.50
Murach's SQL Server 2008 for Developers	52.50
Murach's OS/390 and z/OS JCL	\$62.50
Murach's Mainframe COBOL	59.50
Murach's CICS for the COBOL Programmer	54.00

*\*Prices and availability are subject to change. Please visit our web site or call for current information.*

## **Our unlimited guarantee...when you order directly from us**

You must be satisfied with our books. If they aren't better than any other programming books you've ever used...both for training and reference...you can send them back within 90 days for a full refund. No questions asked!

### **Your opinions count**

If you have any comments on this book, I'm eager to get them. Thanks for your feedback!



### **To comment by**

E-mail: [murachbooks@murach.com](mailto:murachbooks@murach.com)  
Web: [www.murach.com](http://www.murach.com)  
Postal mail: Mike Murach & Associates, Inc.  
4340 N. Knoll Ave.  
Fresno, California 93722

### **To order now,**



**Web:** [www.murach.com](http://www.murach.com)



**Call toll-free:**

1-800-221-5528  
(Weekdays, 8 am to 4 pm Pacific Time)



**Fax:** 1-559-440-0963



**Mike Murach & Associates, Inc.**  
*Professional programming books*

## What software you need for this book

---

- Java SE 6 (JDK 1.6). You can download this software for free from [java.sun.com](http://java.sun.com). Then, you can install and configure this software as described in chapter 1.
- A text editor that's designed for working with Java, such as TextPad. You can download a trial version of this software for free from [www.textpad.com](http://www.textpad.com). Then, you can install and configure this software as described in chapter 1.

## The downloadable files for this book

---

- Complete source code and data for the applications presented in this book so you can view, compile, and run the code for the applications as you read each chapter.
- Starting source code and data for the exercises presented at the end of each chapter so you can get more practice in less time.

## How to download the files for this book

---

- Go to [www.murach.com](http://www.murach.com), and go to the page for *Murach's Java SE 6*.
- Click the link for "FREE download of the book applications." Then, download "All book applications." This will download a file named `jse6_allfiles.exe` to your computer.
- Use the Windows Explorer to find the exe file that you downloaded. Then, double-click this file and respond to the dialog boxes that follow. This installs the files in directories that start with `c:\murach\java6`.
- From that point on, you can find the applications in `c:\murach\java6\applications`, and you can find the starting points for the exercises in `c:\murach\java6\exercises`.

## How to prepare your system for the exercises

---

- Before you do the exercises, you'll want to copy (not move) the `java1.6` subdirectory and all of its subdirectories from `c:\murach\java6\exercises` to the root directory of the C drive. From that point on, you can find the starting points for the exercises in chapter directories like `c:\java1.6\ch01` and `c:\java1.6\ch02`.

## How to use an IDE with this book

---

- To make it easy for you to use an Integrated Development Environment (IDE) like Eclipse, NetBeans, or BlueJ with this book, we have created free tutorials for these IDEs that you can download from our web site. In addition, you can download the applications for this book in formats that are compatible with these IDEs. To learn more, please read pages 36-39 in chapter 1.