A Reference for the Rest of Us!

by Wayne Holder & Doug Bell
### <APPLET> Tag Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CODE</td>
<td>Required: The name of the Java class file containing the compiled Applet subclass to execute.</td>
</tr>
<tr>
<td>WIDTH</td>
<td>Required: The suggested pixel width and height of the area the browser should reserve for the applet in the Web page.</td>
</tr>
<tr>
<td>HEIGHT</td>
<td>The uniform resource locator (URL) of the directory or folder that contains the applet code. If CODEBASE is not specified, then the Web browser viewing the document defaults to the location of the HTML document. CODEBASE allows the applet code to be placed in a different location than the HTML.</td>
</tr>
<tr>
<td>CODEBASE</td>
<td>The applet name that other applets on the Web page can use to find it and communicate with it.</td>
</tr>
<tr>
<td>ALT</td>
<td>Text displayed by browsers that cannot run the applet. The ALT text is displayed, for instance, if the user has turned off the Java option in their browser.</td>
</tr>
<tr>
<td>ALIGN</td>
<td>The alignment of the applet relative to the text line containing it. This attribute works like the ALIGN attribute for the IMG tag. The possible values are top, middle, bottom, left, and right. The alignment is bottom by default.</td>
</tr>
<tr>
<td>HSPACE</td>
<td>The number of pixels of space the browser should leave around the applet on the left and right (HSPACE) and top and bottom (VSPACE).</td>
</tr>
<tr>
<td>VSPACE</td>
<td></td>
</tr>
</tbody>
</table>

### Built-in Java Colors

<table>
<thead>
<tr>
<th>Color</th>
<th>RGB values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Color.black</td>
<td>0,0,0</td>
</tr>
<tr>
<td>Color.blue</td>
<td>0,0,255</td>
</tr>
<tr>
<td>Color.cyan</td>
<td>0,255,255</td>
</tr>
<tr>
<td>Color.darkGray</td>
<td>64,64,64</td>
</tr>
<tr>
<td>Color.gray</td>
<td>128,128,128</td>
</tr>
<tr>
<td>Color.green</td>
<td>0,255,0</td>
</tr>
<tr>
<td>Color.lightGray</td>
<td>192,192,192</td>
</tr>
<tr>
<td>Color.magenta</td>
<td>255,0,255</td>
</tr>
<tr>
<td>Color.orange</td>
<td>255,200,0</td>
</tr>
<tr>
<td>Color.pink</td>
<td>255,175,175</td>
</tr>
<tr>
<td>Color.red</td>
<td>255,0,0</td>
</tr>
<tr>
<td>Color.white</td>
<td>255,255,255</td>
</tr>
<tr>
<td>Color.yellow</td>
<td>255,255,0</td>
</tr>
</tbody>
</table>

### Commonly Overridden Applet Methods

<table>
<thead>
<tr>
<th>Applet Method</th>
<th>Override It To...</th>
</tr>
</thead>
<tbody>
<tr>
<td>void init()</td>
<td>Perform any one-time initialization the applet needs before it runs.</td>
</tr>
<tr>
<td>void start()</td>
<td>Begin animations, processing, or threads.</td>
</tr>
<tr>
<td>void paint()</td>
<td>Draw the applet to the screen.</td>
</tr>
<tr>
<td>void stop()</td>
<td>Suspend animations, processing, or threads initiated in start().</td>
</tr>
<tr>
<td>void destroy()</td>
<td>Clean up after the applet before it quits.</td>
</tr>
</tbody>
</table>
Drawing Outlined Shapes and Lines

<table>
<thead>
<tr>
<th>Shape Outline</th>
<th>Method and Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rectangle</td>
<td>drawRect(int x, int y, int width, int height)</td>
</tr>
<tr>
<td>3-D Rectangle</td>
<td>draw3DRect(int x, int y, int width, int height, boolean raised)</td>
</tr>
<tr>
<td>Rounded Rectangle</td>
<td>drawRoundRect(int x, int y, int width, int height, int arcWidth, int arcHeight)</td>
</tr>
<tr>
<td>Oval</td>
<td>drawOval(int x, int y, int width, int height)</td>
</tr>
<tr>
<td>Arc</td>
<td>drawArc(int x, int y, int width, int height, boolean raised, int startAngle, int arcAngle)</td>
</tr>
<tr>
<td>Polygon</td>
<td>drawPolygon(int[] xPoints, int[] yPoints, int nPoints)</td>
</tr>
<tr>
<td>Line</td>
<td>drawLine(int x1, int y1, int x2, int y2)</td>
</tr>
</tbody>
</table>

Drawing Filled Shapes and Text

<table>
<thead>
<tr>
<th>Filled Shape</th>
<th>Method and Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rectangle</td>
<td>fillRect(int x, int y, int width, int height)</td>
</tr>
<tr>
<td>3-D Rectangle</td>
<td>fill3DRect(int x, int y, int width, int height)</td>
</tr>
<tr>
<td>Rounded Rectangle</td>
<td>fillRoundRect(int x, int y, int width, int arcWidth, int arcHeight)</td>
</tr>
<tr>
<td>Oval</td>
<td>fillOval(int x, int y, int width, int height)</td>
</tr>
<tr>
<td>Arc</td>
<td>fillArc(int x, int y, int width, int height, boolean raised, int startAngle, int arcAngle)</td>
</tr>
<tr>
<td>Polygon</td>
<td>fillPolygon(int[] xPoints, int[] yPoints, int nPoints)</td>
</tr>
<tr>
<td>Text String</td>
<td>drawString(String str, int x, int y)</td>
</tr>
</tbody>
</table>

Useful HTML Tags

<table>
<thead>
<tr>
<th>Tag</th>
<th>Example Usage</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;A&gt;</td>
<td>&lt;A href=&quot;http://www.idgbooks.co.com&quot;&gt;IDG Books&lt;/A&gt;</td>
<td>The anchor tag creates a link to another document or Web page, in this case the IDG Books Web site.</td>
</tr>
<tr>
<td>APPLET</td>
<td>&lt;APPLET CODE=&quot;MyApplet&quot; WIDTH=80 HEIGHT=50&gt;&lt;/APPLET&gt;</td>
<td>Insert a Java applet, in this case an applet with the filename MyApplet.</td>
</tr>
<tr>
<td>IMG</td>
<td>&lt;IMG SRC=&quot;image.gif&quot;&gt;</td>
<td>Insert a GIF image. IMG doesn't require an endtag.</td>
</tr>
<tr>
<td>&lt;P&gt;</td>
<td>This is a new paragraph&lt;br /&gt;</td>
<td>Starts a new paragraph. An end tag &lt;P&gt; is not required, but is good practice.</td>
</tr>
<tr>
<td>&lt;FONT&gt;</td>
<td>&lt;FONT SIZE=5 COLOR=RED&gt;</td>
<td>Set the font size and/or color of the contained text.</td>
</tr>
<tr>
<td></td>
<td>&lt;B&gt;Big Red Text&lt;/b&gt; &lt;/FONT&gt;</td>
<td>Display the contained text with a bold face font.</td>
</tr>
<tr>
<td>&lt;TT&gt;</td>
<td>&lt;TT&gt;Monospaced text&lt;/TT&gt;</td>
<td>The teletype tag displays the contained text using monospaced text.</td>
</tr>
<tr>
<td></td>
<td>&lt;I&gt;Italic text&lt;/I&gt; &lt;/I&gt;</td>
<td>Italicize the contained text.</td>
</tr>
<tr>
<td>&lt;U&gt;</td>
<td>&lt;U&gt;Underlined text&lt;/U&gt; &lt;/U&gt;</td>
<td>Underline the contained text.</td>
</tr>
</tbody>
</table>

For Dummies: #1 Computer Book Series for Beginners
# Table of Contents

**Introduction** ................................................................. 1

  - About This Book ................................................................................................................. 1
  - Who You Are.................................................................................................................. 1
  - About the Java Code in This Book..................................................................................... 2
  - How This Book Is Organized ............................................................................................ 2
    - Part I: Steppin' Out ........................................................................................................... 2
    - Part II: Up to Speed ........................................................................................................ 2
    - Part III: Seven League Boots ........................................................................................ 3
    - Part IV: The Part of Tens ............................................................................................... 3
  - Appendix: About the CD-ROM ......................................................................................... 3
    - CD Chapters: Fundamentals............................................................................................ 3
    - Icons Used in This Book ................................................................................................. 4

**Part 1: Steppin’ Out............................................................... 5**

**Chapter 1: Follow the Bouncing Ball .............................................................. 7**

  - Ticking Off the Time ........................................................................................................... 7
  - Making Things Move .......................................................................................................... 9
    - Floating the point ............................................................................................................ 9
    - Encapsulating the essence of a ball ................................................................................. 9
  - Setting Bounds .................................................................................................................. 10
    - Moving out of bounds .................................................................................................... 11
    - Bouncing back ............................................................................................................... 11
    - Coding movement and bounce ...................................................................................... 11
    - Settin’ things in motion .................................................................................................... 13
  - Drawing the Details ......................................................................................................... 14
    - Drawing offscreen .......................................................................................................... 15
    - Overriding the flicker ...................................................................................................... 15
    - Drawing the background and the ball .............................................................................. 16
    - Putting the action on the screen ...................................................................................... 16

**Chapter 2: Ponglet....................................................................................... 17**

  - Setting State .................................................................................................................... 17
    - Breaking down the task .................................................................................................... 18
    - Serving the ball ................................................................................................................ 20
  - Up Java Creek without a Paddle ........................................................................................ 22
    - Returning the serve ......................................................................................................... 23
    - Changing state ................................................................................................................ 24
    - Creating a computer opponent ....................................................................................... 24
    - Rolling down the gutter .................................................................................................. 25
    - He shoots, he scores! ........................................................................................................ 26
    - We have a winna! .............................................................................................................. 26
# Table of Contents

## Introduction ................................................................. 1
- About This Book .................................................................................................. 1
- Who You Are ......................................................................................................... 1
- About the Java Code in This Book .................................................................... 2
- How This Book Is Organized .............................................................................. 2
  - Part I: Steppin' Out .................................................................................... 2
  - Part II: Up to Speed ..................................................................................... 2
  - Part III: Seven League Boots ..................................................................... 3
  - Part IV: The Part of Tens ........................................................................... 3
- Appendix: About the CD-ROM ........................................................................... 3
- CD Chapters: Fundamentals ..................................................................... 3
- Icons Used in This Book ..................................................................................... 4

## Part I: Steppin' Out ....................................................... 5

### Chapter 1: Follow the Bouncing Ball.......................................................... 7
- Ticking Off the Time ............................................................................................ 7
- Making Things Move ........................................................................................... 9
  - Floating the point....................................................................................... 9
  - Encapsulating the essence of a ball ........................................................ 9
- Setting Bounds ................................................................................................... 10
  - Moving out of bounds ............................................................................. 11
  - Bouncing back .......................................................................................... 11
  - Coding movement and bounce .............................................................. 11
  - Settin' things in motion........................................................................... 13
- Drawing the Details ........................................................................................... 14
  - Drawing offscreen .................................................................................... 15
  - Overriding the flicker .............................................................................. 15
  - Drawing the background and the ball .................................................. 16
  - Putting the action on the screen ........................................................... 16

### Chapter 2: Ponglet........................................................................................ 17
- Setting State ....................................................................................................... 17
- Breaking down the task .................................................................................. 18
- Serving the ball ................................................................................................ 20
- Up Java Creek without a Paddle ...................................................................... 22
  - Returning the serve .................................................................................. 23
  - Changing state ........................................................................................... 24
  - Creating a computer opponent .............................................................. 24
  - Rolling down the gutter ........................................................................... 25
  - He shoots, he scores! ............................................................................... 26
  - We have a winna! ...................................................................................... 26
Table of Contents

Conserving Momentum .................................................................................... 63
Revisiting vectors ....................................................................................... 64
What if both balls are moving? ................................................................. 66
The dot product ......................................................................................... 66
The collide() method .............................................................................. 67
collide() dissected .................................................................................... 67
Putting All the Pieces Together ............................................................... 68

Part II: Up to Speed .................................................................................. 71

Chapter 5: Sliding Blocks Brain Teaser .................................................... 73
Using Images in Games ............................................................................. 74
Drawing while downloading ..................................................................... 77
Loading images with MediaTracker .......................................................... 78
MediaTracker.addImage() ......................................................................... 78
MediaTracker.waitForAll() ................................................................. 78
Loading multiple images ......................................................................... 79
Laying Out the Game Board ..................................................................... 79
Reading the width and height of an Image .............................................. 81
Initializing gridX, gridY, pieceWidth, and pieceHeight ......................... 81
Crafting the Puzzle .................................................................................. 82
Making puzzle pieces that act like real puzzle pieces ......................... 82
Putting the pieces together ..................................................................... 83
Mousing the Pieces Around ................................................................... 85
Selecting a puzzle piece .......................................................................... 85
Moving the pieces ................................................................................... 86
Slide() ing around .................................................................................. 87
Checking for pieces that block the slide path with
Rectangle. intersects() ........................................................................... 87
Checking for the board boundaries Rectangle.union() and
Rectangle.equals() .................................................................................. 88
Cleaning up after a move ........................................................................ 89
Drawing the Board .................................................................................. 90
Declaring the Puzzle Solved and Congratulating the Winner ............... 91

Chapter 6: Blackjack ................................................................................ 93
Understanding the Blackjack Game ....................................................... 93
Playing Blackjack .................................................................................... 94
Designing the game .................................................................................. 95
Creating a Reusable Deck of Cards ....................................................... 96
Shuffling and dealing the deck ............................................................... 97
Building the Card class ......................................................................... 99
Converting cards to strings .................................................................... 102
Extracting card graphics from a composite image .............................. 103
Customizing the deck .......................................................................... 105
# Table of Contents

## Chapter 8: 2-D Sprite Maze

- Gentleman, Start Your Sprite Engines! .............................................. 174
  - Implementing a sprite .......................................................... 174
  - Putting sprites in their place ............................................... 176
  - Moving sprites around the play field .................................... 178
  - Resolving collisions ............................................................. 179
  - Displaying sprites ............................................................... 180
  - Animating sprites ............................................................... 181
- A Sprite Framework ........................................................................ 183
  - The **SpriteEngine** class ................................................. 184
    - Keeping track of all the sprites ...................................... 188
    - Drawing sprites layer by layer ...................................... 189
    - Moving sprites and detecting collisions ....................... 190
    - Improving the accuracy of collision detection .............. 190
    - Selecting a movement frame rate ................................. 192
  - The **BackgroundSpriteEngine** class ............................... 194
  - Sprite events and handling them ........................................ 194
  - Sprite control ..................................................................... 195
- Computer Adversaries .................................................................... 197
  - Using random intelligence to make adversaries smarter ....... 197
  - Using a breadth-first search for adversary navigation ......... 198
  - Prioritizing adversary goals .............................................. 198
- The Sprite Maze Game ................................................................... 200
  - Implementing a cast of sprites ........................................... 201
    - Running into a wall ......................................................... 202
    - Animating maze runners ................................................ 202
    - Animating an adversary who shoots to kill .................... 204
    - Whizzing bullets ............................................................ 205
  - Building on the **BlockMaze** class ................................... 206
    - Initializing the game ........................................................ 210
    - Overriding **drawSquare( )** .......................................... 210
    - Giving the player control ............................................... 211
    - Keeping things moving ................................................... 211
    - Chasing the player ......................................................... 212
    - Finalizing the Sprite Maze applet ................................. 212

## Part III: Seven League Boots

### Chapter 9: Modeling the Real World

- Making Things Happen at the Right Time with a Timeline .......... 217
  - A heap of events ............................................................... 218
  - Adding events to the timeline ............................................ 219
  - Processing events in order ................................................ 221
  - Changing the future: Removing events before they happen ...... 222
    - Removing events ........................................................... 222
    - Searching the timeline ................................................... 222
  - Playing Sounds ................................................................. 223
Matching Animations to Game Events with Scripts ............................................................... 224
  Interfacing the programmer and the artist ........................................................................ 225
  Writing a script ................................................................................................................. 225
  Reading scripts from text files .......................................................................................... 227
  Looping an animation ......................................................................................................... 228
  Adding random behavior .................................................................................................... 228
  Adding special effects and other goodies ......................................................................... 230
  Understanding the code ...................................................................................................... 231
  Organizing scripts by action .............................................................................................. 231
  Filling a script with frames ................................................................................................ 233
  Implementing an Animation Frame .................................................................................. 238
    SoundFrame .................................................................................................................. 238
    BranchFrame ................................................................................................................ 239
  Putting the code to work: The Sprite class ................................................................. 240

Chapter 10: 3-D Polygon Maze .......................................................................................... 243
  Moving into Three Dimensions ....................................................................................... 243
  Calculating perspective .................................................................................................... 243
  Calculating the height of a wall ....................................................................................... 247
  Finding the x-axis intersection ......................................................................................... 247
  Expanding the grid into 3 dimensions ............................................................................. 247
  Sizing up the screen ......................................................................................................... 247
  Drawing the Maze ............................................................................................................ 248
    The painter’s algorithm .................................................................................................. 248
    Draw from the outside in ............................................................................................... 248
    Deeper is wider ............................................................................................................. 249
  Creating a Rat’s-Eye View ............................................................................................... 250
    Writing GridView ......................................................................................................... 250
    Coding MazeMap ......................................................................................................... 252
    Coding PolyMaze ......................................................................................................... 253
  Adding Shading, Light Effects, and a Reason to Solve the Maze ........................................ 255
    Updating MazeMap ...................................................................................................... 257
    Updating PolyMaze .................................................................................................... 258
  Running a Random Maze ............................................................................................... 259
    Extending from BlockMaze ......................................................................................... 260
    Sizing the maze in your HTML .................................................................................... 260

Chapter 11: Texture-Mapped 3-D Maze ............................................................................. 263
  Mapping Some Texture ..................................................................................................... 263
  Scaling Images .................................................................................................................. 264
  Tiling Textures .................................................................................................................. 268
  Texture Mapping a 3-D Maze ........................................................................................... 269
    Introducing Mr. Bresenham ......................................................................................... 270
    Experimenting with Bresenham ................................................................................... 271
    Extending a TexView class from GridView .................................................................... 273
    Loading textures ........................................................................................................... 273
    Overriding drawSquare ............................................................................................ 273
    Alternating wall textures ........................................................................................... 274
Table of Contents

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Drawing front walls</td>
<td>275</td>
</tr>
<tr>
<td>Calculating the front wall's texture offset</td>
<td>275</td>
</tr>
<tr>
<td>Creating the front wall image</td>
<td>276</td>
</tr>
<tr>
<td>Clipping to the view</td>
<td>276</td>
</tr>
<tr>
<td>Slicing a column of texture</td>
<td>277</td>
</tr>
<tr>
<td>Drawing side walls</td>
<td>278</td>
</tr>
<tr>
<td>Calculating the side wall's texture offset</td>
<td>279</td>
</tr>
<tr>
<td>Tracing the side-wall edges</td>
<td>280</td>
</tr>
<tr>
<td>Masking the side walls</td>
<td>280</td>
</tr>
<tr>
<td>Darkening the walls</td>
<td>280</td>
</tr>
<tr>
<td>Computing a darkened color table</td>
<td>280</td>
</tr>
<tr>
<td>Shading the walls</td>
<td>281</td>
</tr>
<tr>
<td>Assembling the Pieces</td>
<td>283</td>
</tr>
</tbody>
</table>

Chapter 12: Advanced Imaging ................................................................. 285

Drawing Partially Transparent Images ......................................................... 286
Creating new images with Memory Image Source ........................................... 286
Coding an Alpha Gradient ............................................................................. 287
Blending the edges of images with alpha masking ....................................... 289
Creating alpha information from a GIF image ............................................ 289
Using Pixel Grabber ...................................................................................... 290
Antialiasing in Java .................................................................................... 293
Rendering to subpixels ............................................................................... 293
Reading from offscreen images .................................................................... 294
Shrinking text ............................................................................................... 296
Drawing Direct .............................................................................................. 297
The ImageProducer interface ......................................................................... 298
Coding an ImageProducer ............................................................................. 298
Dancing the ImageProducer tango ................................................................... 299
Demoing DirectImage ..................................................................................... 301
Modifying GIF Images ................................................................................... 304
Getting at the raw image data with the ImageConsumer interface .................. 304
Recoloring a GIF Image .................................................................................. 307

Part 1 U: The Part of Tens .......................................................................... 309

Chapter 13: Ten Secrets for Making Fun Games ......................................... 311
Knowing What Players Want ............................................................................ 311
Understanding What Makes a Game Addictive ............................................ 312
Start Easy and Then Increase Difficulty .................................................... 312
Making It Easy to "Step In." ........................................................................ 313
Enhancing the Player's Suspension of Disbelief ......................................... 313
Making the Player Feel Smart ....................................................................... 314
What Did I Do Wrong? The Player Should Always Know ............................... 314
## Table of Contents

- GoldWave 3.24 ............................................................... 339
- SoundForge XP 4.0d Demo ............................................. 339
- SoundApp 2.4.4 ............................................................. 339
- Soundlack 0.872 ............................................................ 340
- If You've Got Problems (Of the CD Kind) ......................... 340

### Index

- javaTm Development Kit Version 1.0.2 (Mac OS)
- 1.1.5 (windows) Binary Code License .............................. 356

- IDG Books Worldwide, Inc., End-User License Agreement .................................................. 358

- Installation Instructions ................................................ 360

- Book Registration Information ................................. Back of Book
Welcome to Java Game Programming For Dummies. This book takes you from writing your first, basic game applets all the way through advanced, texture-mapped 3-D. Along the way, you see and apply all the under-the-hood techniques like maze generation, collision detection, and sprites that put the red meat in your game stew.

About This Book

This book shows you the techniques that make games tick, and gives you dozens of working Java code examples. In addition, each example is backed up by detailed explanations that fully deconstruct the code so that you can see how everything works. You can start from these working examples and customize them, use the parts to create entirely new games, or simply use them as a source of ideas for writing your own custom game code.

While this book does, where necessary, discuss a little theory, the real heart of the book is intended more like a hands-on auto shop class than a physics lecture. After all, understanding how a water pump works is a lot easier if you can hold one in your hand and see where it fits on a real car engine. Likewise, understanding game code is a lot easier if you can examine each part of the code in detail and see where it fits in the overall structure of a working program.

Who You Are

We wrote this book in such a way that it is accessible to all levels of Java programmers. If you are fairly new to Java, you can copy the code in this book and, with the tips and instruction we give (and a little adventure), easily customize the games we present. You can, for example, take the JavaPool applet in Chapter 4 and easily figure out how to change the color of the pool table and balls, tweak the speed of play, and so on. If you find that this book is really beyond your understanding, buy it anyway and then also buy Java Programming For Dummies by Donald J. Koosis and David Koosis (IDG Books Worldwide, Inc.) - no seriously, this book doesn't go into detail about the most basic stuff, so if you've never touched Java before, you may want to start with the Koosis' book.
On the other hand, you experienced programmers can find a whole load of tips and game-specific programming techniques in this book. You can also copy and tweak the code we present, as well as get exposure to many game programming techniques to use in creating your own Java games.

About the java Code is This Book'

All the code examples in this book are coded as Java applets so that they can be used with Java-enabled Web browsers and published on the Web. At the time of this writing, the current release of Java is release 1.1.5 with version 1.2 just appearing as a developer release. Java versions 1.1 and later add many new features, such as a completely new event model, but many Web browsers have yet to fully incorporate these new features. Therefore, the applets in this book are coded to be compatible with the earlier Java 1.0.2 standard so that they work with the widest variety of Web browsers.

How This Book Is Organized

This book is divided into three major parts, each covering a progressively more involved array of game programming techniques. We then include three more elements, each with useful tips and additional information. As with all ... For Dummies books, you can pretty much dip in and out of chapters to find information. The only exception is that in some cases, a later section uses material or pieces of code from earlier chapters. We always alert you to these cases when they arise so that you know where to look, and you can always just go to the CD-ROM and pull in the necessary code if you need to.

Part 1: Steppin’ Out

This part covers the basics of animation and simulation and shows you how to program imaginary objects to obey physical rules, such as momentum, acceleration and rebounding from collisions. In this part, you create a Ping-Pong game, putting green, and pool table while exploring some advanced concepts, such as vector math, in a fun, straightforward way.

Part II: up to Speed

This next part introduces the techniques you need to create professional-quality games. Moving beyond the simple, solid-colored graphics of Part 1,
Part II shows you how to use multicolor images in your games. Starting with a logic puzzle, you progress to a multiplayer blackjack game, master 2-D sprites, and combine sprites with code to generate random mazes and create a maze chase game.

**Part Ill: Seven League Boots**

This part moves you beyond the flat world of 2-D games into the realm of 3-D flat-shaded and texture-mapped graphics, and shows you how to create several different styles of 3-D maze games. You also experiment with a variety of advanced game programming techniques, such as using timelines, employing animation scripts, playing sounds, and using the alpha channel to create spectacular image effects - all in 100 percent Java.

**Part U/: The mart of Tens**

If you've previously read any *For Dummies* books, you know that this section is intended to pull together a variety of useful facts and other goodies that just don't fit anywhere else. This book includes "Ten Secrets for Making Fun Games," "Ten Ways to Say Game Over" and "Ten Ways to Optimize Your Java Code."

**Appendix: About the CD-ROM**

The last section in this book contains information on the programs and applets included on the *Java Game Programming For Dummies* CD-ROM.

**Ca Chapters: Fundamentals**

The CD-ROM included with this book contains an additional five chapters of the book in a part called "Fundamentals" which is provided as Adobe Acrobat PDF files on the CD-ROM. These chapters cover many aspects of Java that are particularly useful for game programming, but not necessarily specific to game programming. If you're still new to coding Java and want to brush up on the fine points of applets, threads, graphics, color, user input, or basic HTML, you should check out these chapters. Whenever we discuss topics that rely on information in the CD Chapters, we also include a helpful reference to the appropriate chapter.
The tip icon marks information that can save you time or keep you out of trouble.

This icon introduces a special technique or programming trick that can help you program games like the pros.

This icon points out Java 1.1 differences from Java 1.02.

This icon points out Java 1.2 differences from Java 1.1 or Java 1.02.

This icon marks important information that you need to understand and use later.

_Danger, Will Robinson!_ Ignore this icon at your own peril because the advice given can often save you from making a serious error. However, with appropriate attention, you'll have smooth sailing ahead.

This icon introduces a technical term that can help you find information on this topic in other reference books. You can also sprinkle these terms into your daily conversation to impress your friends.

This icon refers you to stuff you can find on the _Java Game Programming For Dummies_ CD-ROM included at the back of this book.

This icon points out technical details that may be interesting to you, but which are not essential to understanding the topic under discussion.
Part I

Steppin' Out
In this part ... Simulation is at the heart of many computer games because many of them are adapted from games you can play in the real world. Simulation is a tricky subject, though, because you can't put real balls and Ping-Pong paddles into a computer game program. Instead, you have to write code that mimics how these objects act in the real world. Simulation is as much an art as it is a science, and Part I gives you a good solid foundation in both the craft and the technique of simulation.
Follow the Bouncing Ball

Moving objects across the screen is one of the basic skills you need to create action games. The way you simulate motion in a computer game is fairly simple: First, you break time down into a small unit, such as \( 1/100 \) second. Then, between each tiny tick of time, you move the object a small amount. When you repeat this process quickly enough, the small steps of movement blend together to create the illusion of motion.

This chapter discusses the various details and techniques used for animating and modeling a bouncing ball. The completed applet and applet code is on the Java Programming For Dummies CD-ROM.

Chapter 1

Off the Time

Java's Th read class lets you easily construct a program that slices time into tiny intervals using method s l eep ( ) to rest for specified intervals of time. You create a T h r e a d and then use a loop that alternates between doing something, such as updating the position of your object, and sleeping. The framework code you need to set up this alternation is

```java
public class Bounce extends Applet implements Runnable {
    private Thread ticker;
    private boolean running = false;

    public void run () {
        while (running) {
            repaint();
            // Update object position
            // Do something else
            try { Thread.sleep(100); } catch (InterruptedException e) {
            }
        }
    }
}
```
This applet extends the Runnable interface so that it can start the new
ticker Thread in the applet's start() method. The start() method also
sets the boolean variable running to true to tell the run() method to
continue to sleep and loop for as long as running remains true. When
it's time for the animation to stop, the stop() method sets running to
false and the run() method exits.

If the browser calls the start() method again after it has stopped the applet,
the isAlive() method returns false to indicate that the ticker Thread
is no longer running. In response, the code creates a new Thread to restart
the animation.

Your animation code needs to respect the applet's life cycle as described in
the previous paragraph; otherwise the animation can continue to run - even
after the user leaves the page containing your applet - and waste CPU cycles.

The calculation 1000 / 30 inside the call to sleep sets the animation
rate for the applet. The sleep() method expects to be told how long to
sleep in units of 1 millisecond. A millisecond is one 1,000th of a second, so
dividing 1,000 by 30 calculates a time in milliseconds that results in the
animation repeating roughly 30 times a second.

The previous code example provides the applet with a heartbeat, so to speak,
to drive the animation. However, the sole task of the timing loop in run() is
simply to sleep and to call repaint(). You need additional code to make
the applet compute and display the next step, or frame, in the animation.
Part 1: Steppin' Out

(continued)

```java
try {
    ticker.sleep(1000 / 15);
} catch (InterruptedException e) {  
    // Code to handle interruption
}
```  

```java
public synchronized void start () {
    if (ticker == null || !ticker.isAlive()) {
        running = true;
        ticker = new Thread(this);
        ticker.setPriority(Thread.MIN_PRIORITY + 1);
        ticker.start();
    }
}
```

```java
public synchronized void stop () {
    running = false;
}
```

This applet extends the `Runnable` interface so that it can start the new `ticker` Thread in the applet’s `start()` method. The `start()` method also sets the `boolean` variable `running` to `true` to tell the `run()` method to continue to `sleep()` and loop for as long as `running` remains `true`. When it’s time for the animation to stop, the `stop()` method sets `running` to `false` and the `run()` method exits.

If the browser calls the `start()` method again after it has stopped the applet, the `isAlive()` method returns `false` to indicate that the `ticker` Thread is no longer running. In response, the code creates a new `Thread` to restart the animation.

Your animation code needs to respect the applet’s life cycle as described in the previous paragraph; otherwise the animation can continue to run - even after the user leaves the page containing your applet - and waste CPU cycles.

The calculation `1000 / 30` inside the call to `sleep()` sets the animation rate for the applet. The `sleep()` method expects to be told how long to sleep in units of 1 millisecond. A millisecond is one 1,000th of a second, so dividing 1,000 by 30 calculates a time in milliseconds that results in the animation repeating roughly 30 times a second.

The previous code example provides the applet with a `heartbeat`, so to speak, to drive the animation. However, the sole task of the timing loop in `run()` is simply to sleep and to call `repaint()`. You need additional code to make the applet compute and display the next step, or `frame`, in the animation.
Making Things Move

The position of an image in two dimensions can be specified with the x and y coordinates of the image. In order to make the image move, you specify an additional set of x and y values that define the amount to add to the image’s original position for the next frame; think of these as \textit{delta x} and \textit{delta y} values (the Greek letter delta [\(\Delta\)] is used in math and physics to indicate the \textit{difference} between two values). You can simply add the proper values of delta x and delta y to the starting x and y position to specify motion in any direction and at any speed.

For example, say you have an image of a ball at point 1,1. If you then specify a delta x value of 1 and a delta y value of 1, the ball would move to position 2,2 for the next frame; 3,3 for the frame after that, and so on. If your delta x value is 2 and your delta y value is 2, the ball moves in the same direction, only twice as fast (or twice as far, depending on how you think about it) for each new frame.

The x and y coordinates in Java use the upper-left corner of the applet screen as the origin (0,0) and describe x and y locations in terms of pixels.

Floating the point

The best way to specify delta x and delta y values is with \textit{float} type rather than \textit{int} type values. That way, your objects aren’t limited to movement of a whole pixel per frame, nor are they limited to moving in directions that can only be expressed in terms of \textit{int} type values. Not so long ago, people used fixed point math to do fractional calculations, and many books in print still recommend this practice. However, all modern CPUs now include special floating point calculation features so that using floating point (\textit{float}) values for fractional calculations is quicker.

Encapsulating the essence v f a ba!!

Now that you understand the basics, you’re ready to write code to use the ideas presented in this chapter and create a Java class to represent a ball that can move:

```java
class Ball {
    public float x, y, dx, dy;
    private Color color;
    private int size;
}
```

(continued)
The constructor for Ball is straightforward. It simply copies the ball's initial x,y position values, dx,dy delta values, and color and size into the class variables x, y, dx, dy, color, and size, respectively.

Ball (float x, float y, float dx, float dy, int size, Color color) {
    this.x = x;
    this.y = y;
    this.dx = dx;
    this.dy = dy;
    this.color = color;
    this.size = size;
}

public void draw (Graphics g) {
    g.setColor(color);
    g.fillOval((int) x, (int) y, size, size);
}

The constructor for Ball is straightforward. It simply copies the ball's initial x,y position values, dx,dy delta values, and color and size into the class variables x, y, dx, dy, color, and size, respectively.

Ball also defines a draw ( ) method that you can call to make the ball draw itself to a Graphics context. The code is really not much more than calls to setColor ( ) and fillOval ( ), but note that the float values x and y must be cast to int in the call to fillOval ( ) to avoid a compile error. Whenever you intentionally reduce the precision of a number, you must use a cast to tell the compiler that you are doing so intentionally.

Setting Bounds

The final thing you need to add to your Ball class is code to keep the ball inside the bounds of the applet's screen area; you can add code that detects when the ball reaches one of the boundaries and then responds by reversing the appropriate delta value. Reversing either the delta x or delta y value reverses the x or y direction of the ball's movement, respectively; doing so at the boundary of the applet makes the ball appear to bounce off the boundary.

The top boundary of an applet is y=0, and the left boundary of an applet is x=0. The width and height of an applet are set by the applet's WIDTH and HEIGHT attributes in the APPLET HTML tags used to place the applet, as explained in CD Chapter 1.
Moving out of bounds

If the bouncing ball's x position becomes less than the boundary's x position (x < bounds.x), the ball just collided with the left boundary. If the ball's y position becomes less than the boundary's y position (y < bounds.y), the ball just collided with the top edge. Detecting a collision between the ball and the lower and right edges is only slightly more complicated. The right edge is computed by adding `bounds.x` to `bounds.width`. You compare this sum to the ball's x position plus its size (x + size > bounds.x + bounds.width) to check for a collision on the right side. Likewise, you compare the ball's y position plus its size to bounds.y plus bounds.height (y + size > bounds.y + bounds.height) to see if the ball collided with the bottom edge.

Bouncing back

When you detect that the ball's position has moved out of bounds, you need to reverse the sign of `dx` (if the ball collided with the left or right edges), or `dy` (if the ball collided with the top or bottom edges). Reversing the sign of `dx` or `dy` reverses the ball's movement in the given direction, thus making it appear to bounce back from the collision. However, because you can't catch a collision with a boundary until after the ball has actually moved out of bounds, you need to move the ball back in bounds to a spot that makes it appear as if it really bounced off the boundary edge.

After crossing the boundary edge, the ball wants to appear some distance beyond the edge. If the ball had actually bounced off the edge, it would have, instead, moved that same distance back in the other direction. Because a bounce is the action you actually want to create, you need to move the ball from its projected out-of-bounds position to the desired "bounced" position. You do this by moving the ball back by twice the distance it traveled out of bounds, as shown in Figure 1-1.

If the ball bounced off the left edge, this distance is 2 * (x - bounds.x). Likewise, if the ball bounced off the top edge, this distance is 2 * (y - bounds.y). When the ball bounces off the right edge, the distance is 2 * ((x + size) - (bounds.x + bounds.width)) and it's 2 * ((y + size) - (bounds.y - bounds.height)) when it bounces off the bottom edge.

Coding movement and bounce

Your next job is to take all these different collision detection and bounce handling calculations and convert them into code. The most convenient place to put this code is in a new method called `move()` that you can add to your `Ball` class. Here's the complete code for `move()`:
Figure 1-1: After detecting a collision at the top of the boundary, you move the ball back in bounds by moving it twice the distance it traveled out of bounds.

```java
public void move(Rectangle bounds) {
    // Add velocity values dx/dy to position to get ball's new position
    x += dx;
    y += dy;
    // Check for collision with left edge
    if (x < bounds.x && dx < 0)
        dx = -dx;
    x -= 2 * (x - bounds.x);
    // Check for collision with right edge
    else if ((x + size) > (bounds.x + bounds.width))
        dx = -dx;
    x -= 2 * ((x + size) - (bounds.x + bounds.width));
    // Check for collision with top edge
    if (y < bounds.y && dy < 0)
        dy = -dy;
    y -= 2 * (y - bounds.y);
    // Check for collision with bottom edge
}```
The `move()` method starts by adding the dx and dy delta values to the ball’s x and y position values to update the ball’s position. This calculation may move the ball out of bounds, so `move()` then checks the new position against the left and right bounds and then the top and bottom bounds.

You may notice that each collision test code case not only checks the ball’s position, but also checks to see if dx and dy are less than or greater than zero, depending on the case. This extra bit of checking adds a fail-safe feature to the code that prevents a ball from getting stuck should you accidentally initialize it in a position where it’s already colliding with an edge. Without this check, a ball in collision with an edge may not be able to move away from the edge before colliding with the edge again. This would cause the ball’s direction to reverse, then reverse again, on and on, forever.

To avoid this problem when checking for a collision with the left edge, the code also verifies that dx is less than zero. Similarly, the code makes sure that dy is less than zero when checking the top edge for a collision. A collision with the right edge requires that dx be greater than zero, and a bottom edge collision makes sure dy is also greater than zero. The code ignores collisions where dx or dy do not match these tests.

### Settin' things in motion

Now you’re ready to create a `Ball` object initialized with values that move and bounce it around inside the applet’s draw area. You should add the variables as class variables (inside the `Bounce` class, but not inside a method) and initialize them in the applet’s `init()` method, like this:

```java
private Rectangle bounds;
private Ball ball;
private int width, height;

public void init() {
    width = width = size().width;
    height = size().height;
    bounds = new Rectangle(width, height);
    // Initialize Ball position and velocity
    ball = new Ball(width / 3f, height / 4f, 1.5f, 2.3f, 12, Color.blue);
```

```java
else if ((y + size) > (bounds.y + bounds.height) &&
    dy = -dy;
y -= 2 * ((y + size) - (bounds.y + bounds.height));
```
The new instance of ball is saved in the variable ball. The starting position for ball is set to an x position that is 1/3 of the way in from the left boundary, and a y position that is 1/4 of the way down from the top boundary. The dx value is set to 1.5f (you add an f suffix to a number in order to create a floating point constant) and the dy to 1.3f. You can initialize the ball’s position to any value you choose, but the calculations width/3f and height/4f and the values 1.5f and 1.3f set the ball’s position inside the bounds of the applet and start it moving slowly down and to the right.

The init() method also records the applet’s size in int variables called width and height. Then, init() uses these values to create a Rectangle object named bounds. The applet later passes the bounds object to ball’s move() method, which uses it to check for collisions with the edges of the applet.

### Drawing the Details

The next step is to add code to the applet to draw ball on the screen. Also, just so things aren’t too boring, you may want to draw a simple background pattern so that you can more easily see ball move. Here’s the code for a paint() method that draws a 2 x 2 checkerboard pattern for the background, animates ball by calling move(), and draws ball into the background by calling draw().

```java
public void paint (Graphics g) f
    if (offscre == null) {
        offscreenImage = createImage(width, height)
        offscr = offscreenImage.getGraphics();
    }

    // Draw checkerboard background
    int x2 = width » 1;
    int y2 = height » 1;
    offscr.setColor(Color.gray);
    offscr.fillRect(0, 0, x2, y2);
    offscr.fillRect(x2, y2, width - x2, height - y2);
    offscr.setColor(Color.white);
    offscr.fillRect(x2, 0, width - x2, height - y2);
    offscr.fillRect(0, y2, x2, y2);
    ball.move(bounds);
    ball.draw(offscr);  
g.drawImage(offscreenImage, 0, 0, null);
```

Examine this code carefully - it includes a few new things that you may not have seen before, as the following sections explain.
Drawing offscreen

First, much of the code in `paint()` methods doesn’t draw directly to the screen. Notice that the code to draw the background pattern uses a series of `setColor()` and `fillRect()` calls that aren’t prefixed with the `Graphics` context that you normally use. Instead, the `paint()` method starts by creating an `offscreen Image`, like this:

```java
if (offscr == null) {
    offscreenImage = createImage(width, height);
    offscr = offscreenImage.getGraphics();
}
```

You draw into an offscreen `Image` so that you can construct the entire image, containing the background pattern and the image of the ball (both of which need to be redrawn each frame), before you draw it to the screen. Using an offscreen image helps reduce the flicker that results if you draw the pattern and ball directly to the screen.

Creating an offscreen `Image` that you can draw to is done in two steps: First, you create an offscreen `Image` sized the same as the applet, like this:

```java
offscreenImage = createImage(width, height);
```

After you have an offscreen `Image`, you can get a `Graphics` context for this `Image`, like this:

```java
offscr = offscreenImage.getGraphics();
```

You only need to perform this step once, so you can declare class variables at the top of the applet to hold the reference to these two objects, like this:

```java
private Image offscreenImage;
private Graphics offscr;
```

The listing for the complete applet on the CD-ROM shows where to place these two variable declarations.

Overriding the flicker

All applets have a method called `update()` that, by default, clears the screen before `paint()` is called. In many cases, you want the screen cleared before `paint()` is called, and so this default behavior can be useful. But, when you use an offscreen `Image` and then draw this offscreen `Image` to the screen, the old `Image` is erased and the new `Image` is drawn (as you can see
by the momentary screen flicker). Because you're drawing over the applet's entire visible area with the offscreen Image, the `update()` method screen clear is unnecessary.

To remove the screen clear caused by the default version of `update()`, you write a new `update()` method in your applet. Your new `update()` omits the screen clear code and, instead, simply calls `paint()`, like this:

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

**Drawing the background and the ball**

The code to draw the 2 x 2 checkerboard pattern computes two variables `x2` and `y2` that are the center points of the offscreen Image. The code then uses the values for `x2` and `y2` to draw the upper-left and lower-right sections in gray, and the upper-right and lower-left sections in white.

Next, the code calls the methods in `Ball` to move `ball` to its new position and draw `ball` onto the offscreen `Image`, like this:

```java
ball.move(bounds);
bball.draw(offscr);
```

**Putting the action on the screen**

The final step is to copy this `Image` onto the screen. Here's the code you need:

```java
g.drawImage(offscreenImage, 0, 0, null);
```

Figure 1-2 shows your completed applet in action.

The code for your completed applet, with all the details discussed in this chapter filled in, is on the *Java Game Programming For Dummies* CD-ROM.
This Chapter

Designing with state
Using the mouse
Keeping score
Creating a computer opponent

As you go through the examples and read about the techniques presented in this chapter, you may want to follow along with the complete code for the Ponglet applet included on the Java Game Programming For Dummies CD-ROM.

Setting State

Imagine that you are a robot, and your job is to perform a series of tasks that take five minutes each, but every six minutes your power is switched off, and you forget everything. However, you have a detailed book of instructions on how to do your job. Each page in the book is organized like this:

Step 1: IMPORTANT: You have only five minutes to complete this task.
Step 2: The description of the task.
Step 3: When task is complete, turn to page xx, and wait.
Each time you wake up, you are on the page that you turned to in Step 3 and you see the next task to perform (having now forgotten the wait command), and you do it. Then, by turning to the next page, you set up the next task to perform when you wake up again. The page you select serves to set the state of your brain when you wake up. Organizing a task in this fashion is called state-driven design.

The key to state-driven design is how the task is organized. The obvious difficulty for the robot is deciding how to break up a complex job into a set of tasks that can each be completed in five minutes or less. The advantage, when you break up a job this way, is that each individual task is so simple that you don't need to keep track of any other details.

**Breaking down the task**

When you play Ping-Pong, you go through a series of sequential steps. First, your opponent waits for you to get ready. Then, your opponent serves and you scramble to return the serve. Then, your opponent tries to hit your return. This process continues until one player misses. After one player misses, the score of the other player is advanced, and you both get ready for the next round. Finally, after one of you has enough points, the winner of the round is declared, and the victor gets a moment to bask in the glory.

The different steps, or states, in a game of table tennis can be described as wait, serve, return, player1 scores, player2 scores, player1 wins, player2 wins. Of course, unlike the robot example earlier in this chapter, the states of the table tennis game can appear in a variety of different sequences as the game is played.

In a computer simulation of table tennis, each state is a separate action that you need to animate, and each animated action takes a different amount of time to complete. For example, the serve state lasts until the ball travels down to where the returning player hits or misses the ball.

In the bouncing ball example in Chapter 1, the ball bounces around indefinitely or at least until you stop the applet. So the animation loop consists entirely of code to move the ball and check for collisions. However, in the case of a Ponglet game, there are some states where the ball isn't visible, such as when the ball has moved off the table. Therefore, the code to draw the ball needs to check the current game state before it draws.

One way of structuring all this is to define constants for every possible state and a variable to keep track of what state the game is currently in. Then the code in `paint( )` and the control code in `run( )` can check the current state to decide what to paint to the screen and what task the game should perform.
The code that goes in run() is going to be the most complex, so you want to think out a clean way to organize it. Using a switch statement turns out to be a nice approach. You can use the current state variable to select which case to execute. This code goes inside a while (running) loop that uses sleep() to set the animation frame rate. Here’s the complete run() framework for Ponglet:

```java
public void run () { 
    while (running) { 
        switch (gstate) { 
            case WAIT: 
            break; 
            case SERVE: 
            break; 
            case RETURN: 
            break; 
            case PGUTTER: 
            break; 
            case GGUTTER: 
            break; 
            case PSCORE: 
            break; 
            case GSCORE: 
            break; 
            case PWON: 
            case GWON: 
            break; 
            repaint(); 
            try { 
                
                tick.

            } catch (InterruptedException e) { 
        } 
    }
}
```

Note that there is no break between the PWON and GWON states because you want your code to do the same thing for both states. Therefore, when the switch statement selects the PWON state, the code will fall through to the GWON state.

The states that we discuss in the earlier example of a table tennis game are analogous to the case statements in this code. The key to dealing with these case statements is in the code that you add to complete the case statements (this code is missing here - you add it a little later in this chapter).

This code includes a few new states not mentioned in the table tennis example, such as PGUTTER and GGUTTER. The reason for these particular case statements becomes clear as you work through the sections in this chapter and fill in the missing code.
First, though, here are the definitions for the state constants and the g s t a t e variable:

```java
private static final int WAIT = 1;
private static final int SERVE = 2;
private static final int RETURN = 4;
private static final int PGUTTER = 8;
private static final int GGUTTER = 16;
private static final int PSCORE = 32;
private static final int GSCORE = 64;
private static final int PW0N = 128;
private static final int GWON = 256;
private int gstate = WAIT;
```

Note that the declaration of the variable g s t a t e initializes g s t a t e to the value WA I T. This is necessary so that the first case that is executed when the r u n ( ) method starts will be the WA I T case.

**Serving the ball**

The WA I T state is responsible for serving the ball and then setting g s t a t e to SERVE. Here’s the code you need for the WA I T case:

```java
case WAIT:
    if (!mouse in)
        delay = 20;
    else if ( delay < 0) {
        // Serve the ball
        int sLoc = rndInt(table.width - ballSize) +
                    (ballSize << 1);
        ball = new Ball(sLoc, -ballSize, rnd(5f) + 0.5f,
                        rnd(4f) + 3f, ballSize, Color.blue);
        gstate = SERVE;
        win-show = 100;
        delay = 20;
    }
    break;
```

The test if (!mouse i n) checks to see whether the player is ready to play and has moved the mouse pointer into the control area (the area of the applet’s screen that tracks the player’s mouse movements - more about this later in the section "Entering the control zone"). After the player moves the mouse to the control area, the delay value counts down, and the code in the e l s e if block serves the ball.
The value s Loc computes a random location from which to serve the ball, and this value is passed to the constructor for Ball. This code is nearly identical to the B a l l class in Chapter 1, except that it only checks for bounces off the left and right bounds. Here's the complete code:

```java
class Ball {
    public float x, y, dx, dy;
    public int size, radius;
    private Color color;
    Ball(float x, float y, float dx, float dy,
         int size, Color color) {
        this.x = x;
        this.y = y;
        this.dx = dx;
        this.dy = dy;
        this.color = color;
        this.size = size;
        radius = size >> 1;
    }

    public void move(Rectangle pd) {
        // Add velocity to position to get new position
        x += dx;
        y += dy;
        // Check for collision with bounding Rectangle
        if ((x < pd.x && dx < 0) ||
            ((x + size) > (pd.x + pd.width) && dx > 0))
            x += (dx = -dx);
    }

    public void draw(Graphics g) {
        g.setColor(color);
        g.fillOval((int) x - radius, (int) y - radius,
                    size, size);
    }
}
```

The code in WAIT that serves the ball also calls two new methods r n d i n t ( ) and r n d ( ) that generate random i n t and f l o a t values. WAIT uses these methods to generate a random velocity (speed) for the ball and to serve it from a random point along the top edge of the applet window. The r n d i n t ( n n ) method generates a random i n t (integer) between 0 and nn. The r n d ( n n ) method generates a random f l o a t (floating-point value) that is greater than or equal to 0 and less than nn. Here's how you need to write these two methods:
public float rnd (float range) I
    return (float) Math.random() * range;

public int rndint (int range) I
    return (int) (Math.random() * range);

Up java Creek without a Paddle

Okay - the ball is in motion and headed across the table toward you -
time to add code for a Paddle object that you can use to return the serve.
Here's the code:

class Paddle I
    public int x, y, width, height;
    private Color color;

    Paddle (int x, int y, int width, int height,
            Color color) I
        this.x = x;
        this.y = y;
        this.width = width;
        this.height = height;
        this.color = color;

    public void move (int x, Rectangle bd) I
        if (x > (width >> 1) && x < (bd.width - (width >> 1))]
            this.x = x;

    public int checkReturn (Ball ball, boolean plyr,
                             int r1, int r2, int r3) I
        if (plyr && ball.y > (y - ball.radius))
            if (int) Math.abs(ball.x - x) < (width l 2 +
                ball.radius) I
                ball.dy = -ball.dy;
        ! Put a little english on the ball
        ball.dx += (int) (ball.dx * Math.abs(ball.x - x) /
                         (width / 2));
        return r2;

    else
Paddle is structured similarly to the Ball class; it has values to record its x and y position, width, height, and Color. However, because Paddle doesn’t move on its own, its move() method is called by the mouse input code, as we cover in the “Entering the control zone” section a little later in this chapter.

Returning the serve

The checkReturn() may look a little complicated at first, but its main job is simply to check whether the ball hits the paddle. You use the same code to create a paddle for the player and also for the computer. The boolean parameter player is true if checkReturn() is checking the player's paddle; otherwise, it checks the computer's paddle.

The first bit of code in checkReturn() checks to see whether the ball has reached the position on the table where the ball collides with the paddle. The code for the player’s paddle is

\[ \text{ball.y > (y - ball.radius)} \]

and the code for the computer’s paddle is

\[ \text{ball.y < (y + ball.radius)} \]

If the ball is in position to be hit by the paddle, the code checks to see whether the x position of the paddle is correct to connect with the ball. The code for this check is

\[ (\text{int}) \text{Math.abs(ball.x - x)} \leq (\text{width} / 2 + \text{ball.radius}) \]

If the ball does connect with the paddle, the code needs to reverse the dy value for the ball to send it back across the table, like this:

\[ \text{ball.dy} \]
However, the game would be pretty boring if the ball simply reversed direction (dy value is reversed) and retraced its original path every time the player's paddle hit the ball. You can add code to tweak the dx value and apply a little English to the ball, like this:

```java
* \( \text{dx} \rightarrow \text{dy} \) \n* \( \text{dx} \rightarrow \text{dy} / 2 \)
```

You can play with the \( \text{width} / 2 \) value to change the feel of the paddle and how it returns the ball.

Finally, `checkReturn()` returns one of three different parameter values, \( r1 \), \( r2 \), or \( r3 \), depending on the results of its checks. `checkReturn()` returns the value that is passed in \( r1 \) if the ball hasn't yet reached the paddle, \( r2 \) if the ball reached the paddle and the paddle hit the ball, and \( r3 \) if the ball reached the paddle but the paddle missed the ball.

### Changing state

Now that you've created the `Paddle` class, you can add code to call it. You need to call it to check the player's paddle when in the `SERVE` state and to check the computer's paddle when in the `RETURN` state. Here's the code you need to add to the `SERVE` case to call `checkReturn()`. This code goes in the `SERVE` case in the switch statement because the `SERVE` case is the case the code will call when `gstate` equals `SERVE`.

```java
case SERVE:
    // Check for ball in position for player to hit
    gstate = pPaddle.checkReturn(ball, true, SERVE, RETURN, PGUTTER);
    if (gstate == RETURN)
      gPaddle = new Paddle((int) (trackX = width / 2), 3, 20, 3, Color.red);
    break;
```

When the player hits the ball, the call to `checkReturn()` sets `gstate` to `RETURN`. If the player missed the ball, `gstate` is set to `PGUTTER`. When `gstate` is set to `RETURN`, the code also creates a new `Paddle` object for the computer to use to try and return the ball.

### Creating a computer opponent

Time to create a simple computer opponent to play against. You can start out by making the computer fairly easy to beat, but you can easily tweak the program to make the computer harder to beat - it's your choice!
When the player hits the ball, the code in the SERVE case instantiates a paddle for the computer and changes gstate to RETURN. You now need to put code in the RETURN case to control the computer's paddle. Here's that code:

```java
case RETURN:
    // Implement our simple-minded computer opponent
    if (Math.abs(gPaddle.x - ball.x) >= 1)
        gPaddle.move((int) (trackX += (gPaddle.x < ball.x ? 1.5f : -1.5f)), table);
    // Check for ball in position for game to hit
    gstate = gPaddle.checkReturn(ball, false, RETURN, SERVE, GGUTTER);
    break;
```

The code for the computer opponent simply tries to move the paddle to intercept the ball. However, the computer is limited in how fast it can move the paddle by the two constants 1.5f and -1.5f. These constants are added or subtracted from the paddle position each animation tick in order to make the paddle attempt to track the ball.

Set to 1.5f and -1.5f, the paddle can only move 1.5 pixels per tick in either direction. Make these constants larger if you want your computer opponent to be able to move the paddle faster and, therefore, be a more difficult opponent.

![Rolling down the gutter](image)

In the case where the player or the computer misses the ball, you need to wait until the ball moves off the table before serving the next ball. The PGUTTER state waits for the computer's scoring ball to move off the player's side of the table. It then sets gstate to GSCORE to record the score. Here's the code:

```java
case PGUTTER:
    // Wait for computer's scoring ball to move off table
    if ((int) ball.y > (table.height + ball.radius))
        gstate = GSCORE;
    break;
```

The code for GGUTTER is nearly identical:

```java
case GGUTTER:
    // Wait for player's scoring ball to move off table
    if ((int) ball.y < (table.y - ball.radius))
        gstate = PSCORE;
    break;
```
Part 1: Steppin' Out

**He shoots, he scores!**

After scoring a point, the **PSCORE** case increments the player's score and checks whether the player has scored a total of 10 points (the criteria for declaring a winner). If the player has won, `gstate` is set to **PWON**. If the player has not yet reached a score of 10, `gstate` is set back to **WAIT** to wait to serve the next ball. Here's the code:

```java
case PSCORE:
    // Increment player's score and check if she has won
    gstate = (++pScore >= MAX_SCORE ? PWON : WAIT);
    break;
```

This code uses Java's `++` prefix increment operator to advance `pScore` to the new point before checking to see if `pScore` has reached `MAX_SCORE`. If `pScore` equals `MAX_SCORE`, the code sets `gstate` to **PWON**, else it sets `gstate` to **WAIT**.

If the computer scores, the code in the **GSCORE** case is nearly identical:

```java
case GSCORE:
    // Increment computer's score and check if it has won
    gstate = (++gScore >= MAX_SCORE ? GWON : WAIT);
    break;
```

**We have a winna!**

When the player or the computer wins the game, you need to provide time to bask in the thrill of victory. To provide this time, code in the **WAIT** case initializes a variable called `winshow` to a value of 100. The code also counts down `winshow`'s value in the code for the **PWON** and **GWON** states. The code is the same for both states, so the case statements fall through, like this:

```java
    case PWON:
    case GWON:
        // Delay while we show who won
        if (winshow < 0) 1
            gstate = WAIT;
            gScore = pScore = 0;
```
Tracking User Input

Now that you have constructed the game logic, you need to add code to handle input from the user.

Entering the control zone

First, the game draws a small area at the bottom of the applet that serves as the control area for the mouse. Moving the mouse pointer into this area causes the player's paddle to start tracking the mouse's movement to the left or right. To track when the mouse has moved into this area, you need to override the `mouseEnter()` method, like this:

```java
public boolean mouseEnter (Event evt, int x, int y) {
    pPaddle.move(x, table);
    mouse-in = true;
    return true;
}
```

The call to `pPaddle.move(x, table)` sets the position of the paddle to match the mouse's x position when `mouseEnter()` is called. The boolean variable `mouse_in` is set to `true` to indicate that the game can start. Code in the `paint()` method also checks `mouse_in` so that it knows when to draw the player's paddle.

You also need to override `mouseExit()` to provide code to set `mouse_in` to `false` in case the player moves the mouse out of the control area. This resets the `delay` counter in the WAIT state so that the game doesn't serve the ball until the player has had a chance to get ready (by moving the mouse back into the control area). Here's the code for `mouseExit()`:

```java
public boolean mouseExit (Event evt, int x, int y) {
    mouse-in = false;
    return true;
}
```

Tracking the mouse

Finally, you also need to override the `mouseMove()` method to update the position of the player's paddle whenever the mouse moves, like this:

```java
public boolean mouseMove (Event evt, int x, int y) {
    pPaddle.move(x, table);
    return true;
}
```
Now you're in the home stretch. Your final task is to add code to draw the Ping-Pong table, the ball, the paddles, the score, and the control area. You can put most of the new code into the `paint()` method for the applet.

Start with the same framework code that you used to create the bouncing ball example in Chapter 1. You can use the same code from Chapter 1 that draws to an offscreen `Image` in order to reduce flicker in the animation. You can also borrow the code that draws a checkerboard background image, or you can invent your own creative background pattern.

Here's the borrowed framework code:

```java
public void paint(Graphics g) {
    if (offscr == null) {
        offscreenImage = createImage(width, height);
        offscr = offscreenImage.getGraphics();
    }
    // Fill offscreen buffer with a background B/W checkerboard
    int x2 = table.width >> 1;
    int y2 = table.height >> 1;
    offscr.setColor(Color.gray);
    offscr.fillRect(0, 0, x2, y2);
    offscr.fillRect(x2, y2, table.width - x2, table.height - y2);
    offscr.setColor(Color.white);
    offscr.fillRect(x2, 0, table.width - x2, table.height - y2);
    offscr.fillRect(0, y2, x2, y2);
    g.drawImage(offscreenImage, 0, 0, this);
}
```

Your new code goes into the `paint()` method just before the call to `drawImage()` at the end of `paint()`.

You also need to initialize a few variables in the applet's `init()` method to create a font for displaying the score and to handle a few other details for the preceding sections. Here's the code you need to write for `init()`

```java
public void init() {
    width = size().width;
    height = size().height;
    // Set up table and mouse control area dimensions
    table = new Rectangle(width, width);
    mspad = new Dimension(width, height - width);
```
pPaddle = new Paddle(width + 1, table.height - 3, 20, Color.black);
Player = new Point(width - width / 4, 5);
game = new Point(width / 4, 5);
//Create offscreen image
offscreenImage = createImage(width, height);
offscr = offscreenImage.getGraphics();
//Setup text font for displaying the score
font = new Font(TimesRoman, Font.PLAIN, 14);
fontMet = getFontMetrics(font);
fontHeight = fontMet.getAscent();

Keeping score

Next, you can add code to paint() to draw the score, like this:

// Draw Scores
offscr.setFont(font);
centerText(offscr, game, Color.white, + gScore);
centerText(offscr, player, Color.gray, + pScore);

This code uses a new method called centerText() to center the score passed in the String str and draw it in Color clr. Here's the code for centerText():

private void centerText(Graphics g, Point loc, Color clr,
String str) {
g.setColor(clr);
g.drawString(str, loc.x - -(fontMet.stringWidth(str) / 2).
loc.y + fontHeight);
}

The Point parameter loc specifies a location for centerText() to center the score passed in the String str and draw it in Color clr. The FontMetrics object fontMet is called to compute the width of the string. The value fontHeight is added to the y value of loc so that the string is centered relative to the top center of the text.

Game over

If the game is over, you need to declare the winner. The following code displays the string "Win" beneath the winning player's score. Add this code after the code to draw the score:
Part I: Steppin’ Out

if ((gstate & (PWON | GWON)) != 0) {
    Point winner = gstate == GWON ? game : player;
    Point loc = new Point(winner.x, winner.y + 15);
    centerText(offscr, loc, Color.black, Win);
}

If the game isn’t over, you need code to draw the ball and the paddles. You can add this code to an else statement that follows the code to declare a winner:

else
    // Draw ball
    if ((gstate & (SERVE | RETURN | PGUTTER)) != 0)
        ball.draw(offscr);
    // Draw player’s paddle
    if (mouse-in || (gstate & (SERVE | RETURN | GGUTTER)) != 0)
        pPaddle.draw(offscr);
    // Draw computer’s paddle
    if (gstate == RETURN)
        gPaddle.draw(offscr);

Finally, you need to add code to draw the mouse control pad at the bottom of the screen. This code also needs to prompt the player to move the mouse into the control area to start the game, like this:

    // Fill in mouse pad area
    offscr.setColor(Color.yellow);
    offscr.fillRect(0, msePad.width, table.width, msePad.height);
    if (!mouse-in) {
        Point loc = new Point(table.width >> 1, table.height +
                            ((msePad.height - fontHeight) >> 1));
        centerText(offscr, loc, Color.black, Move Mouse Here);
    }

Figure 2-1 shows the completed Ponglet applet in action. You can find the complete code for Ponglet on the Java Game Programming For Dummies CD-ROM included with the book.
Chapter 3

Hole In One

Perhaps you play golf as a personal pastime. Or maybe you’ve putted a few holes of the miniature variety down at the family fun center. If you have, you’re in good company: Golf is a hugely popular sport in the United States, around the world, and even off the world. (Golf has the distinction of being possibly the only game played on the surface of the moon, as Alan Shepard did on February 6, 1971, during the Apollo XIV mission. Reportedly, Shepard hit his first shot about 400 yards and then badly shanked his second.)

Golf is also popular as a computer game. Dozens of versions of computer golf have appeared over the years. Some of these games present fanciful versions of the miniaturized game, some bear the names of famous players or golf courses, and some claim to be accurate simulations that model the physics and aerodynamics of real golf.

In this chapter, you won’t be tackling anything as lofty as trying to calculate the wind forces on a golf ball in flight. Instead, this chapter’s goal is to explore how to simplify the simulation of one aspect of golf, in this case putting, by faking the calculations just well enough to get a result that feels realistic. By the end of the chapter you’ll be ready to turn this knowledge into your own Hole In One applet.

This chapter describes all the techniques used in the Hole In One applet. The complete code and ready-to-use applet is included on the Java Game Programming For Dummies CD-ROM at the back of this book.
Faking physics

Don't be shocked by the idea of faking the calculations for a game. In truth, all calculations that claim to simulate a real phenomenon are really just faking it—you can never take every single variable into account (and even if you do, El Nino is just around the corner). Some calculations just happen to be less fake than others.

Your goal should be to fake (perhaps simplify sounds better) the calculations well enough to create a realistic and fungolf simulation, while avoiding needless complications.

Think of the unpredictability of a golf ball rolling across a grassy surface. Just a single blade of grass at the edge of the hole could prevent the ball from falling in. But do you really spoil the simulation if you treat the hole as a perfectly round, sharp-edged circle? Naah - it works just fine, as this chapter shows.

Modeling the Deceleration of a Ba!!

Chapters 1 and 2 show code that simulates a ball that bounces around the screen and that moves at a constant speed. In the real world, balls behave differently. For example, a golf ball starts out moving at a certain speed proportional to how hard it's hit by the putter. And immediately after the ball is hit, it starts to slow down as it travels toward the hole. This slowdown, or more formally, deceleration, is the result of a variety of forces acting on the ball, but deceleration of a golf ball is mostly caused by the rolling friction of the grass.

When real objects decelerate (or accelerate), Sir Isaac Newton's famous second law gets involved. Mr. Newton says that the deceleration of a real golf ball is proportional to the forces acting on it divided by the mass of the ball. Simulated golf balls don't have real mass, of course, and they don't have real forces acting on them either, but you do need some type of calculation to simulate Sir Newton's law in action.

The code in Chapters 1 and 2 moves the ball by adding values called dx and dy to the ball's x,y position. Therefore, you've certainly deduced that slowing the motion of the ball is going to require you to reduce the dx and dy values. Before you start working out the details, though, you may want to consider vector math, a new way to do these types of motion calculations.

using vectors

A vector (not to be confused with the java.util class of the same name) is another name for a pair of dx and dy values. You can think of a vector as representing the difference between two points; the dx and dy values form a
vector because they represent the difference between two points along the
path of the ball.

You can slow the movement of your golf ball by reducing the distance it
travels in each successive animation frame. However, the tricky part is
reducing the distance the ball travels without changing the direction it's
moving.

If you think of dx and dy as proportional to the length of two sides of a right
triangle (see Figure 3-1), the distance a ball travels when you add dx and dy
is proportional to the length of the diagonal side of the same triangle. To
compute the length of the diagonal you use the formula for the lengths of
sides of right triangles, discovered by one Mr. Pythagoras:

\[ a^2 + b^2 = c^2 \]

By rearranging this formula, you can compute the length of the diagonal as

\[ \text{magnitude} = \sqrt{dx^2 + dy^2} \]

The length of a vector's diagonal is commonly called the \text{magnitude} of the
vector.

Vector magnitude is important because it's the key to understanding how to
decelerate your golf ball. You can visualize what happens when you reduce
the magnitude of a vector by examining the relationship between the three
nested triangles in Figure 3-2.
The biggest triangle, with sides $dx_1/dy_1$, has a diagonal that is as long as the other two triangles combined. To reduce the biggest triangle so that it has a diagonal as long as the next smaller triangle, shown with sides $dx_2/dy_2$, you have to subtract the lengths of the sides of the smallest triangle, $dx_3/dy_3$, from the sides of $dx_1/dy_1$.

However, notice that you subtract more from $dx_1$ than you do from $dy_1$. If you were to subtract the same amount from $dx_1$ and $dy_1$, you would change the shape of the triangle rather than just the length of the diagonal (or more specifically, hypotenuse). To keep the shape of the triangle the same, you need to maintain the same ratio between the lengths of the sides defined by $dx_1$ and $dy_1$.

The secret to keeping the ratio of $dx_1$ and $dy_1$ the same is to divide $dx_1$ and $dy_1$ by the magnitude of the vector formed by $dx_1$ and $dy_1$:

$$dx_1 = dx_1 / \sqrt{dx_1^2 + dy_1^2}$$
$$dy_1 = dy_1 / \sqrt{dx_1^2 + dy_1^2}$$

Dividing by the magnitude changes $dx_1$ and $dy_1$ into a **unit vector** (see the following techno term icon), with values for $dx_1$ and $dy_1$ ranging from 0 to 1. The unit vector represents the direction the ball is moving, independent from the ball’s speed. And because you divide $dx_1$ and $dy_1$ by the same number, you maintain the same ratio between $dx_1$ and $dy_1$ and therefore the same direction for the ball. For example, when the ball’s movement is completely vertical ($dx_1 = 0$), the value of $dy_1$ divided by the magnitude of the vector is 1, regardless of how fast the ball is moving.
When you divide a vector’s dx and dy values by its magnitude, you get a new type of vector called a unit vector. The magnitude of a unit vector is always 1, hence the name. Unit vectors are used extensively in 2-D and 3-D graphics calculations because they can be used to represent a direction independent of speed.

You can also use a unit vector to calculate how to decelerate your ball without altering its direction. First, you calculate a unit vector from the ball’s current motion vector. Then, because the magnitude of this unit vector is 1, you can scale it to any size you want by multiplying its dx and dy values by a value that represents the magnitude of the vector you want to create. Then, you simply subtract this vector from your original motion vector, and voila! you’ve reduced the speed of the ball while keeping it moving in the same direction.

Creating a vector class

The golf simulation in this chapter uses vectors extensively to do many of the motion calculations, and you need code to perform the basic vector math operations, such as adding one vector to another, computing a unit vector, and so on. So why not bundle up all these methods into a useful new utility class called Vec2D, like this:

```java
public class Vec2D {
    public float dx, dy;

    public void setVec(float dx, float dy) {
        this.dx = dx;
        this.dy = dy;
    }

    public float mag() {
        return (float) Math.sqrt(dx * dx + dy * dy);
    }

    public void addVec(Vec2D vec) {
        dx += vec.dx;
        dy += vec.dy;
    }

    public void subVec(Vec2D vec) {
        dx -= vec.dx;
        dy -= vec.dy;
    }
}

(continued)
V\textit{e}c\textit{2}\textit{D} defines internal \(dx\) and \(dy\) values so that you can create a \textit{Vec2D} object whenever you need to keep track of a vector quantity. For example, you can create a \textit{Vec2D} object called \textit{ve1} to control the motion of your golf ball. Then, after you have this \textit{Vec2D} object, you can call the methods on it, such as \textit{setVec()} to set the direction and speed of the ball by setting the \(dx\) and \(dy\) values in \textit{vel}.

The methods \textit{addVec()} and \textit{subVec()} in \textit{Vec2D} are used to add or subtract one vector from another. You use these methods in the Golf game to apply forces to the moving ball, such as the deceleration effect of friction and the force of gravity acting to push the ball into the hole whenever the ball crosses the edge of the rim.

The method \textit{unitVec()} converts a vector into a unit vector. You can use this method in combination with \textit{mulVec()} to proportionally scale the \(dx\) and \(dy\) values in a vector by the \(float\) parameter passed to it by \textit{mulVec()}.

Notice that \textit{unitVec()} is coded to use the method \textit{mag()} that returns the magnitude of the vector.

With just these six vector operations, you can simulate all the motion dynamics needed to create a nice putting simulation. So now that you have these code elements ready to go, move on to the next section where we present the code that uses these elements to simulate the golf ball and the hole on the putting green.

**Starting gain a Circle**

Your golf simulation requires code to simulate both a ball and a hole in the putting green. The code for simulating a hole shares many things in common with the code for a ball. For example, to calculate when the ball rolls into the hole, you need code to compute the distance between the hole and the ball. So thinking along object-oriented lines, why not start by creating a common base class called \textit{Circle} to contain the code that is common to both a hole and a ball?
Creating the C i r c l e class

Your C i r c l e class needs to define x, y, and diam values to store its location and size. It also needs to have a constructor to initialize these values. And to simplify some of the calculations you need to do in your B a l l class, you need the constructor to initialize a value for the radius of the circle. Finally, you need code to compute the distance from one point to another; you can put this code into a method called dist(). Here's the complete code for C i r c l e:

class Circle {
    public float x, y;
    protected float radius;
    protected int diam;

    Circle (int x, int y, int diam) {
        this.x = x;
        this.y = y;
        radius = (float) (this.diam = diam) / 2;
    }

    protected float dist (Circle loc) {
        float xSq = loc.x - x;
        float ySq = loc.y - y;
        return (float) Math.sqrt((xSq * xSq) + (ySq * ySq));
    }
}

Building a B a l l by extending C i r c l e

Next, you can extend C i r c l e to create a new class called B a l l. You put code in B a l l essentially to do the same thing as the bouncing ball in Chapter 1, but you can use your new V e c 2 D class (see “Creating a vector class,” earlier in this chapter) to do the motion calculations. The new features in V e c 2 D also help you handle decelerating the ball as it moves. You can start by declaring the basic class, like this:

glass Ball extends Circle {
    public Vec2D vel = new Vec2D();
    public Vec2D tvec = new Vec2D();
    public boolean sunk = false;

    Ball (int x, int y, int diam) {
        super(x, y, diam);
    }
}
In addition to the constructor, this code defines two Vec2D objects: vel and tvec. The code for the Golf game uses vel to hold the ball’s dx and dy values and tvec as a temporary variable to do your deceleration calculations. You use the boolean flag sunk to keep track of when the ball falls into the hole. In addition, you use sunk in both Ball’s draw() method and in the code you write to implement the hole.

Decelerating the ball

The next job to tackle is the code to handle decelerating the ball. You use this code in two different places, so you may as well put it into its own method called decel(). decel() takes a single parameter called val that specifies the amount you want to subtract from the vector’s magnitude.

Your decel() code needs to start by checking that the magnitude of the ball’s vel vector isn’t less than val. If vel is less than val, the ball has slowed so much that if you were to further slow down the ball by subtracting the amount in val from vel, the ball would start to roll backward. To avoid having the ball roll backward, you can simply set the vel vector to zero. However, if the magnitude is greater than or equal to val (meaning that the ball has not yet rolled to a stop), you can go ahead and do the deceleration calculation, like this:

```java
public void decel(float val) {
    if (val >= vel.mag()) {
        vel.setVec(0, 0);
    } else {
        tvec.setVec(vel.dx, vel.dy);
        tvec.unitVec();
        tvec.mulVec(val);
        vel.subVec(tvec);
    }
}
```

The deceleration code starts by initializing a temporary Vec2D object called tvec. It initializes by calling setVec() and passing vel’s dx and dy values, which gives you a copy of vel in tvec.

Next, you call the unitVec() method to convert tvec into a unit vector. Then you call mulVec() to shrink this vector down to the same magnitude as val. Finally, you subtract this scaled vector from vel by calling subVec().
In addition to the constructor, this code defines two `Vec2D` objects: `vel` and `tvec`. The code for the Golf game uses `vel` to hold the ball's `dx` and `dy` values and `tvec` as a temporary variable to do your deceleration calculations. You use the boolean flag `sink` to keep track of when the ball falls into the hole. In addition, you use `sink` in both `Ball`'s `draw()` method and in the code you write to implement the hole.

### Decelerating the ball

The next job to tackle is the code to handle decelerating the ball. You use this code in two different places, so you may as well put it into its own method called `decel()`. `decel()` takes a single parameter called `val` that specifies the amount you want to subtract from the vector's magnitude.

Your `decel()` code needs to start by checking that the magnitude of the ball's `vel` vector isn't less than `val`. If `vel` is less than `val`, the ball has slowed so much that if you were to further slow down the ball by subtracting the amount in `val` from `vel`, the ball would start to roll backward. To avoid having the ball roll backward, you can simply set the `vel` vector to zero. However, if the magnitude is greater than or equal to `val` (meaning that the ball has not yet rolled to a stop), you can go ahead and do the deceleration calculation, like this:

```java
public void decel(float val)
    if Oval >= vel.mag())
        vel.setvec(0, 0);
    else f
        tvec.setVec(vel.dx, vel.dy);
        tvec.unitVec();
        tvec.mulVec(val);
        vel.subVec(tvec);
```

The deceleration code starts by initializing a temporary `Vec2D` object called `tvec`. It initializes by calling `setVec()` and passing `vel`'s `dx` and `dy` values, which gives you a copy of `vel` in `tvec`.

Next, you call the `unitVec()` method to convert `tvec` into a unit vector. Then you call `mulVec()` to shrink this vector down to the same magnitude as `val`. Finally, you subtract this scaled vector from `vel` by calling `subVec()`.
Moving the ball

Now that you've conquered deceleration, you can write the code to implement Ball's move() method. This method is the one your applet calls to advance the ball's position on each frame of the animation. Your move() method takes two parameters: The first parameter, bd, specifies the bounds of the applet. You need bd to detect when the ball needs to bounce off the edges of the applet so that it doesn't roll out of the applet's play field.

The second parameter, friction, specifies how much to decelerate the ball for each frame of animation. You use friction to call decel() in order to update the ball's deceleration, like this:

Calling decel() updates vel to account for deceleration, after which you can add vel to the ball's position to move the ball. You need to modify the ball's position in one other place in your code, so you may as well create a method in Ball for this purpose: You can call it addPos(). addPos() needs to take a Vec2D object as its input parameter. When you call addVec(), addVec() should add the dx and dy values in the vector to the ball's position, like this:

```java
public void addPos(Vec2D vel) {
    += vel.dx;
    += vel.dy;
}
```

After you code addVec(), you can add the code to move() to call it, like this:

Staying in bounds

After advancing the ball, move() must check whether the ball has moved out of bounds, which move() does by comparing the ball's position to the bd parameter. (bd, remember, is the bounds of the applet.) The code to do this comparison is nearly identical to the code in the "Bouncing back" section of Chapter 1, except that you add the decel() in order to slow the ball's movement when it bounces off an edge, like this:

```java
if (hitEdge:
    "hitEdge = (x < bd.x + radius
        (x + radius) > (bd.x + bd.width)))
```
You use the boolean flag `hitEdge` to signal that the ball has bounced off a vertical or horizontal edge. Then, you use the `decel()` method to reduce the ball's speed by 80 percent after a rebound by multiplying `vel`'s current magnitude by .8.

### Putting the bait

You also need to add code to the `Ball` to support a mouse-driven, click-and-drag putt interface. Using this interface, the player can click on the ball to select it. Then, while holding down the mouse button, the player can drag the mouse cursor back in the direction opposite from the hole, as shown in Figure 3-3. Doing so draws a rubber-band line from the mouse's current position to the ball. The player can then use this line to aim a putt. The player makes the putt by releasing the mouse button. The length of the rubber-band line when the button is released indicates the force of the putt.

**Figure 3-3:**

The golf game interface uses a rubber-band style display to control the direction and force of a putt.

#### Selecting the ball

The `Ball` class needs several methods to support this interface. First, you need a method called `touches()` to detect when the user clicks the ball:

```java
public boolean touches (int mx, int my) {
    return (new Circle(mx, my, 0)).dist(this) < radius;
}
```
When you call `touches()`, you pass it two `int` parameters called `mx` and `my` to indicate where the user clicked. You use these values to create a new `Circle` object located in the spot where the user clicked. Then you can call `Circle`'s `dist()` method to calculate the distance from this point to the center of the ball. If this distance is less than the ball's radius, `touches()` returns `true`, indicating that the user has clicked the ball.

### Freeing the Putt

To actually make a putt, your applet calls a method in `Ball` called `putt()` and passes it a `Point` object called `ptr` that indicates the location of the mouse when the mouse button is released. `Point` is a class in `java.awt`. Creating a `Point` object (in this example, `ptr`) is a convenient way to pass `x,y` values as a single parameter. Using these values, `putt()` calls `setVec()` to set the ball's `vel` variable in order to put the ball in motion. Here's the code for `putt()`:

```java
public void putt(Point ptr) {
    vel.setvec((x - ptr.y) 120, (y - ptr.y) 20);
}
```

Notice that the speed of the putt is defined as the difference between the mouse's position and the ball's position, divided by 20. Dividing by 20 provides greater resolution for aiming the putt without imparting too much force to the ball. However, you can adjust this value to suit your own preferences.

### Waiting for the Ball to Go In

You need a final method to support the putting interface: `moving()`. The Golf applet calls `moving()` in order to check the `dx` and `dy` values in `vel` and returns `true` if the ball is currently in motion. Your interface code can use this method to prevent the player from trying to select the ball while it is still in motion from the last shot. Here's the code:

```java
public boolean moving () {
    return vel.dx != 0 || vel.dy != 0;
}
```

### Drawing the Ball

The last bit of code you need to add to `Ball` is a `draw()` method. To add a nice 3-D effect, you can code `draw()` to put a shadow beneath the ball. You can create a shadow by drawing a dark gray circle offset two pixels down and to the right of the ball.
Part I: Steppin' Out

Fictitious Force?

If you tie a rock to a string and whirl it around your head you can demonstrate centrifugal force. However, centrifugal force isn't really a force at all. You are merely seeing the result of the ball's inertia as it orbits around your head. However, for the sake of convenience, you pretend that a real force is pulling on the rock. This type of pretend force is called, appropriately, a fictitious force.

Why invent a fictitious force? Well, sometimes simulating a fictitious force is easier than computing all the effects of real forces - such is the case for your golf simulation. There isn’t a real force that pulls the ball toward the center of the hole, but the calculations get a lot simpler if you pretend such a force exists.

The shadow needs to be drawn before the ball; however, you don’t want a shadow when the ball is in the hole. You can check the state of the sunk flag to see whether the ball is in the hole, and if so, not draw the shadow. Also, when the ball is in the hole, it looks better to draw it in light gray in order to simulate the darkness of looking down into a real golf hole. Here's the code for draw( ):

```java
public void draw (Graphics g)
    if (!sunk)
        g.setColor(Color.darkGray);
        g.fillOval((int) (x - radius) + 2, (int) (y - radius) + 2, diam, diam);
    
    g.setColor(sunk ? Color.lightGray : Color.white);
    g.fillOval((int) (x - radius), (int) (y - radius), diam, diam);
```

Digging a Note

Now that you have code (the Ball class) to simulate the golf ball, the next step is to create code for the hole in the form of a Hole class. Hole is a little trickier to create than Ball because the physics of how a real golf ball interacts with a real hole are far from simple. However, by using your new Vec2D code and by faking the calculations, you can get nice results without too much work.
Hole, just like Ball, extends from Circle. Inside Hole you can write a constructor that sets the position and size of the hole. Hole also requires a temporary vector for its calculations, so you can go ahead and declare a Vec2D object called tvec for this purpose. And because the draw() method for Hole is so simple, you can go ahead and write it, too:

```java
class Hole extends Circle {
    Vec2D tvec = new Vec2D();

    Hole (int x, int y, int diam) {
        super(x, y, diam);

        public void draw (Graphics g) {
            g.setColor(Color.black);
            g.fillOval((int) (x - radius),
                       (int) (y - radius), diam, diam);
        }
}
```

**Gravitating toward the center**

When the ball is sailing across the green, nowhere near the hole, the hole has no influence on the motion of the ball. However, when the ball strays close enough to the edge of the hole, gravity, using the hole as a lever, tries to push the ball into the hole. If the ball is moving fast enough and is at a sufficient distance from the hole’s center, the ball escapes the force pulling it in. If not, gravity wins, and the ball is captured, spinning around futilely at the bottom of the hole until it slows to a stop.

The force of gravity normally can only push a ball down against the grass. However, when the center of gravity of the ball is inside the radius of the hole, the force of gravity gets redirected by the lip of the hole and creates a fictitious force that seems to push the ball toward the center of the hole.

In your simulated golf game, this happens whenever the distance from the center of the ball to the center of the hole is less than the radius of the hole, as shown in Figure 3-4.

If you were to more closely simulate the forces acting on the ball, you’d have to consider that the fictitious force acting on the ball changes as the ball moves closer toward the center of the hole. This happens because the edge acts like a ramp that gets steeper and steeper as the ball topples into it.

You could add code to simulate this, but you don’t need to be this precise in order to get a realistic result. The important part is that the ball reacts as if a force is pushing it toward the center of the hole.
Vectoring in

When you animate the ball by adding the $v e_1$ vector to the ball’s position, you use $v e_1$ to simulate the force of the putter acting on the ball and the ball’s resulting momentum pushing it in a particular direction. The fictitious force pushing the ball to the center of the hole can also be represented by a vector. However, the effect of the hole doesn’t replace $v e_1$’s effect on the ball. Instead, the hole’s effect gets added to $v e_1$ and changes the direction of the force created by the ball’s momentum.

You can simulate the effect of combining two different forces by adding the vectors that represent those two forces. And, as Figure 3-5 shows, when you add two vectors, you get a new vector that represents their combined forces. The new vector can have a greater magnitude than the two vectors you add, or it can produce a vector that has an equal or lesser magnitude, depending on the values of the two vectors you add.

Curving around the hole

In the case of a fast moving ball that only grazes the edge of the hole, the fictitious force acting to push the ball into the hole is much weaker than the force of the ball’s momentum. So the combined force only manages to deflect the ball’s path. However, because the fictitious force deflects the ball in the direction of the hole’s center, the force keeps on pushing on the ball as long as it stays near the hole. As Figure 3-6 shows, even this weaker force can manage to redirect the ball’s path into one that curves around the lip.
of the hole. And this curved path may even move the ball closer to the center of the hole and wind up causing the ball to spiral in. If not, the ball's path eventually leads away from the hole, and the ball travels off in a new direction.
Part I: Steppin' Out

**Coding the curve**

Now that you've got the theory down, you're ready to start converting it into real code. You can start by adding a method to Hole called influence() in which you put the code that computes the fictitious force acting to push the ball into the hole. You also need to put code in influence() to detect when the ball has been captured by the hole, and in turn set the sunk flag to true. Also, although not strictly necessary, it's fun to add code that simulates the effect of a sunk ball bouncing around in the hole until it settles to a stop.

Your influence() code starts by computing two values to which you need to refer in several places in the code. The first value, dist, must be set to the radius of the hole minus the radius of the ball. The second value, hbdist, must be set to the distance from the center of the ball to the center of the hole; you can determine this distance by calling the Circle method dist(). These values are used to determine if the ball has strayed close enough to the hole that the fictitious force should begin acting on it.

If the hbdist is less than the radius of the hole and greater than dist, then the fictitious force should act on the ball. If hbdist is greater than the radius of the hole, then the ball's center of gravity is not inside the diameter of the hole, and the fictitious force has no effect on the ball. If hbdist is less than dist, then the ball has fallen completely into the hole and is no longer in contact with the lip of the hole, so the fictitious force should stop acting on the ball.

**Pushing to the center**

Whenever the fictitious force is acting on the ball, you can simulate its effect by computing a unit vector that points from the ball's center to the center of the hole. You can use the Vec2 object tvec to do the computation, like this:

```java
  tvec.setECec(x - ball.x, y - ball.y);
  tvec.normalize();
```

You can then multiply this unit vector by some number in order to increase or decrease the force of gravity pulling the ball into the hole, but the force of the unchanged unit vector turns out to be just about right for this simulation. So you can use adVec() to add this new vector to the ball's momentum vector vel, like this:

```java
  ball.vel.addVecf();
```
Sinking the putt

Next, you need to add the code that detects when the ball is sunk. At its heart, this check looks to see whether the distance from the ball's center to the hole's center (hbDist) is less than the radius of the hole minus the radius of the ball (distIn). But for a little extra realism, you want to first make sure that the ball isn't moving too fast to simply skip over the hole. Skipping over is what a real ball does, even if it is hit to the dead center of the hole.

You can check for a reasonable speed by comparing the magnitude of vel to the hole's radius. This comparison isn't a precise calculation, but it works reasonably well. The result of these checks set the ball's sunk flag, like this:

\[ \text{ball.sunk} = \text{ball.vel.naq()} \times \text{radius} \land \text{hbDist} < \text{distIn}; \]

The code sets the radius of the hole at 15 pixels, so if the magnitude of the ball's movement vector is greater than or equal to 15, then the ball is moving too fast to go in.

Spinning in the hole

Even after the ball drops into the hole, the ball's momentum vector still tries to make it move. A real golf ball bounces off the sides of the hole, but your Hole class doesn't yet include any code to simulate this. So unless you add more code, the Java golf ball would simply keep moving as if the hole weren't there.

The way you can simulate a ball bouncing off the inside of a circular hole is similar to the approach you use in the section "Staying in bounds" in order to make the ball bounce off the edges of the applet's display area (see Chapter 1). In effect, you wait until the ball has moved outside the bounds of the hole and then compute a new location and path for the ball that mimics the path the ball would have taken if it had bounced off the sides of a real hole.

The first step tests whether the ball has been sunk. If it has, the code needs to check whether the ball has moved beyond the bounds of the hole:

\[ \text{ball.sunk} \land \text{hbDist} > \text{distIn} \]

Next, you need to write code to go between the \{ \} brackets to calculate the position to which the ball should move after it bounces off the sides of the hole. You also need to calculate the new direction the ball will be moving after this bounce and change the ball's vel vector to make the ball move in that new direction. The calculations to do this so that the movement is modeled on the real-life behavior of a ball can get quite complex. However, because this effect is only for show, you can just fake it.
Part I: Steppin' Out

Step one in faking it is to update the ball's velocity vector \( \mathbf{v} \) to simulate a bounce off the sides. A real ball bounces off the sides of a hole on a path that is related to the angle between the point where the ball touches the side of the hole and a radial line between the center of the hole and this point. However, just calculating the point of intersection requires more math than you need to use here.

Instead, you can simply compute a vector to apply a force to the ball that pushes it back toward the center of the hole by an amount proportional to the distance the ball has strayed outside the hole. Here's the code:

```java
tvec.setVec(x - ball.x, y - ball.y);
tvec.mulVec((hbDist - distIn) / hbDist);
ball.vel.addVec(tvec);
```

You also need to update the ball's position to make it appear that it bounced off the sides. Again, you can resort to sheer fakery by simply moving the ball back toward the center of the hole by an amount proportional to how far the ball moved beyond the bounds of the hole. Here's the code:

```java
tvec.setVec(x - ball.x, y - ball.y);
float m2 = tvec.mag() - distIn;
tvec.unitVec();
tvec.mulVec(m2);
ball.addPos(tvec);
```

**Coding the Hole I n0ne Applet**

Now that you've accomplished the hard part - that of writing the code that simulates the ball and the hole - the code to complete the applet is a straightforward exercise. You mostly need to fill in the details that follow from the work you've already done in Chapters 1 and 2. For example, you need to create a `run()` method and `Thread` to handle the animation. The complete code is on the CD-ROM included with this book, so you can look there if you've forgotten any details.

**Completing the putting interface**

You still need to add the applet side of the code in order to complete the rubber-band putting interface. As discussed in the earlier section "Putting the ball," your code must use a `Point` object to record the position of the mouse and pass it to the `putX()` method in `Ball`. And you need to override the applet methods `mouseDown()`, `mouseDrag()`, and `mouseUp()` to implement the full mouse interface. Here's the complete code for these three methods:
public boolean mouseDown (Event evt, int x, int y) {
    if (ball.sunk)
        ball = new Ball(x, y, BALLSIZE);
    repaint();
    if (lball.moving() && select = ball.touches(x, y))
        putt = new Point(x, y);
    repaint();
    return true;
}

public boolean mouseUp (Event evt, int x, int y) {
    if (select)
        ball.putt(putt);
    repaint();
    select = false;
    return true;
}

public boolean mouseDrag (Event evt, int x, int y) {
    if (select)
        putt = new Point(x, y);
    repaint();
    return true;
}

You can customize the code you write for the applet's paint() method so that it draws the green in any shape you desire, but here's code that draws a simple circular green. This code also includes the code to draw the rubber-band, putt-control line:

```java
public void paint (Graphics g) {
    if (offscr == null) {
        offscreenImage = createImage(width, height);
        offscr = offscreenImage.getGraphics();
        offscr.setColor(roughColor);
        offscr.fillRect(0, 0, width, height);
        offscr.setColor(greenColor);
        offscr.fillRect(0, 0, width, height);
    }
}
```

(continued)
Part I: Steppin’ Out

(continued)

offscr.fillOval(gap / 2, gap / 2, width - gap,
   height - gap);

hole.draw(offscr);

ball.draw(offscr);

if (select)
   offscr.setColor(Color.black);

offscr.drawLine((int) ball.x, (int) ball.y,
   (int) putt.x, (int) putt.y);

eg.drawImage(offscreenImage, 0, 0, this);

The complete code for Ho 1 e In One is included on the Java Game Programming For Dummies CD-ROM at the back of this book.
The mathematics of detecting collisions

Simulating pool

Modeling billiard ball physics

Chapter 4

JavaPool

The game of billiards certainly appeals to barflies and pool hustlers still 'ein' the color of the next guy's money. It also appeals to physicists because it demonstrates, in a fun way, some of the basic laws that make the universe work. For example, when a billiard ball smacks, dead center, into another billiard ball, the moving ball comes to a complete stop. The second ball steals the first ball's momentum and travels off at nearly the same velocity as the first ball - basic physics demonstrated with elegant simplicity.

This chapter shows you how to create a simplified game of billiards in Java. However, the main point of this chapter is to introduce you to the art and science of collision detection. Because of the math involved, programmers often regard collision detection as one of the more difficult problems lurking in game design. However, the goal of this chapter is to get you past the math and down to useful techniques that you can use to get results.

This chapter also shows you how to simulate the physical laws that control how one billiard ball bounces off another. Simulating billiards and programming collision detection requires a bit of math, but don't panic; the math isn't that hard to use, even if you don't understand all the physics behind it. In the end, all equations turn back into Java code so that only your computer has to worry about them.

This chapter largely deals with the concepts you need to understand to write code that can detect and handle collisions. The full code for the applet described in this chapter is contained on the Java Game Programming For Dummies CD-ROM.
calculating 13att-to-46att collisions

Chapters 1 and 3 show you how to simulate balls that move, bounce off fixed boundaries, and fall into holes on a simulated golf green. However, simulating the interaction of billiard balls is a little more complicated because billiard balls bounce off each other, not just static boundaries or holes. This is tricky to simulate because you have to compute both the exact moment when two balls collide and the exact point at which they touch in order to simulate properly the rebound from the collision.

Passing in the night

Before you think too much about billiard balls, start by imagining two ocean liners sailing across the sea. One liner is heading in a northeast direction and the other is heading in a southeast direction. Further, the path each is traveling crosses the other's path at some distance in front of their present locations.

If both ships are the same distance from this intersection (crossing point) and if both ships are traveling at the same speed, it's obvious that the two ships arrive at the intersecting point at exactly the same time. In other words, the ships are on a collision course (man the lifeboats).

If one ship is just a ship's length closer or farther from the point of intersection, the two ships won't collide. Instead, the closer ship passes the intersection point, just as the other ship arrives at it. The passengers scream, but the ships don't collide. Likewise, if one ship travels sufficiently slower or faster than the other, the two ships don't collide because the faster ship clears the intersection point before the slower ship arrives.

In between the possibility of one ship passing the intersection point before the other arrives and a full on collision, is a tiny window of time where the slower or more distant ship reaches the intersection point before the other ship completely passes it. Exactly when the slower or more distant ship arrives determines where it hits the other ship. If it arrives at nearly the same time as the other ship arrives, it rams into the front of the other ship. If the slower, or more distant ship arrives just slightly before the other ship passes the intersection point, it clips the rear of the other ship.

Reducing the distance

As the two ships approach the point where their paths cross, the distance between the two ships gets smaller and smaller. Conversely, after one of the ships passes the intersection point, the distance between the ships starts to...
increase. All this decreasing and increasing of distance means that there must be a point in time at which the distance between the ships is as small as it's going to get.

If this distance is small enough, both ships will try to sail into the same place at the same time and means that the ships are doomed to collide. However, if this distance is large enough, both ships can pass without a collision. In between these two distances, you need to know the shape and size of each ship in order to calculate how close the ships can pass before risking a collision.

You see the obvious parallel between ships on an ocean and billiard balls on a pool table, of course. However, unlike ships, billiard balls are spheres of the same size, and calculating how close two billiard balls can get to each other without colliding is much easier than a ship shape. Because real billiard balls are never exact spheres, this distance, measured from the center of one ball to the center of the other, is just a hair larger than twice the radius of a billiard ball. However, for your pool simulation you can simplify this to simply twice the radius.

**Calculating position over time**

Imagine a billiard ball rolling across the felt surface of a real pool table at a constant speed. Then, imagine a ruler lying on the table parallel to the path of the billiard ball, as shown in Figure 4-1. At time zero, the ball is at position 1 on the diagram, and the ball is moving at a speed that carries it to position 4 one second later. Therefore, you know that in 1/4 second, the ball arrives at position 2 and in 1/2 second, the ball reaches position 3. In other words, the distance the ball travels is proportional to time.

If you know the position of the ball at any two points and you also know the time it takes for the ball to travel between those two points, you can calculate the position of the ball at any point in time. For example, suppose that you know that the coordinates for the ball when it is in position 1 are x=10 and y=20 and that the coordinates for the ball when it is at position 2 are x=12 and y=17. You know that it takes 1/4 second for the ball to travel from position 1 to position 2, and you also know that it takes 1 full second for the ball to travel from position 1 to position 4. This means that in 1/4 second, the x value of the ball's position increases by 2 and the y values, decreases by 3. In one full second, the x value increases by 8 (4 x 2) and the y value decreases by 12 (4 x 3.) Therefore, in one second, the ball reaches coordinates x=18, y=7.
Because distance is proportional to time, you can calculate the position of a ball moving at a constant speed if you know two points and the time it takes to travel between them.

When you animate a moving ball, you add \( dx \) and \( dy \) to the ball’s \( x \) and \( y \) position values at each tick of the animation. This means that in one animation tick, the ball moves from coordinate \( x, y \) to coordinate \( x + dx, y + dy \). Therefore, in three ticks of the animation clock, the ball moves to coordinate \( x + 3 \times dx, y + 3 \times dy \). If you replace a specific number of ticks with the variable \( t \) to represent any number of animation ticks, you can easily write equations for \( x \) and \( y \) to calculate the position of the ball at a new point in time \( nx, ny \) like this:

\[
\begin{align*}
\text{Calculating the distance to a collision} \\
\text{Chapter 3 covers using the relationship } a^2 + b^2 = c^2 \text{ (the Pythagorean Theorem) to calculate the distance between two points. You may recall that the distance between two points point1 and point2 is the square root of } \\
(\text{point1x - point2x})^2 + (\text{point1y - point2y})^2. \text{ This formula is called the distance formula. Now that you know how to calculate the position of a moving ball over time, you can use this formula to compute the distance between two moving balls, b1 and b2, over time.}
\end{align*}
\]
Ball bl's current position is given by bl.x, hl.y, and ball b2's current position is given by b2.x, b2.y. For each animation tick, ball bl adds bl.dx to its x position and bl.dy to its y position. Likewise, ball b2 adds bl.dx and bl.dy to its x and y position. Therefore, the x and y coordinates of ball bl at point t in time is x = bl.x + bl.dx x t, y = bl.y + bl.dy x t and the coordinate for ball b2 at the same point in time is x = b2.x + bl.dx x t, y = b2.y + bl.dy x t.

Combining the formula to compute the distance between two points and the formulas for the position of balls bl and b2 over time produces this formula that computes the distance d between ball bl and b2 at time t:

\[ d = \sqrt{(bl.x + bl.dx x t) - (b2.x + bl.dx x t))^2} + \sqrt{(bl.y + bl.dy x t) - (b2.y + bl.dy x t))^2} \]

Think of this equation as a formula for computing distance over time. It’s a rather large formula, but it’s really just an expanded form of the distance formula listed at the start of this section. The difference is that the expression (bl.x + bl.dx x t) replaces pl.x in the original formula, (b2.x + bl.dx x t) replaces p2.x, (bl.y + bl.dy x t) replaces pl.y, and (b2.y + bl.dy x t) replaces p2.y.

Using this formula, you can take any two moving balls and calculate the distance between them at any future point in time. For example, you can use this formula to see whether two balls collide in the next tick of the animation clock by computing the distance and checking whether it is less than twice the radius of the balls.

However, using the formula in this fashion isn't a foolproof solution. For example, Figure 4-2 shows the paths of two moving balls. Ball A moves from BAl to A2 in one tick of the animation clock, and ball B moves from B1 to B2. The distance between the balls at Al/B1 isn't close enough to collide, and the same is true at A2/B2. As Figure 4-2 shows, the two balls should have collided at the position shown by the dotted outlines. However, the code can fail to detect this if it only checks for collisions at fixed time intervals.

You can try to solve this "missed collision" problem by using a loop to check the distance between the two balls at points in time even closer together than a single animation tick. However, unless you were to loop until you were checking extremely tiny distances, computing the exact time one ball hits another would be difficult. You need to know the exact time to calculate the exact location of each ball when the collision happens. If you don't know the exact position of both balls at the moment of collision, you can't properly calculate the result of the collision. Even a small error in position can make a big difference in the direction and speed of the balls after the collision.
Solving for time

If you still remember any high-school algebra, you probably recall that formulas - like the one presented in the previous section for computing distance over time - can be rearranged to solve for specific values. From the position, direction, and velocity information for two objects at a specified time, the distance-over-time formula calculates the distance separating the two objects.

You already know that the only distance you care about is the distance at which two balls collide, which is twice a ball's radius. So what you want is an equation that assumes \( d = \text{radius} \times 2 \) and solves for time.

Two solutions?

Although your algebra may be a bit too rusty to figure out how to solve the distance-over-time equation for time, some quite sophisticated computer programs are available that can do it for you. One such program, Mathematica 3.0 from Wolfram Research (www.mathematica.com), takes only a few seconds to figure out the correct solution. With the solved equation, you can spend time working on your code rather than digging through your old math textbooks.

However, before you start examining this equation, you need to know that it actually has two possible solutions that solve for time when \( d = \text{radius} \times 2 \), as demonstrated in Figure 43.
In Figure 43, ball A is moving from A1 to A2, and ball B is moving from B1 to B2. As the balls approach the intersection point, the distance between the balls becomes equal to radius \( x \) at the point marked Solution 1. However, as Figure 43 shows, at another place on the path marked as Solution 2, the distance between the balls is also equal to radius \( x \). Physically, only Solution 1 makes any sense because the balls collide at that point and can never reach the position for Solution 2 unless they pass through one another. Solution 2 is a perfectly valid mathematical solution, it just doesn’t make sense for a pair of solid billiard balls.

**Rearrange the equation**

To solve the distance-over-time formula for \( t \), you first rearrange the equation into a polynomial equation of the form

\[
a x t^2 + b x t + c = 0
\]

Rearranging the distance-over-time formula into a fully expanded polynomial form produces
The distance-over-time formula from the previous section, expanded here into the general polynomial form.

\[
((b1.x - b2.x)^2 + (b1.dy - b2.dy)^2) x t^2 + 2 x ((b1.x - b2.x) x (b1.x - b2.x) x (b1.dx - b2.y) x (b1.dy - b2.dy)) x t + ((b1.x - b2.x)' + (b1.y - b2.y)' - (b1.radius + b2.radius)') = 0
\]

Note: This equation is broken into three lines to show the parts that correspond to the a, b, and c terms of the polynomial form.

You then solve this polynomial equation using the quadratic formula. The general form of the quadratic formula is

\[
t = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}
\]

The strange ± notation shows how you get two solutions to the equation. Solution 1 results when you subtract the subexpression \(\sqrt{b^2 - 4ac}\) and Solution 2 results from adding the same subexpression. Given that you know that you only want to find the first collision, you need only Solution 1.
The complete set of equations (all you really need)

The values a, b, and c in the quadratic formula are only placeholders for the real subexpressions, which you pull from the polynomial form of the distance-over-time formula. You can, in turn, reduce these equations to a simpler set by noting the repeated subexpressions in the fully expanded polynomial form. The complete set of calculations for a, b, and c, when calculated for your two balls bl and b2, is as follows:

\[
\begin{align*}
    d &= bl\text{.radius} + bl\text{.radius} \\
    ddx &= bl\text{.dx} - bl\text{.dx} \\
    ddy &= bl\text{.dy} - bl\text{.dy} \\
    distx &= b1\text{.x} - b2\text{.x} \\
    disty &= b1\text{.y} - b2\text{.y} \\
    a &= ddx' + ddy' \\
    b &= 2x(ddx' + ddy + dxddx + dyddy) \\
    c &= ddx' \cdot ddy' - d'
\end{align*}
\]

Note: The values d, ddx, ddy, distx, and disty are simply intermediate values that show how to avoid duplicate calculations in the equations that calculate the values for a, b, and c.

Then you can plug these computed values for a, b, and c into the Solution 1 version of the quadratic formula to precisely calculate the time t when two balls first collide, like this:

\[
    t = \frac{-b - \sqrt{b^2 - 4ac}}{2a}
\]

If the value of t that you compute with this formula is exactly zero, the balls are at the point of collision. If the value of t is greater than zero but less than 1, the balls collide at some point before the balls reach their respective x + dx and y + dy positions, that is, some time before the next frame in the animation.

You can also use the value of t to compute the precise positions where bl and b2 collide by multiplying each ball’s dx and dy values by t and adding the result to each ball’s x and y values, like this:
nbl.x = bl.x + bl.dx x t
nbl.y = bl.y + bl.dx x t
nbl.x = b2.x + bl.dx x t
nbl.y = b2.y + bl.dx x t

nbl.x / nbl.y represents the position where ball bl collides with ball b2, and nbl.x / nbl.y represents the position where ball b2 collides with ball bl.

Timing and order

The solution worked out in the previous sections is a great way to calculate the precise time and place where two balls collide. What happens, though, when you have more than two balls on a collision course? With two balls you only had to calculate when they would collide. With three balls you have three different ways for the balls to collide. With four balls you have six different possible collisions, and the combinations climb faster as you add more balls.

Also, when two balls collide, the collision sends the balls off in new directions. This means that your code needs to redo all your collision calculations to consider the new courses of the two balls that collided. However, instead of being a problem, this fact leads to the key idea at the center of the billiards simulation. At any given moment, you only need to calculate when the next collision is going to take place; it doesn’t matter which balls are involved. If you know the time of the next collision, you can run the motion simulation forward to that point in time, calculate the result of the collision that occurs, and then repeat the process.

You can find the first collision between a set of balls by computing the collision times for the different combinations of balls and then selecting the shortest time. Take three balls for example: balls bl and b2 can collide, balls b2 and b3 can collide, and balls bl and b3 can collide. Whichever pair of balls collides first becomes the next collision that your code needs to handle, and in the meantime, the code can proceed smoothly through the motion simulation for the balls.

Of course, all three balls can collide at the exact same time as well, which may seem to complicate things. However, arbitrarily picking one pair of balls to handle first works just fine for a game simulation because the calculations all happen so fast that the player doesn’t notice.
Checking the combinations

Finding the collision times for different combinations of balls requires a method to figure out which combinations of balls to check. The obvious approach is to use two nested loops, like this:

```java
for (int ii = 0; ii < numBalls; ii++)
  for (ii = 0; ii < numBalls; jj++)
    // check ball[ii] to ball[jj]
```

However, this approach isn't optimal. First of all, it checks for a ball colliding with itself. In addition, it checks mirror combinations, such as comparing ball [0] to ball [1] and ball [1] to ball [0]. Here's a more efficient way to arrange your loops:

```java
for (int ii = 1; ii < numBalls; ii++)
  for (ii = 0; ii < ii; jj++)
    // check ball[ii] to ball[jj]
```

By changing the first loop to start at 1 and by changing the second loop's comparison test to \( jj < i \), you create a loop that checks each combination only once.

You can see the complete code for checking all the different combinations of balls and edges in the JavaPool applet's `updateBalls()` method in the listing on the Java Game Programming For Dummies CD-ROM.

ncinq Of f the Bumpers

Chapters 1, 2, and 3 introduce a simple technique to detect and handle a collision between the ball and the applet boundary. However, the technique in those chapters isn't suitable for your pool applet because it can only detect and process collisions after they've occurred. Instead, you need a new method that works like your ball-to-ball collision code and computes the time when a ball hits an edge so that you can decide if the first collision that happens is a ball-to-ball collision or a ball-to-wall collision.

Computing when a ball hits an edge is much easier than computing when a ball hits another ball. First, a moving ball, always eventually hits an edge (unless of course, the ball slows down to a stop before reaching an edge - but more on that later in the "Putting All the Pieces Together" section). Second, the sign of the ball's dx and dy values limits which edge the ball can hit. For example, if the dx value is positive, the ball is moving to the right and can hit the right edge but can't ever hit the left edge. Likewise, if the value of dy is positive, the ball can hit the bottom edge but not the top edge.
Part 1: Steppin’ Out

After you know which edge (left, right, top, or bottom) the ball can hit, you can compute the distance to each edge and then divide by \( dx \) or \( dy \), respectively, to get the time to reach each edge. For example, the time to reach the left or right edge is the distance to the edge divided by \( dx \). Likewise, the time to reach the top or bottom edge is the distance to the edge divided by \( dy \). The next ball-to-wall collision is the one with the shorter time to collision.

Ball-to-wall collisions are different from ball-to-ball collisions, but your pool simulation code will have to watch for both types of collisions at the same time. At any given moment, the code needs to know what type of collision will happen next and how to handle it - the next section shows you how.

**Coding the Collisions**

Now you need to convert your collision math into code. You can use an approach similar to the one discussed in Chapter 3 and extend a new Ball class from the Circle class and extend Circle from the Point2D class.

The code for Ball is similar to the Ball class in Chapter 3 except that it contains several new methods to compute ball-to-ball and ball-to-wall collisions. The method to compute ball-to-ball collisions is called pathIntercept(), and it contains code that uses your new formula to compute the time one ball hits another. Here’s the code for pathIntercept():

```java
public float pathIntercept(Ball b) {
    float d = radius + b.radius;
    float ddx = vel.dx - b.vel.dx;
    float ddy = vel.dy - b.vel.dy;
    float dx = x - b.x;
    float dy = y - b.y;
    float A = ddx * ddx + ddy * ddy;
    float B = 2 * (dx * ddx + dy * ddy);
    float C = (dx * dx + dy * dy) - d * d;
    return (-B - (float) Math.sqrt(B*B - 4*A*C)) / (2*A);
}
```

The code to compute the time for a ball to collide with an edge goes into a new method called edgeIntercept(). Here’s the code:

```java
public float edgeIntercept(Rectangle bd) {
    if (vel.dx >= 0)
        hCol = (bd.width + bd.x - x - radius) / vel.dx;
    else
        hCol = (bd.x - x + radius) / vel.dx;
}
```
You need to declare $hCol$ and $vCol$ as class variables so that you can use the values to compute the new direction for a ball that bounces off an edge. Then, code a method called $bounce()$ that uses these values to compute the result of an edge bounce. Here's the code for $bounce()$

```java
public void bounce(float t)
    if (t == hCol)
        vel.dx = -vel.dx;
    if (t == vCol)
        vel.dy = -vel.dy;
```

Notice that $bounce()$ accepts a single float parameter called $t$. $bounce()$ uses $t$ to decide if the ball bounces off a left/right edge, a top/bottom edge, or both. Your code needs to calculate the time to the next collision, run the simulation forward to this point in time, and then resolve that collision. (This stuff is all covered in the "Timing and order" section, later in this chapter.)

The code only calls $bounce()$ when the next collision is a ball-to-wall collision, and so $hCol$ and $vCol$ values are set by $edgeIntercept()$ just prior to calling $bounce()$. Therefore, if $bounce()$'s $t$ parameter passes in the same time value returned by $edgeIntercept()$, you compare this time value to $hCol$ and $vCol$ to determine if the ball bounced off a left/right or top/bottom edge, or both.

After you know which edge the ball bounces off of, you handle the collision by reversing the appropriate $dx$ or $dy$ value in the ball’s $vel$ vector. For a collision on a left/right edge, you reverse $dx$, and for a collision along a top/bottom edge, you reverse $dy$. (See Chapter 1 for more details.)

**Conserving Momentum**

Handling a ball-to-ball collision is a bit more complicated than handling a ball-to-wall collision. When one ball collides with another, Newton’s law of conservation of momentum controls how each ball reacts, and you need to write code that simulates this behavior.
The momentum of a moving object is equal to the object’s mass times its velocity. When two objects collide, if you calculate the sum of the momentum of the two objects before and after the collision, the law of conservation of momentum says that you have to get the same sum in both cases (minus friction, of course). To appear realistic, your collision calculations must maintain this balance (you don’t want to break the law, do you?).

Imagine that a moving ball strikes a stationary ball and that at the point of collision, the stationary ball is exactly 45 degrees to the right of the path of the moving ball. After the collision, the previously stationary ball moves off at the 45 degree angle. Conversely, the path of the moving ball is deflected 45 degrees to the left of its original direction. The law of conservation of momentum tells you that the sum of the momentum of both balls after the collision is equal to the momentum of the moving ball before the collision. However, because you are working in two dimensions, you need to vector math to calculate the velocity of both balls after the collision.

Because billiard balls all have the same mass, you can assign all your simulated billiard balls a mass of one and greatly simplify your calculations (one times any value equals the same value). This trick means that you can use a ball’s velocity as its momentum.

**Revisiting Vectors**

Chapter 3 demonstrates that the result of adding two vectors is a new vector that combines the effects of the original two. This same principle also applies in reverse. If you add the velocity vectors for the two balls after the collision, you must get a value that exactly equals the velocity vector of the moving ball before the collision. Figure 4-4 graphically illustrates conservation of momentum by using a vector diagram to show how adding the velocity vectors for the balls after the collision produces a vector that equals the original moving ball’s velocity vector.

To determine the velocities of the two balls after the collision, you need to compute how momentum is redistributed. As Figure 4-5 shows, the transfer of momentum from the moving ball to the stationary ball is proportional to the cosine of the angle formed by a line drawn between the centers of the balls at the moment of impact and the line defined by the motion of the moving ball. Conversely, the momentum retained by the moving ball is proportional to the sine of this same angle.
After a lesion, lesion vectors of the brain usually become a ball and a sphere.
For example, in the case where the moving ball hits in the dead center of the moving ball, this angle is zero. Therefore, \( \cos(0) = 1 \) and \( \sin(0) = 0 \) and the stationary ball receives all the momentum from the moving ball and the previously moving ball comes to a complete stop. Likewise, if the mower ball barely grazes the stationary ball, the angle between the centers is near 90 degrees with respect to the path of the ball. Therefore, because the cosine of an angle close to 90 degrees is a very small number, very little of the moving ball's momentum transfers to the stationary ball.

**What if both balls are moving?**

You may be wondering how to deal with distributing momentum if both balls are moving before the collision. Easy: You can pretend that one ball is stationary simply by calculating the collision in that ball's frame of reference. For example, if you were riding in a tiny helicopter that was exactly matching the speed of one of the moving balls, a physicist would say you were in that ball's frame of reference. From your aerial perspective, the collision would appear to have happened between a moving ball and a stationary ball. Albert Einstein's theory of relativity says that the laws of physics have to look valid no matter where you observe from.

With the following approach, you convert the problem of two moving balls, ball A and ball B, into one where you always have one moving ball and one stationary ball:

1. Subtract ball A’s velocity vector from ball B.
2. Set ball A’s velocity vector to zero. This is the same as subtracting the ball A’s vector from itself.
3. Compute the collision as though ball A (with the modified vector) strikes a stationary ball A.
4. Add ball A’s original vector back to both balls.

The code to perform these calculations is fairly simple to write using a slightly improved and expanded version of the Vec2D class from Chapter 3. The main addition is a new method called `dotProd()` that calculates the *dot product* of two vectors.

**The dot product**

The *dot product* is what you get when you multiply the first vector's dx value times the second vector's dx value and then add this to the first vector's dy value times the second vector's dy value. This may seem like a strange calculation, but when both vectors are *unit vectors* (unit vectors are explained in Chapter 3), the dot product is just a fast way to compute the cosine of the angle between the two vectors.
collide() method

Use this dot product trick to write a new method called collide() for your Ball class. collide() computes the result of a collision between two balls. You call collide() by passing it a reference to a second ball. For example, to collide ball b1 with ball b2 you write:

ba.collide(b2)

Here's the code for collide():

```java
public void collide(Ball b) {
    // calculate collision in b's reference frame
    float mv = vel.subVec(b.vel).mag();
    Vec2D v12 = (new Vec2D(this, b)).unitVec();
    Vec2D vlc = vel.copy().unitVec();
    float cos = vlc.dotProd(vl2);
    vel.subVec(v12, mulVec(cos * mv)).addVec(b.vel);
    b.vel.addVec(v12);
}
```

This code is made more compact by a revision to the Vec2D class (see "Creating a vector class" in Chapter 3) which changes any method that previously returned void to instead return a reference to the same object. This trick means that you can combine several calls to successive Vec2D methods into a single statement. For example, instead of writing

```java
vel.subVec(b.vel);
float my = vel.mag();
```

you can write the more compact

```java
float my = vel.subVec(b.vel).mag();
```

collide() dissected

After you understand this new way of writing vector code, you can examine how collide() works in more detail. The first line of code

```java
float mv = vel.subVec(b.vel).mag();
```

makes the collision calculation relative to ball b's frame of reference. It does this by subtracting its velocity vector from the current ball's velocity vector. This code also calculates the magnitude of the current ball's velocity vector after subtracting ball b's vector and saves this in the variable mv.
The code then creates two new Vec2D objects: \( \text{v12} \) and \( \text{v1c} \). The code that creates \( \text{v12} \) uses a new Vec2D constructor, takes references to two Point2D objects, and creates a vector that is the difference between the two Point2D objects. The code then converts this difference to a unit vector. The following line of code accomplishes these steps:

\[
\text{Vec2D v12} = \text{(new Vec2D}(\text{this, b1?.unitVec}\text{);}\); 
\]

Next, the code creates the \( \text{v1c} \) Vec2D object by copying the current ball's vel vector and converting the copy to a unit vector, like this:

\[
\text{Vec2D vlc} = \text{vgl.copy.unitVec();} 
\]

The code then calculates the cosine of the angle between \( \text{v12} \) and \( \text{v1c} \) by computing the dot product, like this:

\[
\text{float cos} = \text{vlc.dotProd(v12);} 
\]

Next, the code sets the magnitude of \( \text{v12} \) to equal \( \text{my} \ast \cos \). Before this calculation, \( \text{v12} \) is a unit vector that points along a line from \( \text{b} \) to the current ball. Adjusting the magnitude to \( \cos \ast \text{my} \) converts \( \text{v12} \) into a vector that represents the amount of momentum to be transferred from the current ball to ball \( \text{b} \).

In the next step, the code subtracts \( \text{v12} \) from the current ball's vel vector. Doing so removes the momentum from the current ball - the same momentum that the code later transfers to ball \( \text{b} \) in the next step. Then, the code restores the current ball's original frame of reference by adding back the original, unmodified vector for ball \( \text{b} \). All these steps are accomplished in this line of code:

\[
\text{vvel.subVec( V12.mulycos, * mvj).addVec(b.vel);} 
\]

Finally, the code transfers the momentum taken from the current ball to ball \( \text{b} \) and restores its original frame of reference by adding \( \text{v12} \) to ball \( \text{b} \)'s original vel vector, like this:

\[
\text{b.vel.addVec(v12);} 
\]

---

**Putting A« the Pieces Together**

Much of the code for the Java POOL applet is copied directly from the HoleN0ne applet in Chapter 3, so there's no point in describing it again here. However, there is new code to watch for.
First, the JavaPool applet creates and maintains a list of active Ball objects using the Vector object balls. resetTable() creates four balls and adds them to the empty Vector list. It adds a white cue ball and arranges three colored balls into a triangular shape that resembles a rack of billiard balls.

The controlling code for JavaPool applet is the code in the method updateBalls(). The code in updateBalls() is based on the ideas discussed earlier in the "Timing and order" section. For each tick of the animation, updateBalls() calls each ball’s edgeIntercept() and pathIntercept() methods to see whether a collision occurs during the current animation tick. If updateBalls() finds a collision, it processes the collision by calling bounce() or collide(), depending on the type of collision - ball-to-ball or ball-to-wall, respectively - it finds.

When updateBalls() can’t find any more collisions that occur during the current animation cycle, it calls each ball’s decel() method to simulate slowing the ball’s motion due to friction. updateBalls() also checks to see whether each ball is close enough to the hole to fall in or be influenced by it by calling the Hole method influence(). (See the section "Digging a Hole" in Chapter 3 for the whole hole story.) If influence() returns true, the code knows that the ball has fallen into the hole and removes it from the list of active balls.

The interface for shooting a ball is identical to the click-and-drag interface described in Chapter 3, except that with the JavaPool interface you can select and shoot any of the four balls. The applet’s init() method also creates a pocket hole in the lower-left corner of the applet. You can move this hole to another location by changing the values passed to the constructor.

The completed JavaPool applet is shown in Figure 46.

You can find the completed JavaPool applet and the complete code for the applet on the Java Game Programming For Dummies CD-ROM.
Putting All the Pieces Together
Up to Speed
Part II
Producing a professional-quality game means mastering more than just the basics of game coding. A finished game must attend to a myriad of practical details while also serving up a heaping measure of eye appeal and style. Part II shows you how to apply spit and polish to your core game logic in order to create that professional look.

Part II also delves into the ins and outs of mazes by showing you how to create them and how to solve them. Mazes are an integral part of many games, and the maze code this part presents is used to create some of the games in both Part II and Part III.
Chapter 5

Sliding Blocks Brain Teaser

In This Chapter
Spicing up your games with images
Using the MediaTracker class
Implementing puzzle logic
Implementing a click and drag interface

In the 1870s, an American named Sam Loyd drove the world crazy with a new type of game called a Sliding Block Puzzle. Sam arranged 15 wooden tiles in a 4 x 4 grid in a small cardboard box. Because Loyd left out a tile, the box had room for you to slide one tile past another, and by a series of moves, rearrange the order of tiles.

Each tile was numbered, and the box started with tile number 1 in the upper-left corner. The sequence continued to the right and then down, but the last two numbers, 14 and 15, were reversed, as shown in Figure 5-1. Thus, Loyd called his invention the 14-15 puzzle and offered a prize of $1,000 to the first person who could solve the puzzle by putting all the numbers in sequence.

The blocks in Loyd's puzzle can be arranged in over 600 billion ways, but each rearrangement of the tiles can result only from an even number of exchanges between the blocks. Therefore, Loyd's fiendish little puzzle is impossible to solve, and the $1,000 prize, was never claimed.

In this chapter, we show you how to construct your own sliding block puzzle. This puzzle is difficult to solve, but unlike sneaky Sam's puzzle, it does have a solution. And instead of using Sam's numbers, the puzzle in this chapter has sliding blocks with colorful images.
The puzzle presented in this chapter simulates a set of wooden blocks that slide around inside a recessed rectangular area cut into a game board. You move puzzle pieces by clicking and dragging with the mouse, but the particular arrangement of the pieces (see Figure 5-2) constrains how the pieces can move. Solving the puzzle requires the player to discover the sequence of moves needed to relocate the large square piece from the top center of the board to the winning position at the bottom center.

Using Images is Games

To paraphrase an old saw: When programming games, one picture can be far cooler than a thousand fillRect() calls. Besides, modern game players expect games to have snazzy graphics, which usually means using fancy artwork rather than plain, solid colors. With modern tools, like Adobe Photoshop, you can easily create custom pictures to use in your games, even if you aren't the next Rembrandt.

The puzzle we present in this chapter uses GIF (Graphics Interchange Format - one of two graphics formats used on the World Wide Web) images to create the illusion that the puzzle is constructed from wooden blocks. These pieces slide around on a puzzle board, which is also made to resemble wood.

The puzzle uses puzzle pieces in three different sizes. If you think of the smallest pieces as 1 x 1 unit squares, the remaining two sizes of pieces are 1 x 2 and 2 x 2. These pieces slide around in a recessed rectangular area on the puzzle board. Using the 1 x 1 puzzle pieces as a unit, the size of this rectangular area is four puzzle-piece units wide by five puzzle-piece units high.
The puzzle pieces and the puzzle board shown in Figure 5-2 are all constructed in Photoshop using a third-party plug-in called PhotoTools (from Extensis Corporation) that modifies a background texture, in this case a picture of wood grain, to create the look of beveled edges similar to the effect created by the fill 3D Rect() in the Graphics class (see CD Chapter 3 for more information). This effect creates a raised look on the pieces, as if they were cut with a routing tool. The effect is reversed to create the recessed look of the rectangular area that holds the pieces. The same effect is also used on the outside edge of the puzzle board, except that the plug-in is set to create a rounded bevel.

You can easily create your own graphics to replace the files provided on the Java Game Programming For Dummies CD by using the included GIF files as templates. You need to construct 10 different piece files to replace the files piece0.gif through piece9.gif. You can also replace the game board by creating your own board.gif file. You can use almost any image editing program that can save files in the GIF format.

Using Image in Java is like having a digital stamp pad that you can use to stamp down copies of a picture onto a Graphics context. In this case, the stamp pad is an Image object, and you stamp it using the drawImage() method provided in the Graphics class. However, before you can call drawImage(), you first have to load an image file and create a Java Image object. You can create an Image object by loading files from a Web server or from your hard disk.

To create an Image object, you use a method called getImage() that is provided in the Applet class. When you call getImage(), you pass it a URL parameter (Universal Resource Locator, or more simply, Web address) that tells getImage() where to find an image file. Usually the image file is located on a Web server, but it can also come from your hard drive if you only need to run your applet on your computer. getImage() loads the data from this image file and uses it to construct an Image object.

You can also call getImage() and pass it a URL and a string that specifies the name of an image file. The string is appended to the URL to specify the exact location of the file. This form of getImage() can be conveniently used with two other Applet methods called getCodeBase() and getDocumentBase(). Calling getCodeBase() returns a URL that points to the directory on the Web server from which the applet was loaded. Calling getDocumentBase() returns a URL that points to the directory from which the HTML document that created the applet was loaded. So you can easily fetch an Image from the same directory that contains an applet's class files like this:
After you have an Image, you can draw it to a Graphics context by calling the Graphics method drawImage(). Here's an example of a simple apple that fetches an Image using getImage() and then draws the same Image with drawImage():

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

public class Example extends Applet {
    Image coffee;

    public void init() {
        coffee = getImage(getCodeBase(), coffee.gif);
    }

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

You can use getImage() to fetch an Image from a file encoded in the GIF format, but you can also call getImage() to fetch a JPEG encoded Image. The code is basically the same in either case; you just pass the name of the image file, whether it be JPEG or GIF. JPEG files let you use images that contain millions of colors, whereas GIF files have a limit of 256 colors. However, JPEG's larger color palette may sometimes be a disadvantage.

Some people may want to play your games on systems that can only display 256 colors. In this case, Java has to convert a JPEG before it can display it. This conversion process, called dithering, can produce a grainy, undesirable result. You are best off testing your games on a 256-color system to make sure that you like the result. To completely avoid dithering, you need to be careful to create your GIF files using only the 216-color browser-safe palette. (CD Chapter 4 covers the ins and outs of the browser-safe palette.)

If you use Adobe Photoshop to create your GIF files, you can convert any type of image to the 216-color browser-safe palette. If you are starting with a JPEG file and want to convert it to a GIF file, simply select Image->Mode->Indexed Color. Then in the dialog box that appears, set the palette option to Web. You can also let Photoshop predither the image by selecting something other than None for the Dither option in the same dialog box.
In this code, `drawImage()` gets passed four parameters. The first parameter is the reference to the `Image` object you want `drawImage()` to draw. The next two parameters are the x and y offset that tell `drawImage()` where to position the upper-left corner of the `Image`. In this example, the `Image` is drawn exactly in the upper-left corner at 0,0 (the origin) of the applet's screen area.

The last parameter (null) passed to `drawImage()` is used to pass a reference to an `ImageObserver`. `ImageObserver` is an interface you can implement in your applet if you want the applet to be notified of the status of an `Image` while it is downloading. However, you don't need this capability for the applet in this chapter and can simply pass `null` when you call `drawImage()`.

**Drawing while downloading**

When you try the previous applet, you may notice a strange thing, depending on how fast your browser or applet viewer loads images. When you call `getImage()`, it returns almost immediately, passing back a reference to an `Image` object. However, the `Image` hasn't actually been loaded at this point and doesn't start to load until the first time `drawImage()` is called. To make things even stranger, `drawImage()` attempts to draw an `Image` even when the `Image` isn't fully loaded.

According to the developers of Java, Java's incremental display of images as they are loaded is a "feature," not a bug. The intent is to let you duplicate the effect of a browser displaying an image while it is still loading it. However, for many game applications this can be a real nuisance.

One way to solve this problem is to use the `ImageObserver` interface, mentioned earlier in this chapter, to write code that keeps track of the status of the images you are fetching and determines when all of them have been fully loaded. An even simpler solution is to use Java's `MediaTracker` class to manage image loading and keep track of when the images are loaded, as the next section explains.

**Loading images with MediaTracker**

The `MediaTracker` class lets you construct a list of images you have requested with `getImage()`. After this list is complete, you can tell `MediaTracker` to start loading all of them and to wait until all the images have been loaded before proceeding.
The first step in using MediaTracker is to create a new MediaTracker object, like this:

    MediaTracker tracker = new MediaTracker(this);

The this parameter passed to the MediaTracker constructor is a reference to the applet that needs to use the images. MediaTracker uses this parameter to register with the code that actually loads the images.

MediaTracker.addImage()

After you have a new MediaTracker object, you can start making a list of images for MediaTracker to manage. You add an image to the list by calling MediaTracker's addImage() method. Here's an example:

    Image boardImage = getImage(getCodeBase(), board.gif);
    tracker.addImage(boardImage, board.gif, 0);

Notice that you still need to call getImage() to request the Image you want. However, you then also call addImage() to add the Image to the list of images your MediaTracker is managing.

When you call addImage(), you are required to give it an int parameter that specifies an ID value for the Image you are telling MediaTracker to track. MediaTracker supports several different methods for monitoring image loading. For example, the MediaTracker method waitForID() waits for all images that were assigned a particular ID value to load before it returns. However, it's usually easier to call the MediaTracker method waitForAll(), which doesn't return until all the images you added with addImage() are loaded. If you use waitForAll(), it doesn't matter what ID value you pass to addImage().

MediaTracker.waitForAll()

After you add all the images to your MediaTracker object, you call the method waitForAll() to start loading the images and wait for loading to finish. This call doesn't return until all the images you added with addImage() are fully loaded and ready to use. However, the waitForAll() method has one more detail you have to handle.

The waitForAll() method specifies that it can throw an InterruptedException. This exception isn't currently implemented in Java 1.0.2, but you still need to provide a try/catch block so that the code compiles properly, like this:

    try {
        tracker.waitForAll();
    } catch (InterruptedException e) { // Do something
    }
Loading multiple images

The sliding blocks puzzle has 11 different images that you need to load - 10 puzzle pieces plus the puzzle board. The image for the puzzle board is called board.gif; the images for the puzzle pieces are named piece0.gif through piece9.gif. You can simplify the code to load the puzzle pieces by taking advantage of the sequential naming of the puzzle piece files. Code a loop and then use string concatenation (fancy lingo meaning to combine multiple strings into one) to create the name of each file from the loop counter variable. For example, if you have an int variable called ii and the current value of ii is zero, the following code creates the string piece0.gif:

```java
String frame = piece + ii + gif;
```

Using this string concatenation trick, here is the code you need to load the puzzle board and all 10 puzzle pieces:

```java
MediaTracker tracker = new MediaTracker(this);
boardImage = getImage(getCodeBase(), board.gif);
tracker.addImage(boardImage, 0);

for (int ii = 0; ii < 10; ii++) {
    pieceImages[ii] = getImage(getCodeBase(), piece + ii + gif);
    tracker.addImage(pieceImages[ii], 0);
}

try {
    tracker.waitForAll();
} catch (InterruptedException e) {
}
```

You can put this code in the init() method of the puzzle applet. You also need to declare boardImage and the pieces[] array as class variables, like this:

```java
private Image offscreenImage, boardImage;
private Image[] pieceImages = new Image[10];
```

Lr~r~iaq Out the Game t3oard

The design of the puzzle board, shown in Figure 5-3, aligns all the puzzle pieces onto an invisible 4 x 5 grid to make calculating how to draw each puzzle piece onto the board as easy as possible. The origin (upper-left corner) of the grid is offset from the applet's origin by the values specified in the variables gridX and gridY. The width and height of a grid square is specified by the variables pieceWidth and pieceHeight.
The grid squares are arranged such that grid square $x=0$, $y=0$ is the upper-left square, and $x=3$, $y=4$ is the lower-right square, as shown in Figure 5-3. When you draw a puzzle piece onto the board, you calculate the upper-left corner for any grid square from these four variables, $gridX$, $gridY$, $pieceWidth$, and $pieceHeight$, using the following formulas:

\[
\begin{align*}
\text{imageX} &= gridX \times pieceWidth + gridX \\
\text{imageY} &= gridY \times pieceHeight + gridY
\end{align*}
\]

However, you can't apply these formulas until you know the proper values to assign to $gridX$, $gridY$, $pieceWidth$, and $pieceHeight$. You can write the code to initialize these values from constants, but this means you can't change the size of the puzzle graphics without recompiling the code. Instead, because the size of the grid square is equal to the size of the smallest puzzle piece, you can initialize $pieceWidth$ and $pieceHeight$ by reading the width and height of one of the puzzle piece images.
Chapter 5: Sliding Blocks Brain Teaser

Reading the width and height of an Image

The Image class contains methods called `getWidth()` and `getHeight()` that determine the width and height of an Image. For example, to read the width and height of the game board `boardImage`, use the following code:

```java
int width = boardImage.getWidth(null);
int height = boardImage.getHeight(null);
```

Due to the design of the Java AWT, you can't reliably read an Image's width and height until the Image has been fully loaded. This isn't a problem when you use `MediaTracker` to wait for an Image to fully load before you use it - a very good reason to always use `MediaTracker` to load your images.

Initializing gridX, gridY, pieceWidth, and pieceHeight

Using `getWidth()` and `getHeight()` lets you initialize `pieceWidth` and `pieceHeight` by reading the width and height of one of the small puzzle piece images. The earlier section "Loading multiple images" shows code that loads all the puzzle piece image files and saves references to the `Image` objects in an `Image[]` array called `pieceImages`. `pieceImages[51]` contains a reference to the image file `piece5.gif`, which is one of the small puzzle pieces. Read the width and height of `pieceImages[51]` to initialize `pieceWidth` and `pieceHeight`, like this:

```java
int pieceWidth = pieceImages[51].getWidth(null);
int pieceHeight = pieceImages[51].getHeight(null);
```

You still haven't initialized `gridX` or `gridY`, but you can easily calculate these values from the width and height of the game board image `boardImage` and `pieceWidth` and `pieceHeight`. Because the grid is centered in the middle of the game board, you calculate `gridX` by subtracting the width of the grid from the width of the game board `boardImage` and then divide the result by two. The width of the grid is `4 x pieceWidth`, so you calculate `gridX` like this:

```java
int gridX = (boardImage.getWidth(null) - (pieceWidth * 4)) / 2;
```

You calculate `gridY` using nearly identical code:

```java
int gridY = (boardImage.getHeight(null) - (pieceHeight * 5)) / 2;
```
Part II: Up to Speed

Craftinly the Puzzle

To make the puzzle work, you need to construct a class to encapsulate the logic that handles each individual puzzle piece. This new class, called `Piece`, needs to contain a constructor to instantiate (create and define) the pieces needed by the puzzle, code to draw the image that represents the piece on the board, and code to handle sliding the piece from place to place on the board.

Making puzzle pieces that act like real puzzle pieces

One of the trickiest aspects of coding the puzzle is designing the logic that makes the puzzle pieces act like real puzzle pieces placed on a real board. For example, when you try to slide a puzzle in the direction of an adjacent piece, the adjacent piece either blocks the first piece from moving, or, if the piece is able to slide in the same direction, is pushed along.

To accomplish this, each puzzle piece needs to know its size and current position and be able to check the size and position of the other pieces. In addition, you need to have some way to monitor when one piece pushes another piece.

Thankfully, the Java AWT includes a built-in class called `Rectangle` that is designed to represent a movable rectangular area. `Rectangle` includes code that can check whether one rectangular area overlaps or intersects another rectangular area and greatly simplifies your task of creating the logic of sliding puzzle pieces. `Rectangle`'s built-in features make it the perfect superclass for your new `Piece` class.

Starting with `Rectangle`, you can easily code the beginnings of your new `Piece` class, including code to draw the image of the puzzle piece, like this:

```java
class Piece extends Rectangle {
    private Image pic;
    Piece (int bx, int by, Rectangle grid, Image img) {
        super (bx * grid.width + grid.x,
               by * grid.height + grid.y,
               img.getWidth(null), img.getHeight(null));
        pic = img;
    }
}
```
The first two parameters to `Piece`, `bx` and `by`, specify the location of the piece on the invisible 4 x 5 grid shown in Figure 5-3. For example, to position a piece in the upper-left position on the grid, specify `bx = 0` and `by = 0`.

Even though you specify the location of a piece on the grid, the piece can't calculate where to draw the `Image` that represents the piece unless it knows the values you computed for `gridX`, `gridY`, `pieceWidth`, and `phyt`. You could pass these parameters to the constructor for `Piece` in four `int` parameters. However, it's easier to create a `Rectangle` object from these four values and pass a reference to this object to `Piece` in a parameter called `grid`.

The last parameter for `Piece`, `img`, is a reference to the `Image` that represents the piece on the board. The constructor saves this reference in the variable `pic`, which is used by `Piece`'s `draw()` method. The constructor calls `img.getWidth()` and `img.getHeight()` to determine the size of the `Image` and passes this to the superclass constructor, along with the pixel position. Therefore, after a piece is instantiated, `Piece`'s `x` and `y` values record the pixel position of the piece on the board, and the `Piece`'s `width` and `height` values give the real size of the piece, in pixels.

`Piece`'s `draw()` method, as shown in the code in this section, uses the `Image` variable `pic` and the `x` and `y` pixel position values to draw the image that represents the piece onto the game board.

### Putting the pieces together

Now that you can instantiate a puzzle piece, you're ready to finish the applet's `init()` method by adding the code to instantiate all the pieces for the puzzle. Here's the completed code for `init()` along with all the needed class variables for the puzzle applet:

```java
public class Puzzle extends Applet {
    private Image offscreenImage, boardImage;
    private Image[] pieceImages = new Image[10];
    private Graphics offscr;
    private Piece[] pieces = new Piece[10];
    private Piece picked = null;
    private Rectangle grid, clickArea;
    private Font bbCourier = new Font( Courier , Font.BOLD , 48 );
    
    public void draw(Graphics g) {
        g.drawImage(pic, x, y, null);
    }

    // Completing the init() method
    public void init() {
        // Instantiate all puzzle pieces...
    }

    // Piece class methods...
}
```

(continued)
The puzzle pieces are instantiated inside the last for loop. The values in the pcs[] array define the starting position for each piece. Notice also how grid is initialized using the values computed for grid X, grid Y, piece Width, and pieceHeight.
The puzzle uses an offscreen Image to draw the puzzle graphics, asset up by the calls to createImage() and getGraphics(). (See the section "Drawing offscreen" in Chapter 1 for more on creating offscreen Images.) However, notice the call to repaint() in the last line of init(). In many examples, you don't need this call, but because MediaTracker can take some time to load all the images and because Java is multithreaded (see CD Chapter 2), repaint() is almost certainly called before the graphics are loaded. So you need to call repaint() to schedule another call to paint() after the images are loaded so that they actually appear onscreen.

The Rectangle object clickArea is created for use in the interface, as described in the next section.

Using the Pieces Around

To make your puzzle easy to play (or playable at all, for that matter), you need to create an interface. Probably the easiest way to come up with an interface is to create simple code that lets the user click a piece and then, while holding the mouse button down, drag the piece to a new position.

Selecting a puzzle piece

The first step in implementing your user interface, that of detecting when the user has clicked on a piece to select it, is implemented by overriding the mouseDown( ) method, like this:

```java
public boolean mouseDown(Event evt, int x, int y) {
    if (clickArea.inside(x, y)) {
        for (int ii = 0; ii < pieces.length; ii++) {
            if (pieces[ii].inside(x, y)) {
                picked = pieces[ii];
                selectedPiece = new Point(x, y);
                break;
            }
        }
    }
    return true;
}
```

This code first checks whether the user clicks the mouse inside the puzzle board's recessed area. The applet's inito method, described in the earlier section "Putting the pieces together," creates a Rectangle called clickArea that defines the boundaries of the area where the pieces can move. The Rectangle class has a method called inside0 that returns true if a point, defined by its two x and y parameters, is inside the rectangle.
Next, the code uses a `for` loop to iterate through the list of pieces and, again, uses the `inside()` method to check whether the user clicks the mouse inside this piece. If the mouse is clicked inside a piece, the reference for that `Piece` is copied to the `picked` variable, and the location where the user clicked is recorded by creating the `Point` object `selectedPiece` and passing the x and y location of the click to the `Point` constructor.

A `Point` is an AWT class that holds the values of an x,y pair.

**Moving the pieces**

Next, you override the `mouseDrag()` method so that you can track the movement of the mouse while the mouse button is held down. However, you need to make sure that you only try to track the mouse when a puzzle piece is selected. So the code starts by testing to see if `picked != null` and `selectedPiece != null`.

Your users can drag the mouse in any direction, including diagonally, but the code to slide the pieces is simpler if you constrain the mouse to moving either vertically or horizontally. You can divide the code into separate sections to handle movement along each axis like this:

```java
public boolean mouseDrag(Event evt, int x, int y) {
    if (picked != null && selectedPiece != null) {
        int dx, dy;
        while ((dx = limit(y - selectedPiece.x)) != 0 &&
            picked.slide(pieces, dx, 0, clickArea)) {
            selectedPiece.translate(dx, 0);
        }
        while ((dy = limit(y - selectedPiece.y)) != 0 &&
            picked.slide(pieces, 0, dy, clickArea)) {
            selectedPiece.translate(0, dy);
        }
        repaint();
        return true;
    }
    return false;
}
```

Taking the horizontal movement first, the code calls a small helper method called `limit()` that constrains the movement to stepping by a single pixel, or not at all, by limiting the movement to ±1 or zero. Here’s the code for `limit`:

```java
private int limit(int val) {
    return (val < 0 ? -1 : 0);
}
```
Chapter 5: Sliding Blocks Brain Teaser

If you're not familiar with it, let's use of the conditional operator(?) in the return statement may look strange. However, this is just a more compact way of writing:

```
if (val > 0) {
  return 1;
} else if (val == 0) {
  return -1;
} else {
  return 0;
}
```

Inside `mouseDrag()`, the code computes the difference between the original location where the user clicked the mouse to select the piece, which is held in the `Point` value `selectedPiece`, and the current position of the mouse, which is passed to `mouseDrag()` in the x and y parameters. Then, `mouseDrag()` uses two `while` loops to move the piece, one pixel at a time, until the position of the piece matches the current location of the mouse. The first `while` loop handles moving the piece left or right, and the second `while` loop controls movement up or down.

The `while` loops also have to handle one other important detail: As the user drags the piece around on the game board, the piece the user is moving may bump into another piece. If the piece does bump into another piece, the code needs to check if the piece that the selected piece bumped into blocks further movement of the selected piece or if the selected piece can push the blocking piece out of the way. The check to determine if the selected piece moves a blocking piece or is blocked by it is determined by a new method called `slide()`.

**Slide C)ing around**

Writing `slide()` is the trickiest bit of code in this book so far, but it isn't that hard to write if you break the problem down into simpler steps. The key is to leverage several of the methods provided in `Rectangle`.

**Checking for pieces that block the slide path**

With `Rectangle.intersects()`

Given two rectangles, the `Rectangle` method `intersects()` returns `true` if one `Rectangle` overlaps or `intersects` the other. You can use `intersects()` to determine whether the player is trying to move one puzzle piece on top of another by creating a new `Rectangle` that holds the new position for the
piece (the one that indicates the move the player intends to make) and then coding a loop that checks this new position against all the other puzzle pieces in the game. If the new position $\text{Rectangle}$ intersects with any other puzzle piece, the piece that the player wants to move is potentially blocked from moving.

Of course, you also need to check whether the puzzle piece that is potentially blocking the player's move is itself able to slide in the same direction. You can check this potential movement by allowing $\text{Piece}$ to recursively call itself and attempt to move the blocking puzzle piece. Then, if the blocking piece is able to move, you allow the original piece to move as well. The neat trick about the way the code makes this check recursively is that it allows a blocking puzzle piece, in turn, to move a puzzle piece that blocks it, and so on - as though the code walks down the line and checks all the pieces in a given direction to see whether they can move.

**Checking for the board boundaries** $\text{Rectangle.union()}$ and $\text{Rectangle.equals()}$

After the code lets the player move a piece, it also needs to make sure that the move doesn't push the piece beyond the bounds of the playing area on the board. The $\text{Rectangle}$ class provides a simple way to check for the piece's position in the form of two additional methods: $\text{union()}$ and $\text{equals()}$.

The $\text{union()}$ method starts with two $\text{Rectangles}$ and creates a new $\text{Rectangle}$ that is as large as the smallest $\text{Rectangle}$ you can draw that would contain both the original rectangles. The $\text{equals()}$ method compares two $\text{Rectangles}$ and returns $\text{true}$ if both $\text{Rectangles}$ describe exactly the same rectangular area. Using these two methods, you can easily check whether a piece is trying to move outside the bounds of the playing area, as shown in the following code where $\text{bd}$ is the $\text{Rectangle}$ that describes the bounds of the board and $\text{np}$ is a $\text{Rectangle}$ object that describes the new position for the piece:

$$\text{if (bd.union(np).equals(bd))}$$

---

**Recursion**

When a method calls itself, that is a recursive call. Writing recursive code is not something you do every day, but there are some types of calculations that are easier to accomplish if you use recursion. The classic example is using recursion to compute factorials, like this:

```java
int factorial (int n) {
    if (n > 1) {
        return n * factorial(n - 1);
    } else {
        return 1;
    }
}
```
Chapter 5: Sliding Blocks Brain Teaser

This statement is false only if rectangle np goes outside the bounds of bd and causes bd. union(np) to return a Rectangle larger than bd. Note that np is created by adding dx and dy to the current position of the piece. The variables x, y, width, and height are inherited from the Rectangle class from which Piece extends.

Now that you understand the basic approach, here’s the complete code for slide():

```java
public boolean slide(Piece[] pp, int dx, int dy,
    Rectangle bd)
    Rectangle np = new Rectangle(x + dx, y + dy,
        width, height);
    for (int jj = 0; jj < pp.length; jj++)
        if (this != pp[jj] && pp[jj].intersects(np))
            if (pp[jj].slide(pp, dx, dy, bd))
                continue;
        return false;
    if (bd.union(np).equals(bd))
        translate(dx, dy);
    return true;
    return false;
```

The final step in moving a puzzle piece is the call to translate(), which moves the position of the Piece to the same location that your code just checked (as we discuss earlier in this section). Notice that the code only allows translate() to happen when all the other checks have passed, the puzzle piece has space to move, and the move keeps the piece on the board.

Cleaning up after a move

One last step is “cleaning up” after the player releases the mouse button. To keep the layout of the puzzle pieces nice and tidy, and also to simplify the check for solving the puzzle, you need to add code to slide the pieces to the closest grid position when the player releases the mouse button after a move. You can put this code in mouseUp() like this:
The code in `mouseUp()` uses a loop to adjust every puzzle piece on the board by getting the reference to each piece from the `pieces` array and calling a new `Piece` method called `snap()`. You need to add code for `snap()` to the `Piece` class, like this:

```java
public void snap(Rectangle grid)
    move((x - grid.x + grid.width / 2) / grid.width)
    grid.width + grid.x,
    (y - grid.y + grid.height / 2) / grid.height)
    grid.height + grid.y;
```

`snap()` calculates the grid position closest to the current location of a piece and calls the `Rectangle` method `move()` to move the piece to this location. The `Rectangle` parameter `grid` passes in the values for `gridX` and `gridY` (described in the section "Laying Out the Game Board") in `grid.x` and `grid.y` and the width and height of a grid square in `grid.width` and `grid.height`.

The calculation `(x - grid.x + grid.width / 2)` computes the position of the center of the piece on the x axis and the calculation `(y - grid.y + grid.height / 2)` does the same on the y axis. Then, dividing by `grid.width` and `grid.height`, respectively, computes the closest position in grid coordinates (take a look back at Figure 5-3). Finally, multiplying the x axis value by `grid.width` and adding `grid.x` (gridX) converts the piece's horizontal position on the grid back to a pixel position on the game board. Likewise, multiplying by `grid.height` and adding `grid.y` (gridY) converts the vertical grid position back to a pixel position.

**Drawing the Board**

You're best off drawing the board to an offscreen `Image` and then copying this `Image` to the screen to get smooth screen updates (see "Drawing offscreen" in Chapter I for more info). Draw the board first and then draw the puzzle pieces on top of the board with a loop to call each piece's `draw()` method:
public void paint (Graphics g)  
  if (offscr != null)  
    offscr.drawImage(boardlmage, 0, 0, null);  
    for (int ii = 0; ii < pieces.length; ii++)  
      aces[ii].draw(offscr);  
    g.drawImage(offscreenlmage, 0, 0, this);  

And when drawing to an offscreen image, you need to always override update() to remove the flicker that it can cause:

public void update (Graphics g)  
  paint(g);

---

**Declaring the Puzzle Sawed and Congratutating the Winner**

Finally, you need to add code to determine when the player has solved the puzzle and to display an appropriate congratulatory message. As we cover in the "Putting pieces together" section earlier in this chapter, you use the init() method to create a Point object called winLocation that contains the upper-left pixel coordinates for a 2 x 2 square in the winning location. winLocation thus allows your code to check for a solved puzzle simply by checking whether the 2 x 2 sized puzzle piece (which is the first entry in the pieces[] array) has moved to the same location as winLocation, like this:

    if (pieces[Cl].x == winLocation.x &&  
      pieces[Ol].y == winLocation.y)

Next, you need to display a nice simple message like "Win!" to herald the success of the player. You can display such a message by adapting the code from CD Chapter 3 that displays centered text. However, to create a nice effect and help make the text stand out against the somewhat busy background of the puzzle board, you may also want to put a shadow beneath the text. You can add a shadow by drawing the text twice: First, set the draw color to black and draw the text offset down and to the right by a few pixels. Then, set the color of the text, in this case yellow, and draw the text at the centered position.
This new code, along with the \texttt{winLocation} check, is best added to the \texttt{paint()} method just after the \texttt{for} loop that draws the pieces. Here's the code:

```java
if (pieces[0].x == winLocation.x && pieces[0].y == winLocation.y) {
    FontMetrics fm = offscr.getFontMetrics(bbCourier);
    offscr.setFont(bbCourier);
    int strHyt = fm.getAscent();
    int OOffset = (size().width - fm.stringWidth(winMsg)) / 2;
    int OOffset = (size().height - strHyt) / 2 + strHyt;
    offscr.setColor(Color.black);
    offscr.drawString(winMsg, Offset + 3, OOffset + 3);
    offscr.setColor(Color.yellow);
    offscr.drawString(winMsg, Offset, OOffset);
}
```

The solved puzzle is shown in Figure 5-4. Have fun trying to solve the puzzle yourself!

The complete code for the Sliding Blocks puzzle is on the \textit{Java Game Programming For Dummies} CD-ROM included with this book.
In This Chapter

- Creating a complete game of Blackjack
- Programming the fundamentals of card games
- Extracting graphics from composite images
- Creating a user interface using AWT components
- Placing AWT components on the screen

Card games make fantastic Java games: They are relatively easy to program and are great for playing on a computer, whether by single or multiple players. You can create Java card games from any of the popular card games you're already familiar with, or you can invent your own.

All most card games require is a deck of cards and a playing surface. In this chapter, we present a complete game of Blackjack to demonstrate a reusable deck of cards, and show how to create a playing surface using the standard component classes from the Abstract Window Toolkit (AWT). We also show you how to use the button and text components, and how to arrange various components on the screen using the AWT's LayoutManager classes; both of these techniques are applicable to many types of games.

Understanding the Blackjack Game

Blackjack (also called Twenty-One) is the most popular casino card game in the United States. The Blackjack dealer plays against one or more players, and in the casino version of Blackjack, the house rules strictly regulate the dealer's options for play. The dealer's predictable behavior makes Blackjack an excellent choice for a computer card game: You can make the computer play the role of dealer and easily program a strict set of dealer actions.
"lacing Blackjack

The object in Blackjack is to end up with a hand that scores higher than the dealer's hand without going over 21. A hand's score is the sum of the value of all the cards in the hand, using the following point values for the cards:

- Face cards (Jack, Queen, and King) are each worth 10 points.
- 2 through 10 are worth points equal to the face value of the card.
- Ace is worth 11 points, except when the addition of 11 points makes the player's hand total more than 21 points, in which case the Ace is worth 1 point.

Each player places a bet to begin play. Next the dealer deals one card face-down to everyone at the table, including herself. Then the dealer deals a second card face-down to each player and a second card face-up to herself. Each player then has an opportunity to receive additional cards dealt face-up one at a time (be hit) until either the cards in his hand exceed 21 points (the hand busts) or he declines additional cards (stand). After all players receive any additional cards they want, if any, the dealer exposes her face-down card and then takes cards according to the house rules - typically hitting the hand until it busts or totals at least 17 points.

The player loses his bet if he busts (goes over 21) or if the dealer's hand totals more than the player's hand. The dealer returns the bet if the player and dealer have hands with the same point totals and pays the player an amount equal to the bet if the dealer busts or the player's hand is higher than the dealer's hand.

In addition, Blackjack has a few special situations:

Blackjack's Special Situations

**Blackjack** A Blackjack, or natural, is a hand that totals 21 points after the first two cards (an Ace counting as 11 points and a 10 or a face card). If a player has a natural and the dealer doesn't, the player wins 1 1/2 times the initial bet, and gets back his original bet as well. A natural beats a non-natural 21.

**Double Down** After the first two cards are dealt, a player may opt to double down, which means the player doubles his bet and takes a single additional card. Some casinos restrict doubling down to when the player's hand totals 10 or 11.
A player may split a hand consisting of two cards with the same point value into two hands, thereby doubling the chance for a win (or a loss, of course). The player puts the two hands side by side on the table and places a bet on the new hand equal to the original bet. The player then plays each hand normally, except that if a split hand totals 21 after two cards, it doesn't count as a natural.

**Designing the game**

The first step in designing an object-oriented program (Remember: Java is an object-oriented programming language) is to look at the objects being modeled by the program. Figure 6-1 shows the objects in a real game of Blackjack.
After you identify the objects your game needs to model, you need to organize the objects according to their relationships and functions in the game. Although programming challenges usually have more than one solution, the elements typically suggest certain logical relationships that, in turn, suggest the most elegant way to program them. The Blackjack game presented in this chapter organizes the elements of Blackjack, as shown in Figure 6-2.

As Figure 6-2 shows, the Blackjack Applet provides the playing surface and contains an array of player objects, a dealer hand object, a deck object, and a user interface. The deck and dealer hand objects each contain an array of card objects. Each player object contains a player hand object and a bankroll. The player hand contains an array of card objects and a bet. The user interface is a panel component that contains the button objects for the various options available to the player. The fact that the object relationships for Blackjack look similar to the layout of the real-world "objects" is no coincidence: The game design is derived from the real-world objects.

Creating a Reusable Deck of Cards

A deck of cards is a common element to all card games (bet you didn't know that), and many of the techniques presented in this section can be applied to any card game you want to program. You can even create specialized decks (for example, Pinochle or Poker with the Joker added) with minor changes here and there, as we explain a bit later in this chapter.

One of the powerful aspects of object-oriented programming is how it facilitates reusing code. This section shows how to create a reusable Deck class that you can use in any card game that uses a standard deck. This section also presents the Card objects that the Deck class shuffles and deals.
Shuffling and dealing the deck

The `Deck` class uses an array to keep track of the cards in the deck. Rather than actually remove cards from the array as they are dealt, the `Deck` class maintains an index to the last card in the card array and deals cards from the end of the array. As each card is dealt, the index to the last card in the array is decremented. The complete `Deck` class is:

```java
import java.util.NoSuchElementException;
public class Deck {
    protected Card[] cards;
    protected int top;
    protected int packs;

    public int getSize() { return top; } // If undealt cards
    public int getPacks() { return packs; } // Packs in deck
    public int packSize() { return 52; } // Cards in pack
    public Deck() { this(1); }
    public Deck(int packs) {
        if ((this.packs = packs) > 0)
            top = packs * packSize();
        cards = new Card[top];
        reshuffle();
    }

    // Fills the deck with cards and shuffles
    public synchronized void reshuffle() {
        top = 0;
        for (int packs = this.packs; --packs >= 0; )
            for (int suit = Card.CLUB; suit <= Card.SPADE; suit++)
                for (int rank = Card.ACE; rank <= Card.KING; rank++)
                    cards[top++] = new Card(rank, suit);
        shuffle();
    }

    // Shuffles the undealt cards in the deck
    public synchronized void shuffle() {
        if (top > 1)
            for (int ii = top; --ii >= 0; )
                int rnd = (int) (Math.random() * top);
            Card temp = cards[ii];
            cards[ii] = cards[rnd];
            cards[rnd] = temp;
    }
}
```

(continued)
Part I: Up to Speed

(continued)

// Deals the top, card from the deck
public Card deal () throws NoSuchElementException I
    return deal(0);

// Deals a card from the deck. <pos> is relative to the top of the deck (0 = top, 1 = second from top, etc.)
public synchronized Card deal (int pos)
    throws NoSuchElementException
{
    Card c = peek(pos);
    if (pos > 0) //deal card from middle of deck
        System.arraycopy(cards, top - pos,
                        cards, top - pos - 1, pos);
    top--;    
    return c;
}

// Returns the top card from the deck without dealing it
public Card peek () throws NoSuchElementException I
    return peek(0);

// Returns a card from the deck without dealing it.
// <pos> is relative to the top of the deck (0 = top)
public synchronized Card peek -(int pos)
    throws NoSuchElementException
{
    if (POs < 0)
        throw new NoSuchElementException();
    try I return cards[top - (pos + 1)];
    catch (IndexOutOfBoundsException e) {
        throw new NoSuchElementException();
    }
}

// end class Deck

The algorithm used in Deck. shuffle() works by swapping each card in the deck with another randomly selected card. This swapping randomizes the deck after a single pass through the deck and is much simpler than simulating a real shuffle.
Chapter 6: Blackjack

The `deal()` and `peek()` methods fetch cards from the deck. `deal()` removes the fetched cards from the deck, but `peek()` does not. Both `deal()` and `peek()` are overloaded methods, meaning that they are given alternate versions with different parameters. One version operates on the top card in the deck; the other version accepts a parameter specifying the card position. The Blackjack applet doesn’t use `peek()` or the ability to deal from the middle of the deck; neither is necessary for a functioning Blackjack game. However, other computer card games, like Solitaire, use these methods to cycle through the deck multiple times and play cards from arbitrary positions.

Notice that instead of defining its own subclass of `Exception` to indicate when a requested card is unavailable, `Deck` uses the standard `java.util.NoSuchElementException` class included as part of the Java API. Using this standard class eliminates your having to write and the player’s web browser having to download extra code to define a class to handle unavailable cards.

Some card games, including Blackjack, use a deck built from combining several packs of cards. The `Deck` class supports multipack decks by overloading the `Deck()` constructor to accept the number of packs to use.

Building the `Card` class

Each card must remember just two things: its suit (Club, Diamond, Heart, Spade) and its `rank` (Ace, 2, 3, 4, 5, 6, 7, 8, 9, 10, Jack, Queen, King). You can give cards the following capabilities to make working with them as easy as possible:

- **Assign the suits and ranks to static final variables.**
- **Override `Object.toString()` so that a card can display its name.**
- **Have cards draw themselves to a specified `Graphics` context.**
- **Override `Object.equals()` so that cards can compare themselves.**

The resulting `Card` class is

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

public class Card {
    public static final int CLUB = 0, DIAMOND = 1, HEART = 2, SPADE = 3;
    ...
}
```

(continued)
public static final int JOKER = 0, ACE = 1, TWO = 2, THREE = 3, FOUR = A,
FIVE = 5, SIX = 6, SEVEN = 7, EIGHT = 8, NINE = 9,
TEN = 10, JACK = 11, QUEEN = 12, KING = 13;

private static final String[] suitNames =
{ Club, Diamond, Heart, Spade };

private static final String[] rankNames =
{ Joker, Ace, Two, Three, Four, Five,
Six, Seven, Eight, Nine, Ten, Jack,
Queen, King };

private static Image cardsImage;

private static int cardWidth, cardHeight; // in pixels

private int rank;
private int suit;

public final int getRank () { return rank; }
public final int getSuit () { return suit; }

public Card (int rank, int suit) {
if ((this.rank = rank) != JOKER)
this.suit = suit;
}

public final String getSuitName ()
return rank == JOKER ? suitNames[suit];

public final String getPluralSuitName () {
return rank == JOKER ? suitNames[suit] + s;
}

public final String getRankName ()
return rankNames[rank];

public final String getPluralRankName ()
return rankNames[rank] + (rank == SIX ? es : s);

public final String getArticle ()
return rank == ACE || rank == EIGHT ? an : a;

public final boolean isRed ()
return suit == DIAMOND || suit == HEART;

public final boolean isFace ()
return rank >= JACK;
Chapter 6: Blackjack

public static int getCardWidth ()
{ return cardWidth;

public static int getCardHeight ()
{ return cardHeight;

/* If not currently loaded, then loads and inits the
  cards.gif image. This image is laid out as a 14 wide
  * by 4 tall grid of card images laid out as:
  * JOKER 1 2 3 4 5 6 7 8 9 10 J Q K (CLUBS)
  * CARDBACK A 2 3 4 5 6 7 8 9 10 J Q K (DIAMONDS)
  * BLANK A 2 3 4 5 6 7 8 9 10 J Q K (HEARTS)
  * BLANK A 2 3 4 5 6 7 8 9 10 J Q K (SPADES)

public static synchronized void initGraphics(Applet app)
if (cardsImage == null) {
  MediaTracker tracker = new MediaTracker(app);
  cardsImage = app.getImage(app.getCodeBase(),
  cards.gif);
  tracker.addImage(cardsImage, 0);
  try { tracker.waitForAll(); } 
  catch (InterruptedException e) {}  
  cardWidth = cardsImage.getWidth(app) / 14;
  cardHeight = cardsImage.getHeight(app) / 4;

public static void drawCardBack (Graphics g, int x, int y) {
// card back is second card down in first column
doDraw(g, x, y, 0, cardHeight);

private static void doDraw (Graphics g, int x, int y,
  int xaff, int yoff) {
  if (cardsImage != null) {
    Graphics gcopy = g.create();
    gcopy.clipRect(x, y, cardWidth, cardHeight);
    gcopy.drawImage(cardsImage, x - xaff, y - yoff, null);
    gcopy.dispose();

public void draw (Graphics g, int x, int y)
doDraw(g, x, y, rank * cardWidth, suit * cardHeight);

(continued)
public String toString ()
    StringBuffer buf = new StringBuffer(rankNames[rankl);
    if (rank != JOKER)
        buf.append( of ).append(getPluralSuitName(-));
    return buf.toString();

public boolean equals (Object obj) {
    if (obj instanceof Card) f
        Card c = (Card)obj;
        return crank == rank && c.suit == suit;
    return false;

public int hashCode () t
    return (rank << 2) + suit;
}

The Card class supports Jokers even though Blackjack does not use a Joker. Card is a general-purpose class useful for all kinds of games, including those that use a Joker.

Converting cards to strings

Most computer games use an image of a card to represent a card on screen, but sometimes cards need to be displayed as text. For example, if you are trying to develop and debug a game, you may want the cards to appear as text so that you can work out the process of the game before creating the images for the cards.

To convert cards to strings, override Object.toString() so that it implicitly converts an object to a string when using the + string concatenation operator. A Card object has methods for returning the plural and singular names of its suit and rank. Card uses these methods in toString() to construct and return the name of the card.

For example, the following code displays The card is an Ace of Spades:

Card c = new Card(Card.ACE, Card.SPADE);
System.out.println( The card is + c.getArticlet) +
Overriding equals()

Notice that Card overrides both Object.equals() and Object.hashCode(). The Vector,Hashtable, and other data structure classes use equals(). A Vector is like an array that automatically grows as new objects are added to it. A Hashtable is a vector where the objects are located by using a key object, such as a string, instead of an index. Hashtables use a hash code (an integer) calculated by calling the hashCode() method for the key object to find the object in the hash table. You need to maintain certain relationships between equals() and hashCode() in order for an object to work correctly with these data structures. While the objects work fine with the array used by Deck, the games that use the Card class may place Card objects in Vectors or Hashtables. So whenever you override equals() or hashCode(), you must make sure that the following expressions remain true:

\[
\begin{align*}
\text{a.equals(b)} & = \text{b.equals(a)} \\
\text{a.hashCode()} & = \text{b.hashCode()}
\end{align*}
\]

In addition, both equals() and hashCode() need to return consistent results throughout the lifetime of the object. The data structures use equals() and hashCode() to organize and find the objects placed in the data structure. If the result of equals() or hashCode() changes, the object could become "lost" in the data structure.

You use immutable (unchangeable) attributes to calculate the hash code and perform the equality comparison consistently. For example, the Card class uses the suit and rank attributes in both equals() and hashCode() but never changes their values after the Card() constructor runs.

Extracting card graphics from a composite image

Card contains several methods that load and display card graphics. Instead of loading 54 individual images (52 cards plus a joker and a card back), the methods in Card use a composite image (a single large image that contains all the individual images) and then extract a piece of the composite image to draw the particular card to the screen, as shown in Figure 6-3. Think of the composite image as a quilt of card "patches" from which the code extracts the necessary patch to display.

A single large image downloads faster, requires less memory to store, and is easier to create and edit than multiple small images. It also keeps all the images for a deck of cards encapsulated in a single object, and that fits well with an object-oriented design.
Part II: Up to Speed

Figure 6-3: The "cards.gif" composite image.

Four rows of 14 card images - one row for each suit - combine to form the composite image. You call the static method initGraphics() to load the combined image. initGraphics() uses the MediaTracker class discussed in Chapter 5 to load the cardsImage image. initGraphics() calculates the width (cardWidth) of an individual card by dividing the composite image's width by 14 and the height (cardHeight) by dividing the composite's height by 4.

To draw a single card rather than the entire cardsImage graphic, you need to define a clipping region to restrict where the graphics context draws pixels. To understand how a clipping region works, imagine that the graphics context is a piece of paper. To create a clipping region take another piece of paper the same size, cut a rectangular hole in it, and position it over the first piece of paper. Now when you draw on the graphics context, you can only draw within the clipping region. To draw a card from the cardsImage graphic, you make the clipping region the size of a single card, then position cardsImage to line up the desired card with the clipping region and call drawImage() to draw the card. Figure 6-4 shows what this clipping image looks like if cardsImage is a third piece of paper and you place it between the other two sheets of paper.

The draw() and drawCardBack() methods use cardWidth and cardHeight to calculate the x,y pixel coordinates of the upper-left corner of the image to extract from cardsImage. These methods pass the x,y pixel coordinates to doDraw(), which uses them as the offsets for positioning cardsImage so that the card to be drawn is in the correct place relative to the clipping region. doDraw() sets the clipping region and draws the card using the following code:

```java
Graphics gcopy = g.create();
gcopy.clipRect(x, y, cardWidth, cardHeight);
gcopy.drawImage(cardsImage, x - xoff, y - yoff, null);
gcopy.dispose();
```
The `clipRect()` call sets the upper-left corner of the clipping region to the specified x and y coordinates and sets the width and height of the region to the dimensions of a single card. `drawImage()` accepts the position for the upper-left corner of `cardsImage()`. You calculate the position of `cardsImage` relative to the clipping region by subtracting the x and y offsets within `cardsImage(xoff and yoff)` of the upper-left corner of the card to draw.

Setting the clipping region of a graphics context is an irreversible operation. Each subsequent call to `clipRect()` sets the clipping region to the intersection of the current clipping region and the new region, which means that you can never enlarge the clipping region. In order to keep from permanently setting the main clipping region of the component, the preceding code from `doDraw()` obtains a temporary copy of the graphics context, sets the clipping region in the copy, and performs the draw operations using the copy. The code obtains a copy of the graphics context by calling the context's `create()` method. After it finishes with the temporary copy, the code calls its `dispose()` method so that the Java Virtual Machine (JVM) can reclaim its memory.

Java 1.1 adds the `setClip()` method to the `Graphics` class and allows the clipping area to be more flexibly resized. You don't need to make a copy of the graphics context before using `setClip()`. However, until Java 1.1 becomes more widely used, use `clipRect()` for maximum portability.

**Customizing the deck**

You can give the cards in the deck custom graphics by creating a new `cards.gif` image. The `Card` class automatically calculates the size of the cards based on the image.
In addition, you can extend the Deck class to play Poker with a joker in the deck or to play games such as Pinochle that use a nonstandard deck of cards. You override `packSize()` and `reshuffle()` to initialize the custom deck. A Pinochle deck has 48 cards - two of each suit of the ranks nine through ace. You implement a Pinochle deck like this:

```java
class PinochleDeck extends Deck {
    public PinochleDeck () {
        super();
    }
    public PinochleDeck (int packs) {
        super(packs);
    }
    public int packSize () {
        return 48;
    }
    public synchronized void reshuffle () {
        top = 0;
        for (int packs = this.packs * 2; --packs >= 0; )
            for (int suit = Card.CLUB; suit <= Card.SPADE; suit++)
                for (int rank = Card.NINE; rank <= Card.KING; rank++)
                    cards[top++] = new Card(rank, suit);
                cards[top++] = new Card(Card.ACE, suit);
        shuffle();
    }
}
```

### Creating a User Interface with Components

You create a user interface in Java from AWT components. Component is an abstract class in the java.awt package that embodies all the common functions of user interface elements. You build your interface from the subclasses of Component: Button, Canvas, Checkbox, Choice, Label, List, Scrollbar, TextArea, and TextField.

### Using buttons

Buttons are a common interface element. You use buttons to represent an action the user can perform. Unlike menus, which also represent available actions, a button is

- Visible, so the user doesn't have to look through menus to determine what actions are available.
- Convenient, because it requires a single mouse click to perform the associated action.
You use the `Button` class in the `java.awt` package to add buttons to your game.

### Creating and placing buttons

In order to create a button, you invoke the `Button()` constructor and pass it a string to use as the button label. The button automatically adjusts its size to fit the label. Optionally, you can omit the string in the constructor call and call the button's `setLabel()` method to set the label string.

```java
Button dealButton = new Button(Deal);
Button cancel = new Button();
cancel.setLabel(Cancel + operation);
```

After you create a button, you must call `add()` to add the button to a `Container` component—a `Panel`, `Applet`, `Window`, `Frame`, or `Dialog`—in order to use it. After adding components to a container, you need to call the container's `layout()` or `validate()` method to tell the container to arrange its components and set their sizes. `layout()` arranges the current container; `validate()` arranges the current container as well as any containers inside the current container.

### Component peers

Don't usually subclass `Button` or most components to create custom versions. `Canvas` and the container components—`Panel`, `Applet`, `Window`, `Frame`, and `FileDialog`—are designed for!/* extended. The reason is that each AWT applet has a platform-specific `component` peer, which is a class that connects an `component` such as a button to the platform's native implementation. The peers then draw the component and handle the component events. So for example, Microsoft Windows implements the `Button Peer` using a Windows button, and Apple Computer's MacOS implements the `Button Peer` using a Mac button, and so on. The component peer feature is why Java buttons on a Mac look different than Java buttons on a PC. The two figures below show the same applet (Colored Applet) running in Netscape Navigator 3.0, one in Windows 95, a PC, and one on a Mac running OS 8.
Having your game respond to buttons

Button converts a mouse click into an ACTION-EVENT. You override Component.action() in the parent container in order to handle action events. You identify the button that generated the event in action() by comparing the Event.target field to the Button object, alternately, by comparing the Event.arg field to the button label string.

Here is an example of an applet creating and responding to a button:

```java
import java.awt.*;
public class ColoredApplet extends java.applet.Applet
private Button red = new Button("Red");
private Button blue = new Button("Blue");
public void init()
    add(red);
    add(blue);
    layout();

public boolean action(Event evt, Object arg)
    if (evt.target == red)
        setBackground(Color.red);
    else if (Blue.equals(arg))
        setBackground(Color.blue);
    repaint();
    return true;

// end class ColoredApplet
```

Notice that the action() method in ColoredApplet uses the evt.target field to find the red button and uses the arg parameter to identify the blue button. We include both techniques in the example just to show both ways of checking. However, the evt.target way is faster, because comparing object references is less work than comparing strings.

Reading and displaying text

At some point, most programs need to display text or collect information entered by the user. Java 1.0 provides three components for displaying text: TextField, TextArea, and Label. The Blackjack applet uses labels to display each player’s name and current bankroll. TextField and TextArea also accept text typed by the user. TextField displays and accepts a single line of text; TextArea creates a scrollable, editable text area that displays multiple lines. This section shows how to use each of these components.
Displaying status and scores with labels
You use a Label component to display text in a container. The program code can change the text in a label, but the user cannot edit the text on screen. You can use labels to display things such as game scores or to create, well ... labels such as those used to describe on-screen buttons or other objects. You create a label and add it to a container like this:

```java
import java.awt.*;
public class ScoreLabel extends java.applet.Applet {
    private int score;
    private Label scoreDisplay;

    public void init () {
        add(new Label(Score));
        add(scoreDisplay = new Label(0, Label.RIGHT));
        layout();
    }

    public boolean mouseDown(Event evt, int x, int y) {
        scoreDisplay.setText(Integer.toString(score += x));
        return true;
    }
}
```

This example initially sets the scoreDisplay label to a right-aligned string with several leading zeros, as shown in Figure 6-5. The spaces cause the label to automatically set its size to be larger than the space required for just the 0 character. The RIGHT alignment positions the label text from the right edge of the label so that additional digits appear on the left. You align a label to the LEFT, CENTER, or RIGHT - the default alignment is LEFT.

Getting a few words from the user
Use a TextField component to collect a single line of text from the user. As with other components, you add a text field to a Container component. You override action() in the container to detect when the user enters text.
Text Field is a subclass of TextComponent. A Text Component supports the normal cut, copy, paste, and text edit functions of the underlying operating system (OS). Table 6-1 lists some of the TextComponent methods for working with text.

**Table 6-1**

<table>
<thead>
<tr>
<th>Method</th>
<th>Use This Method to</th>
</tr>
</thead>
<tbody>
<tr>
<td>getText()</td>
<td>Retrieve the current text from the component.</td>
</tr>
<tr>
<td>setText(str)</td>
<td>Set the value of the component’s text to the string str.</td>
</tr>
<tr>
<td>getSelectedText()</td>
<td>Retrieve the currently selected text in the component.</td>
</tr>
<tr>
<td>select(start, end)</td>
<td>Set the component’s selection to the characters from offset start to offset end, inclusive.</td>
</tr>
<tr>
<td>setEditable(edit)</td>
<td>If the boolean parameter edit is false, user editing is disabled for the component; otherwise editing is enabled.</td>
</tr>
</tbody>
</table>

The offset of a character is the position in a string of that character, starting from 0 and counting up. For example, the letter “n” has an offset of 3 in the word “Barney”.

**Creating scrolling text areas**

You use a Text Area to create editable and scrollable text displays. As is Text Field, TextArea is a subclass of TextComponent and has all the text editing features listed in Table 6-1. TextArea also has the three additional methods for working with the text, listed in Table 6-2.

**Table 6-2**

<table>
<thead>
<tr>
<th>Method</th>
<th>Use This Method to</th>
</tr>
</thead>
<tbody>
<tr>
<td>appendText(str)</td>
<td>Append the string str to the end of the area’s current text.</td>
</tr>
<tr>
<td>insertText(str, pos)</td>
<td>Insert the string str at the character offset pos.</td>
</tr>
<tr>
<td>replaceText(str, start, end)</td>
<td>Replace the text from character offset start to offset end, inclusive, with the string str.</td>
</tr>
</tbody>
</table>

Additional TextArea Methods
Chapter 6: Blackjack

The following example uses a Label, TextField, and TextArea:

```java
import java.awt.*;
pUBLIC class ListEntry extends java.applet.Applet I
    static final String newline =
        System.getProperty(line.separator);
    private TextArea list = new TextArea(5, 20);
    private TextField entry = new TextField(20);

    public void init () I
        list.setEditable(false):
        add(list);
        add(new Label(Name:. Label.RIGHT));
        add(entry);

    public boolean action (Event evt, Object arg)
        if (evt.target == entry) I
            list.appendText(entry.getText() + newline):
            entry.selectAll();
            return true;
        return false;
I 11 end class ListEntry

ListEntry creates a TextArea that holds five lines of approximately 20 characters each and a TextField that holds a single line of approximately 20 characters. The AWT calls ListEntry.action() when the user presses the Enter or Return key in the entry text field. action() appends the entry text to the list text area and selects the text in entry so that the next text that the user types replaces the current text. Figure 6-6 shows the program running.

App
George Washington
Thomas Jefferson
Roosevelt Lincoln
Teddy Roosevelt
Ronald Reagan

Name: Ronald Reagan
As Ut sorted
Part II: Up to Speed

The `System.getProperty(line.separator)` call in the preceding example gets the system-dependent character string used to separate lines text. You would think that operating systems would standardize on something as simple and basic as a line separator, but they don’t. Under UNIX-based operating systems, the separator is a linefeed character (`\n`); under Mac OS, the separator is a carriage return character (`\r`); and under DOS and Windows operating systems, the separator is a carriage return followed by a linefeed (`\r\n`).

**Using Canvas to create new components**

A `Canvas` component is a way to create your own component - it provides all the hooks to collect user events and draw itself, but the default canvas is basically a blank slate. You can use `Canvas` to create your game’s playfield, displays, and just about anything else that you want to display a certain way.

**Customizing your game’s appearance With `ImageButton`**

The AWT components are functional but not particularly colorful. You can design your own custom components by subclassing the `Canvas` component. The simplest and probably the most useful component to customize is `Button`. The following `ImageButton` class uses two images—the pressed and unpressed button images - rather than the standard button with a text label:

```java
import java.awt.*;
public class ImageButton extends Canvas implements Runnable {
    private ThreadGroup tg;
    private Image[] img = new Image[2]; // up = 0, down = 1
    private int imgndx; // index into img[1]

    public ImageButton(Image up, Image down) {
        tg = Thread.currentThread().getThreadGroup();
        img[0] = up;
        img[1] = down;
    }

    public synchronized Dimension minimumSize () {
        int x = Math.max(img[0].getWidth(null), img[1].getWidth(null));
        int y = Math.max(img[0].getHeight(null), img[1].getHeight(null));
        return new Dimension(x < 0 ? 10 : x, y < 0 ? 10 : y);
    }
}
```
public Dimension preferredSize () {
    return minimumSize();
}

public void paint (Graphics g) {
    if (img[imgndx] != null)
        g.drawImage(img[imgndx], 0, 0, this);
}

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

public synchronized void run () {
    imgndx = l;
    repaint();
    Component p = getParent();
    if (p != null)
        p.postEvent(new Event(this, Event.ACTION_EVENT,
            img[0]1));
    try {
        Thread.sleep(200); // press for 1/5 second
    } catch (InterruptedException e) {
    }
    imgndx = 0;
    repaint();
}

public synchronized boolean mouseDown (Event evt, int x, int y) {
    (new Thread(tg, this)).start();
    return true;
}

The following are the key points for implementing ImageButton:

- ImageButton overrides Component. preferredSize() and Component. minimumSize() in order to have the button automatically size itself to the size of the largest image.
- mouseDown() spawns a thread to animate the button press so that it doesn't perform the animation in the AWT Interface thread. (CD Chapter 2 explains what the AWT Interface thread is and how to spawn threads from mouseDown () and other event handlers.)
- mouseDown() and run() are synchronized to prevent the animation from being interrupted. Synchronization prevents mouse clicks from being processed faster than the animation rate.
— run() passes an ACT I ON-EVENT to the parent container’s post Event() method so that you handle ImageButton events in the same way as Button events. Because an ImageButton doesn’t have a label string, it uses the ”up” image for the Event. arg field.

— The button animation is state-driven; run() sets the img ndx variaY to change the state, and paint() uses img ndx to display the current state.

Java 1.2 has something called lightweight components that support custom appearances without having to re-create the entire functionality of the component.

Displaying a hand of cards

The Blackjack computer game extends Canvas to create a displayable hand of cards, as shown in Figure 6-7.

The BlackjackHand class includes both the code for displaying the hand and for handling the details of a Blackjack hand - dealing cards to the hand, calculating the value of the hand, testing for a Blackjack (natural), and so on. The BlackjackHand class is

```java
import java.awt.*;
import java.util.*;

class BlackjackHand extends Canvas
{
    protected int horizInset;
    protected boolean soft;
    protected Vector hand = new Vector(6);
    private boolean exposed, active;
    private Image offscreenImage;
    private Graphics offsccr;

    public Dimension preferredSize ()
    {
        return new Dimension((Card.getCardWidth() + horizInset)* 2, Card.getCardHeight());
    }
```
BlackjackHand ()
    resize(preferredSize());

int cardCount ()
    return hand.size();

protected boolean isDealer ()
    return true;

boolean blackjack ()
    return hand.size() == 2 && value() == 21;

boolean isSoft ()
    value(); // sets <soft> field
    return soft;

void setActive (boolean on)
    active = on;
    repaint();

void expose ()
    exposed = true;
    repaint();

void clearHand ()
    hand.removeAllElements();
    exposed = active = soft = false;
    repaint();
    Thread.yield();

void deal (Card card)
    hand.addElement(card);
    repaint();

public void paint (Graphics g)
    if (offscreenImage == null) {
        offscreenImage = createImage(size().width,
                                      size().height);
        offscr = offscreenImage.getGraphics();
    }

(continued)
offscr.setColor(Active ? Color.yellow : Color.gray);
offscr.fillRect(0, 0, size().width, size().height);
int handsize = hand.size();
if (handsize > 0)
    int overlap = Math.min(Card.getCardWidth(),
        size().width - Card.getCardWidth() -
        2*horizinset) / Math.max(1, handsize-1));
Enumeration deal = hand.elements();
'int xoff = horizInset;
while (deal.hasMoreElements())
    Card card = (Card)deal.nextElement();
    if (!exposed && isDealer() && xoff == horizInset)
        Card.drawCardBack(offscr, xoff, 0);
    else
        card.draw(offscr, xoff, 0);
    xoff += overlap;
}
} //end class BlackjackHand

public void update (Graphics g) { paint(g); }
} //returns the point value for the hand
int value () { int val = 0;
    boolean ace = false;
    for (int ii = hand.size(); --ii >= 0; ) :
        int v = value(ii);
        Val += v;
        if (v == Card.ACE) ace = true;
    if (soft = (val <= 11 && ace))
        Val += 10;
    return val;
}

ten return base point value of card <cardNum> in the hand
protected int value (int cardNum) {
    int rank = ((Card)hand.elementAt(cardNum)).getRank();
    return Math.min(Card.TEN, rank); // face cards are 10's
} // end class BlackjackHand
Here are the key elements in Blackjack:

- BlackjackHand extends java.awt.Canvas so that it can display itself on the screen.
- The BlackjackHand constructor sets the size of the canvas - necessary because by default a canvas is zero pixels wide and zero pixels tall.
- The java.awt.Vector field hand stores the cards. (Remember, a Vector works like an array that automatically resizes itself as more objects are added to it) Using a vector allows a hand to accept any number of cards dealt to it. deal() adds cards to the vector and clearHand() removes all the cards from the vector.
- BlackjackHand overrides paint() to create the display for the hand. If necessary, paint() overlaps the cards to fit within the display area - this makes the cards look like they are "fanned out" on the table, as shown back in Figure 6-1.
- paint() uses the graphics context offscreenImage. offscreenImage.paint() copies the offscreen image to the screen to smoothly draw the hand to the screen in a single operation. paint() initializes the offscreen buffer the first time it runs. (Chapter 1 discusses the details of using an offscreen image.)
- horizInset holds the number of pixels to inset the hand from the left and right edges of the canvas. The Blackjack applet uses the inset space to highlight the active hand. setActive() sets or clears the hand's active state.

Arranging the User Interface

Like the Web pages in which Java runs, Java configures its screen area to fit the available space and allows a Java applet to automatically adapt to smaller or larger screens. Creating an interface that configures itself presents challenges. On the one hand, you don't have to worry about specifying the pixel location of each interface component. On the other hand, trying to get your game to look just the way you want can be frustrating. Compounding this frustration is the fact that the size and look of the individual components vary between platforms. But rest assured, with a little work you can create a user interface that looks good and intelligently configures itself to fit a variety of screen sizes.
**Positioning components With a LayoutManager**

The AWT uses classes that implement the `java.awt layoutManager` interface to determine what size to make AWT components and where to place them. Each container - a `Panel`, `Applet`, `Window`, `Frame`, `Dialog`, or `FileDialog` - has its own layout manager to accomplish this task.

A layout manager positions each of the components in the container and can also set the sizes of the contained components. The layout manager calls a component's `move()` method to position the component, calls `resize()` to set the component's size, and calls `reshape()` to set both the size and position. Most layout managers use the component's `preferredSize()` and `minimumSize()` methods to determine its dimensions, but some layout managers ignore these suggested sizes (see Table 6-3).

You can subclass `Label`, `Button`, and other AWT components to override the `preferredSize()` and `minimumSize()` methods. By doing so, you can explicitly control the size of the laid-out component.

The AWT provides five layout managers - `Border Layout`, `Card Layout`, `FlowLayout`, `Grid Layout`, `GridBagLayout`. Table 6-3 lists types of layout managers assigned by default to each type of container.

<table>
<thead>
<tr>
<th>Container</th>
<th>Layout Manager</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Panel</td>
<td>FlowLayout</td>
<td></td>
</tr>
<tr>
<td>Applet</td>
<td>FlowLayout</td>
<td></td>
</tr>
<tr>
<td>Window</td>
<td>BorderLayout</td>
<td>(ignores suggested size)</td>
</tr>
<tr>
<td>Frame</td>
<td>BorderLayout</td>
<td>(ignores suggested size)</td>
</tr>
<tr>
<td>Dialog</td>
<td>BorderLayout</td>
<td>(ignores suggested size)</td>
</tr>
<tr>
<td>FileDialog</td>
<td>null</td>
<td></td>
</tr>
</tbody>
</table>

A container can have a single layout manager for which you must explicitly position its components. A container with a layout that doesn't need to adapt to different screen sizes - for example, a container that conforms to a background graphic - may be best with a single layout manager.

If you want to use a different layout manager than the default, pass the new layout manager to the container's `setLayout()` method. For example, if you want to have an applet use a `Border Layout`, place the following in the applet's `init()` method:

```
```
If you change a container’s layout manager after it has already been laid out, you need to tell the container. To do so, call `invalidate()` to tell the container to invalidate its current layout and then call `validate()` to have the container recursively layout itself along with any nested containers.

**FlowLayout**

You use a `FlowLayout` manager to arrange the components from left to right and top to bottom. Each horizontal row aligns its components `LEFT`, `CENTER`, or `RIGHT`. The default alignment is `CENTER`.

The following code produces the applet that Figure 6-8 shows:

```java
public class Flow extends java.applet.Applet
public void init ()
for (int i = 1; i < 10; i++)
    add(new java.awt.Button(Integer.toString(i)));```

**BorderLayout**

Manages as many as five components: one along each of the four sides and one in the center of the container. You call the container’s `add()` method with the name of the location (`North`, `South`, `East`, `West`, or `Center`) at which to add a component to a `BorderLayout` container. If you don’t specify a location, the layout manager places the component in the center location.

`BorderLayout` resizes the components according to the following rules:

- The north and south components use their preferred height but are resized as wide as the container.
  The east and west components use their preferred width but are resized to be, as tall as the container minus the heights of the north and south components.
The center component is resized to fill the remaining space not used by the other components in the container. It is as tall as the east and west components, and as wide as the container minus the widths of the east and west components.

The following code produces the applet shown in Figure 6-9:

```java
public class Border extends java.applet.Applet {
    public void init () {
        setLayout(new java.awt.BorderLayout());
        add( North , new java.awt.Button( North ));
        add( East , new java.awt.Button( East ));
        add( West , new java.awt.Button( West ));
        add( South , new java.awt.Button( South ));
        add( Center , new java.awt.Button( Center ));
    }
}
```

**GridLayout**

You use a GridLayout to arrange all the components on an evenly spaced grid. GridLayout resizes each component to fit the grid. You can create the applet Figure 6-10 shows by replacing the `setLayout()` call in the preceding `Border` example with

```java
public class Border extends java.applet.Applet {
    public void init () {
        setLayout(new java.awt.GridLayout());
        add( North , new java.awt.Button( North ));
        add( East , new java.awt.Button( East ));
        add( West , new java.awt.Button( West ));
        add( South , new java.awt.Button( South ));
        add( Center , new java.awt.Button( Center ));
    }
}
```

**Your own LayoutManager**

If none of the layout managers supplied by the AWT is right for your container, you can always write your own. You create a layout manager by implementing the `LayoutManager` interface in your class.
A layout manager needs to do two things:

- Determine the preferred and minimum sizes needed to layout the components in a container.
- Lay out a container's components, which requires positioning and possibly sizing the components.

The Blackjack applet uses a custom layout manager to arrange the hands for a player. A Blackjack player starts out with a single hand of cards. When a player splits a hand, the hand becomes two new hands. (Remember: The player may split a hand into two hands if the first two cards have the same point value.) Further, the player can potentially split either or both of the new hands. Because it takes a lot of space to display each hand, reserving space for the unlikely possibility that each player would end up with four hands more than doubles the size of the applet. A better solution is to overlap the hands to cover up part of the hand, as Figure 6-11 shows.

The following custom layout manager lays out the hands as Figure 6-11 shows:

```java
import java.awt.

class PlayerHandLayout implements layoutManager {
    private BlackjackPlayer player;
    private float overlap;

    public PlayerHandLayout (BlackjackPlayer player, float overlap) {
        this.player = player;
        this.overlap = overlap;
    }

    public void addLayoutComponent (String nm, Component c) {}
    public void removeLayoutComponent (Component c) {}
    public Dimension preferredLayoutSize (Container parent) {
        Dimension ds = ((BlackjackHand)parent.getComponent(0)).preferredSize();
        int numHands = parent.countComponents();
        if (numHands <= 2)
            return new Dimension(ds.width, ds.height * numHands);
        return null;
    }
}
```
Part II: Up to Speed

(continued)

```java
else
    return new Dimension(ds.width, (ds.height * 2) +
                          (numHands - 2) * (int)(overlap * ds.height));
```

```java
public Dimension minimumLayoutsize (Container parent) I
    return preferredLayoutSize(parent);
```

```java
public void layoutContainer (Container parent) I
    Dimension hsize = ((BlackjackHand)parent.getComponent(0)).preferredSize();
    int numHands = parent.countComponents();
    int h, y = D;
    for (int ii = 0; ii < numHands ; ii++)
        if (ii == player.getHandIndex() ||
            ii == player.getActiveHandCount(3-1))
            h = hsize.height;
        else
            h = (int)(hsize.height * overlap);
    parent.getComponent(ii).
        reshape(0, y, hsize.width, hsize.height);
    y += h;
```

The prevalent aspects of P1ayerHandLayout are:

- The BlackjackPlayer passes a reference to itself and the vertical overlap ratio to the P1ayerHandLayout() constructor.
- layoutContainer() uses player to access information about the hand.
- layoutContainer() leaves the active hand exposed so that it’s easier to recognize. Figure 6-12 shows the layout of four split hands where the second hand is the exposed, active hand.
- layoutContainer() calls reshape() to set the size of overlapped hands to the size of the exposed area of the hand. Using reshape() is important, because you can’t rely on the draw order to obscure other hands; the AWT doesn’t guarantee any particular draw order for the components.
Frequently, you cannot arrange an applet or other container with a single layout manager. In these cases, you can arrange your interface hierarchically by grouping components within panel containers and using the panel’s layout manager to arrange the components inside the panel. The panel itself is a single component arranged by the layout manager for the container containing the panel.

For example, consider an applet with four groups of radio buttons and several standard buttons. You can use these steps to arrange the radio buttons into separate columns and the standard buttons into a row across the bottom:

1. Create a panel for each column of radio buttons.
   - Give each of these panels a Grid Layout (0, 1) layout manager. Add the radio button Check box components to their respective groups.
2. Create a panel to hold the radio button column panels.
   Leave its layout manager set to the default FlowLayout. Add the column panels to this group.

3. Create a panel for the standard buttons.
   Leave its layout manager set to the default FlowLayout. Add the Button components to this group.

4. Give the applet a BorderLayout layout manager.
   Add the panel from Step 2 to the Center location of the border layout. Add the panel from Step 3 to the South location of the border layout.

Figure 6-13 shows the hierarchical organization these steps produce.

Panel with Grid Layout 

You can also arrange this example with a GridBagLayout. A GridBagLayout is a powerful layout manager for creating complex arrangements. Unfortunately, using a GridBagLayout is a fairly complicated process, and nested panels, as in this example, are usually easier to create and maintain.

Laying out a game of Blackjack

Figure 6-2 earlier in this chapter shows the logical organization of the Blackjack applet. This organization translates to a physical organization on the screen, as Figure 6-14 shows.

The top-level applet
The Blackjack applet that implements the top level of the game is as follows:
import java.awt.*;
import java.util.StringTokenizer;
public class BlackjackApplet extends java.applet.Applet
   implements Runnable
private Deck deck;
private BlackjackHand dealer, lastActive;
private BlackjackPlayer[] players;
private int curPlayer;
private Button newdeal, stand, hit, doubledown, split;
private ThreadGroup appTG;
private Panel dpan, ppan, bpan;

public void init () {
   int numPlayers = 1, deckSize = 1, bankroll = 0;
   String[] names = null;
   Thread curT = Thread.currentThread();
   appTG = curT.getThreadGroup();
   curT.setPriority(curT.getPriority()) = 1);
   setLayout(new BorderLayout());
   (continued)
if ((param = getParameter("PLAYERS")) != null)
    StringTokenizer st = new StringTokenizer(param,
    names = new String[st.countTokens()];
    for (numPlayers = 0; st.hasMoreTokens(); numPlayers++)
        names[numPlayers] = st.nextToken();

Card.initGraphics(this);  // load the card images
    deck = new Deck(deckSize);
    players = new BlackjackPlayer[numPlayers];
    Bpan = new Panel();
    dpan.add(dealer = new BlackjackHand());
    add(North, dpan);
    bpan = new Panel();
    add(Center, bpan);
    bpan.add(newdeal = new Button("Deal"));
    bpan.add(new Label( ""));
    bpan.add(stand = new Button("Stand"));
    bpan.add(hit = new Button("Hit"));
    bpan.add(doubledown = new Button("Double Down"));
    bpan.add(split = new Button("Split"));
    setButtons(null);
    ppan = new Panel();
    add(South, ppan);
    for (int ii = 0; ii < numPlayers; ii++)
        players[ii] = new BlackjackPlayer(ppan,
            names == null ? null : names[ii], bankroll);

public boolean action (Event evt, Object what) I
    if (evt.target == newdeal)
        new Thread(appTG, this).start();
    else I
        BlackjackPlayer player = getPlayer();
        if (evt.target == stand)
            nextHand(false);
        else
            BlackjackPlayerHand hand = player.getHand();
            if (evt.target == hit) i
                hand.deal(deal());
                if (hand.value() > 21)
                    nextHand(false);
            else
                setButtons(player);
else if (evt.target == doubledown) {
    hand.deal(deal());
    player.addToBankroll(-hand.bet);
    hand.bet <<= 1;
    nextHand(false);
}

else if (evt.target == split) {
    BlackjackPlayerHand splitHand =
        player.newHand(hand.bet);
    hand.split(splitHand);
    hand.deal(deal());
    splitHand.deal(deal());
    setButtons(player);
    return super.action(evt, what);
}

public synchronized void run () {
newDeal();
newDeal.disable();
try {
    BlackjackPlayerHand hand;
    for (int card = 2; --card != 0; )
        for (int pp = 0; pp < players.length; pp++)
            if ((hand = players[pp].getHand()) != null) {
                hand.deal(deal());
                Thread.sleep(500);
            }
    dealer.deal(deal());
    Thread.sleep(500);
    if ((hand = nextHand(true)) != null) {
        setButtons(getPlayer());
        wait(); // wait for players to play their hands
    }
    setButtons(null);
    dealer.expose();
    Thread.sleep(1000);
    if (hand != null)
        while (dealer.value() <= 16) {
            dealer.deal(deal());
            Thread.sleep(500);
        }
    
    // The remaining code continues here.
}
Part I: Up to Speed

(continued)

```java
catch (InterruptedException e) {
    for (int pp = 0; pp < players.length; pp++)
        players[pp].resolveDeal(dealer);
    newDeal.enable();
}

BlackjackPlayer getPlayer ()
    return curPlayer == players.length ?
        null : players[curPlayer];

BlackjackPlayer nextPlayer ()
    curPlayer++;
    return getPlayer();

BlackjackPlayerHand nextHand (boolean firstHand) {
    BlackjackPlayer player = null;
    BlackjackPlayerHand hand = null;
    if (dealer.blackjack())
        curPlayer = players.length;
    else if ((player = getPlayer()) != null)
        hand = firstHand ?
            player.getHand() : player.nextHand();
    while (player != null && hand == null)
        if ((player = nextPlayer()) != null)
            hand = player.getHand();
    if (player == null)
        setButtons(null);
    synchronized (this) I notify(); // deal to dealer
    return null;

    if (hand.blackjack()) // skip this hand
        return nextHand(false);
    setButtons(player);
    return hand;
}

boolean newDeal ()
    int ii = players.length;
    boolean shuffled;
    curPlayer = 0;
    if (shuffled = (deck.getSize() < ii * 3 + 3))
        deck.reshuffle();
```
dealer.clearHand();

for (int i = players.length - 1; i >= 0; i--)
    players[i].clearHands();

newDeal.enable();

setButtons(null);

return shuffled;

private void setButtons(BlackjackPlayer player) {
    if (lastActive != null) {
        lastActive.setActive(false);
        lastActive = null;
    }

    stand.disable();
    hit.disable();
    split.disable();
    doubledown.disable();

    if (player != null) {
        BlackjackPlayerHand hand = player.getHand();
        if (hand != null) {
            lastActive = hand;
            lastActive.setActive(true);
            stand.enable();
        }
        int val = hand.value();
        if (val < 21) {
            hit.enable();
            if (player.getBankroll() >= hand.bet)
                if (player.canSplit() && hand.canSplit())
                    split.enable();
                if (val <= 11 && hand.cardCount() == 2)
                    doubledown.enable();
        }
    }
}

private Card deal() {
    try {
        return deck.deal();
    }
    catch (NoSuchElementException e) {
        deck.reshuffle();
        return deal();
    }
}

J1 end class BlackjackApplet
The HTML that loads the applet

The HyperText Markup Language (HTML) document invokes the Blackjack applet using the following applet tag:

```html
<APPLET CODE=BlackjackApplet WIDTH=480 HEIGHT=460>
<PARAM NAME=PLAYERS VALUE=Doug,Wayne,Fast Freddy
</APPLET>
```

The *BlackjackApplet* gets going like this:

1. HTML passes the player names to the applet.

   The applet automatically adjusts the layout for more or fewer players.

2. The applet uses a *BorderLayout* to divide the applet into the dealer's hand at the top (*North*), the button controls in the middle (*Center*), and the players at the bottom (*South*).

3. *init()* creates a panel for each of the *BorderLayout* locations to hold the individual components.

   The *North* location contains only the dealer's hand, but still uses a panel in order to prevent the border layout from resizing the dealer's hand, and cause border layout to resize the panel instead. Each of these panels uses the default *FlowLayout* manager.

4. *setButtons()* enables and disables the buttons to match the legal options for a given point in the game.

   The state-driven approach determines by examining the state of the game the required state of the buttons.

5. *deal()* deals cards from the deck and automatically reshuffles if the cards have all been dealt.

6. The applet spawns a new thread when the user presses the Deal button.

   The thread controls the tempo of the deal from the *run()* method.

7. After *run()* deals the initial hands, the applet's Deal thread needs to wait for the players to play their hands.

   *run()* calls *wait()* on the applet and waits for *action()* to handle the user options.

8. *nextHand()* determines the next hand to play and sets it as the active hand.

   If the next hand is the dealer's hand, *nextHand()* calls *notify()* to wake up the Deal thread. When the Deal thread wakes up, *run()* plays the dealer's hand, settles the bets with each player, enables the Deal button, and exits the thread.
The players

Each player requires additional user interface elements:

- Label s to display the player's name and current bankroll
- A C h o i c e selector to allow the amount of the next bet to be selected
- A P a n e l to display the player's Blackjack hand(s)

B l a c k j a c k P l a y e r panel groups together these interface elements. A B l a c k j a c k P l a y e r also includes all the necessary methods and fields for keeping track of a player. The following code implements B l a c k j a c k P l a y e r:

```java
import java.awt.*;

class BlackjackPlayer extends Panel {
    public static final int STANDARD_BET = 10;
    public static final int STANDARD_BANKROLL = 1000;
    private static int[] betAmount = {1, 5, 10, 20, 30, 50, 100, 200, 500, 1000};
    private static int playerCount;
    private String name;
    private int lastBet, bankroll, curHand, numHands;

    private final BlackjackPlayerHand[] hands = new BlackjackPlayerHand[4];
    private Label bankrollLabel;
    private Choice betEntry;
    private Panel handsPanel;

    BlackjackPlayerHand getHand () {
        return curHand >= numHands ? null : hands[curHand];
    }

    int getHandIndex () {
        return curHand >= numHands ? 1 : curHand;
    }

    BlackjackPlayerHand nextHand () {
        curHand++;
        handsPanel.layout();
        return getHand();
    }

    int getBankroll () { return bankroll; }
    int getActiveHandCount () { return numHands; }
}
```

(continued)
(continued)

```java
boolean canSplit () { return numHands < hands.length; }
BlackjackPlayer (Container parent) {
    this(parent, null, STANDARD_BANKROLL);
}

BlackjackPlayer (Container parent, String name, int bankroll) {
    parent.add(this);
    setLayout(new BorderLayout());
    playerCount++;
    if (name == null)
        name = Player + playerCount;
    add(North, new Label(this.name = name, Label.CENTER));
    if (bankroll <= 0)
        bankroll = STANDARD_BANKROLL;
    String bankStr = Integer.toString(bankroll);
    Font font = new Font(Courier, Font.PLAIN, 10);
    bankrollLabel = new Label(bankStr, Label.RIGHT);
    betEntry = new Choice();
    for (int ii = 0; ii < betAmount.length; ii++)
        betEntry.addItem(Integer.toString(betAmount[ii]));
    bankrollLabel.setFont(font);
    betEntry.setFont(font);
    Panel pbank = new Panel();
    add(Center, pbank);
    pbank.add(bankrollLabel);
    pbank.add(betEntry);
    addToBankroll(bankroll);
    setBet(STANDARDBET);
}

handsPanel = new Panel():
    handsPanel.setLayout(new PlayerHandLayout(this, .3Of));
    handsPanel.setBackground(Color.gray);
    add(South, handsPanel);
    for (int hh = 0; hh < hands.length; hh++)
        handsPanel.add(hands[hh] =
            new BlackjackPlayerHand(this));

public boolean action (Event evt, Object what)
    if (evt.target == betEntry)
        setBet(Integer.parseInt(betEntry.getSelectedItem()));
```
synchronized void clearHands () {
    while (numHands > 0)
        hands[--numHands].clearHand();
    curHand = 0;
    handsPanel.layout();
}

synchronized void resolveDeal (BlackjackHand dealer) {
    for (int hh = numHands; --hh >= 0; )
        addToBankroll(hands[hh].winnings(dealer));
}

synchronized BlackjackPlayerHand newHand (int bet) {
    BlackjackPlayerHand result = null;
    if (bet == 0)
        bet = lastBet;
    if (numHands < hands.length && (bet = setBet(bet)) > 0) {
        result = hands[numHands++];
        addToBankroll(-(result.bet = bet));
        handsPanel.layout();
    }
    return result;
}

int setBet (int bet) {
    if (bet > bankroll)
        for (int i = betAmount.length; i > 0 && (bet = betAmount[--i]) > bankroll; )
            betEntry.select(Integer.toString(bet));
    return lastBet = bet;
}

void addToBankroll (int amount) {
    this.bankroll += amount;
    bankrollLabel.setText(Integer.toString(this.bankroll));
}

Chapter 6: Blackjack
The responsibilities of a BlackjackPlayer are

- **BlackjackPlayer** can have up to four hands active at once. The `hand` array field stores all the hands.

- BlackjackPlayer uses a BorderLayout and places the name label in the **North** location, the current bankroll and the bet selector in the **Center**, and the player's Blackjack hands in the **South**.

- `getHand()` and `nextHand()` return and update the current active hand. `newHand()` activates a new hand after first checking that the player has enough money to cover the bet.

- The `addToBankroll()` and `setBet()` methods manage the player's money. These methods are responsible for making sure that the player never gambles money he doesn't have.

- BlackjackPlayer overrides `action()` to handle the events from the `Choice` selector used to set the player's next bet.

### The players' hands

The hand for a player is a specialized case of the `BlackjackHand` that the dealer uses, as discussed earlier in this chapter. On top of the dealer's hand’s function, the player’s hand has to handle the additional duties of wagering and splitting. The `BlackjackPlayerHand` extends the `BlackjackHand` like this:

```java
import java.awt.*;

class BlackjackPlayerHand extends BlackjackHand {
    int bet;
    private boolean hasSplit;

    BlackjackPlayerHand (Container parent) {
        super();
        horizInset = 3;
        parent.add(this);

        protected boolean isDealer () { return false }
        boolean blackjack () {
            return !hasSplit && super.blackjack();
        }

        boolean canSplit () {
            return hand.size() == 2 && value(o) == value(1);
        }
    }
}
```
void clearHand ()
    super.clearHand();
    hasSplit = false;

void split (BlackjackPlayerHand splitHand) {
    splitHand.deal((Card)hand.elementAt(l));
    hand.removeElementAt(l);
    hasSplit = splitHand.hasSplit = true;
}

int winnings (BlackjackHand dealer) {
    int hval = value();
    if (hval <= 21) {
        if (blackjack()) {
            if (dealer.blackjack())
                return bet;
            else
                return (bet << 1) + (bet >> 1);
        } else {
            int dval = dealer.value();
            if (rival ? dval > 21 || rival < hval)
                return bet << 1;
            else if (dual == hval && !dealer.blackjack())
                return bet;
        }
    }
    return 0;
}
Part II: Up to Speed
Chapter 7

2-D Maze

Computer games are about challenges that the game player attempts to overcome. A maze is a confusing, intricate network of passages. The challenge of finding your way around a maze makes computer games and mazes a perfect match. In one form or another, many games base their game environments on a maze. Here are some examples of the types of mazes computer games use:

Adventure games use graph mazes in which locations in the environment connect to each other in an arbitrary arrangement. Any point on the graph can connect to any other point. These mazes have less to do with representing a physical structure than with organizing the sequence in which the game is played.

Games like PacMan and DungeonMaster use a block grid maze in which a flat grid of uniform rectangles defines the maze. Each square is either open (a floor) or closed (a solid wall).

Wizardry, one of the original fantasy role-playing games, uses a wall grid maze in which a flat grid of uniform rectangles defines the maze. The edges of the rectangles define the walls, and the planes of the rectangles define the floor and ceiling.

Games like Doom use an extruded polygon maze in which adjacent polygonal columns define the regions in the game. The bottom and top surfaces of the polygonal column define the floor and ceiling, and the sides of the columns define the walls. (Doom varies the height of the floor and ceiling polygons in order to effectively create the illusion of a 3-D maze, but the actual topography of the maze is two-dimensional.)
This chapter shows how to create block and wall grid mazes and how to find a path between two locations in a maze. Chapter 8 uses the block maze from this chapter as a playing field for "intelligent" computer adversaries called sprites that incorporate the capability of navigating a maze.

**Creating the Maze Class**

You use a two-dimensional array to represent grid mazes. Wall and block mazes (both types use a grid) have many common features, so the mazes this chapter presents use a common abstract class named Maze. Maze extends the Canvas component from the java.awt package so that a maze can display itself. The WallMaze and BlockMaze classes extend Maze to implement the features specific to each type of maze.

You can extend an abstract class, but you can’t instantiate (have objects created from) the class. You use abstract classes as superclasses to implement common functionality for subclasses. An abstract class can contain abstract methods. Abstract methods don’t have implementations and simply define methods that must be implemented by nonabstract subclasses.

The Maze class uses a two-dimensional array of bytes (a byte holds an 8-bit value in the range -128 to 127) to represent the maze. Each byte in the array represents a single rectangle in the grid that makes up the maze. The values in the array have different meaning in the WallMaze and BlockMaze subclasses. However, both classes reserve the high bit (the bit corresponding to the value 0x80) in each byte for the display code in Maze. The display code (the methods in Maze involved with drawing the maze on the screen) uses the high bit as a Dirty flag to keep track of which squares have changed since paint() drew them on the screen - when the high bit is set to one, the square is "dirty" and needs to be drawn. (The "Displaying a 2-D Maze" section later in this chapter discusses how Maze uses the Dirty flag.)

Computers store values using binary numbers composed of a series of bits set to 0 or 1. The bits are ordered from the high bit on the left to the low bit on the right. Sometimes it is useful to use the individual bits to store several values in a single number.

The declaration of the Maze class starts with the following:

```java
abstract class Maze extends Canvas {
    static final byte TOP = 0x01;
}
```
Maze also contains a constructor Maze() method for initializing a blank maze. Because wall and block mazes use different values for the squares in the maze[][] array, Maze defines the abstract method initSq() to return the initial value for the squares with this line:

```java
protected abstract byte initSq();
```

Declaring initSq() as an abstract method requires that any classes that extend the Maze class must implement initSq().

The complete source code for the Maze class is on the Java Game Programming For Dummies CD-ROM at the back of this book.

**The Block Maze subclass**

Each square in a block maze is either floor or wall. Figure 7-1 shows an example of a block maze.

```java
static final byte RIGHT = 0x02;
static final byte BOTTOM = 0x04;
static final byte LEFT = 0x08;
static final byte DIRTY = (byte) 0x10;

protected byte[][] maze;

protected abstract byte initSq();
```

To extend Maze to implement a block maze, you implement the abstract method initSq() that Maze defines. The portion of the BlockMaze class responsible for representing the maze looks like this:

```java
public class BlockMaze extends Maze
    public static final byte WALL = 0, FLOOR = 1;
    protected byte initSq() { return (byte) (WALL | DIRTY); 1
    // end class BlockMaze
```
The code (\texttt{WALL | DIRTY}) "ORs" the \texttt{DIRTY} flag with the \texttt{WALL} value to set the high bit and indicate that the code has changed the square since the last time the maze displayed the square, or in this case, has set a square that has never been displayed.

\textbf{The \texttt{WallMaze} subclass}

Each square in a wall maze needs to keep track of which sides of the square have walls and which sides are open. Figure 7-2 shows an example of a wall maze.

You give squares the capability to keep track of themselves by setting or clearing a different bit for each wall in the byte for the maze grid square. You set the bit when the wall exists and clear the bit when the wall is open. The \texttt{LEFT}, \texttt{RIGHT}, \texttt{TOP}, and \texttt{BOTTOM} static final variables in \texttt{Maze} define the bit values used in \texttt{Maze}. The portion of the \texttt{WallMaze} class responsible for representing the maze looks like

```java
public class WallMaze extends Maze {
    public static final byte BLOCKED = (byte) (TOP | RIGHT | BOTTOM | LEFT);

    protected byte initSq() { 
        return (byte) (BLOCKED | DIRTY);
    }
}
```

You display a square with all walls set as a solid wall. The static final \texttt{BLOCKED} variable holds the value with all the walls set - a solid wall is on all sides. \texttt{initSq()} creates its return value from \texttt{BLOCKED}.

Note that because squares share walls with adjacent squares, \texttt{Maze} records every interior wall in the two squares that share it. (We define an interior wall as a wall between two squares as opposed to an exterior wall on the edge of the maze.) Recording the wall in two places means that you must always change both squares when setting or clearing a wall. Although this approach is redundant and therefore susceptible to errors, it makes
People count using decimal numbers. Each digit in a decimal number can be one of the ten values from 0 to 9. The decimal counting system is called \textit{base} 10 because it has ten digits. People probably use decimal numbers because our counting system arose from primitive people counting on their fingers. You (most likely) have ten fingers and can represent ten different values by holding up some number of ur fingers.

Computers don’t have fingers to count on – at least not yet – and instead use a series of on/off values as a counting system. Every value stored in a computer is composed of bits. Each bit is either an off value or an on value. By convention, an off bit represents the value 0 and an on bit represents 1. The computer represents values larger than 1 by combining bits to form \textit{base} 2 or binary numbers.

Java doesn’t have a representation for using msnary numbers in your code, but it does have \_e for \textit{hexadecimal} (\textit{base} 16) numbers. In the \textit{hexadecimal} counting system, each digit is one of the values from 0 to 9 or from A to F, where A = 10, B = 11, and so on. A hexadecimal digit represents exactly four binary digits so converting between hexadecimal and binary numbers is relatively easy. You indicate a hexadecimal number in Java by beginning the member with a zero followed by an x, like this:

```
0x10 0xA82B3 0xDEADBEEF 0x17
```

For example, the decimal number 23 is 10111 in binary and 0x17 in hexadecimal. 23 stands for 2 tens and 3 ones; 10111 stands for 1 sixteen, 0 eights, 1 four, 1 two, and 1 one; and 0x17 stands for 1 sixteen and 7 ones.

\begin{align*}
20 + 3 & = 23 \\
10 & = 16 + 4 + 2 + 1 = 10111 \\
16 + 7 & = 17 16
\end{align*}

You use the “bitwise” operators \&, \mid, \&\&\&, <<, >>, and >>> to work with individual bits and the “or” (\mid) operator to set or combine bits:

\begin{align*}
\text{x \&= 1} & \quad \text{// sets the ones bit in x} \\
\text{x \mid= 1} & \quad \text{// sets all the bits in x except the ones bit.}
\end{align*}

You use the “and” (\&) operator to clear or test bits:

\begin{align*}
\text{(x \& 1) \neq 0} & \quad \text{// true if the ones bit in x is set} \\
\text{x \&= 1} & \quad \text{// clears all bits in x except the ones bit.}
\end{align*}

You use the “exclusive-or” (^) operator to toggle bits:

\begin{align*}
\text{x \^= 1} & \quad \text{// toggles the ones bit in x (0\rightarrow1, 1\rightarrow0)}
\end{align*}

You use the “bitwise complement” (\~) operator to toggle all the bits in a value:

\begin{align*}
\text{x \~= 1} & \quad \text{// clears the ones bit in x}
\end{align*}

You use the “left shift” operator (\ll) to move all the bits in the value to the left:

\begin{align*}
\text{x \ll= 1} & \quad \text{// equivalent to x *= 2 (10111 \ll 1 = 101110)}
\end{align*}

You use the “right shift” operator (\gg) to move all the bits in the value to the right:

\begin{align*}
\text{x \gg= 1} & \quad \text{// equivalent to x i= i = 10111)}
\end{align*}
Generating a Maze

One way to add infinite variety to your game is to randomly generate the game environment. The problem with random-generated environments is that they tend to be less interesting than hand-crafted environments. However, when the central element in the game environment is a maze, pve have some good reasons to use a randomly generated maze rather than a hand-crafted maze:

- To prevent players in a multiplayer game from gaining or losing advantage due to familiarity with the terrain.
- To connect hand-crafted environments with sections of randomly generated mazes so that each time the player plays the game, or perhaps each time they enter the section with the generated maze, the maze is different.
- To create environments that extend indefinitely.
- To reduce download time by creating the environment on the player's browser instead of downloading it.
- Because solving the maze is the game.

The code in this section generates mazes that start on the left edge and finish on the right edge. The only reason the maze generation uses start and finish squares is to make the animations for generating and solving the maze more interesting. You can arbitrarily select start and finish squares, if you even need them, after generating the maze.

Selecting an algorithm

You can use a number of different algorithms to generate mazes. The most important consideration when selecting an algorithm is what type of maze you want to generate. Some of the questions to consider when selecting a maze generation algorithm are

- Do you want to be able to navigate between any two points in the maze?
- Do you want the maze to connect back on itself so that it has more than one path between two points in the maze? If so, how much interconnection do you want?
- Do you want rooms or open spaces in the maze?
- Do you want a dense maze or a sparse maze?
Do you want to favor straight hallways or twisty passages?

- Do you want lots of branching passageways or longer stretches between branches?

This section shows you how to implement two different maze generation algorithms: one for generating wall mazes and one for block mazes. Both of these algorithms create dense mazes that allow navigation between any two points in the maze.

The wall maze algorithm creates mazes that have a single path between any two points in the maze, are constructed entirely of passages with no open space, and favor twisty passages with moderate branching.

The block maze algorithm is configurable: You can change the settings for the generator in order to produce generated mazes with different characteristics. The default settings create mazes that allow multiple paths between points, have small rooms and open spaces, and favor straighter passages with lots of branching.

Wall mazes and block mazes impose different constraints on the generator. In particular, block mazes are a little trickier to generate because you have to leave room in the grid to create walls. Wall mazes can place a wall between any two squares on the grid, so you don't need to reserve squares on the grid to separate passages.

In general terms, the algorithm for generating a maze is

1. Initialize the maze so that every square is a solid wall.
2. Select a square in which to start the maze.
3. Extend the path from the selected square to an adjacent square.
4. Select a square on the current path.
    - Frequently, the algorithm selects the adjacent square from Step 3 in order to continue along the same path.
5. Repeat Steps 3 and 4 until done.

The algorithm may consider the maze done when all squares have been included, or may use a combination of criteria to decide to call it quits.

The wall maze and block maze generation algorithms are both based on this general maze-generating algorithm, but the specific details of each algorithm vary. In particular, the algorithms differ in how they choose the adjacent square in Step 3 and how they select a new square in Step 4.
Adding to the Maze class

You add the common methods and fields needed to generate mazes to the Maze superclass. (The "Creating the Maze Class" section earlier in this chapter presents the Maze class.) You add the specific generation algorithms to the WallMaze and BlockMaze subclasses. Here are the fields and methods you add to the Maze superclass to keep track of which squares are "dirty" and need to be redrawn by paint0ffScreen Image()

protected int minXdirty, minYdirty, maxXdirty, maxYdirty;

protected int dirtySquare (int x, int y)
if (x < minXdirty) minXdirty = x;
if (x > maxXdirty) maxXdirty = x;
if (y < minYdirty) minYdirty = y;
if (y > maxYdirty) maxYdirty = y;
return maze[x][y] = DIRTY;

The (minXdirty, minYdirty) and (maxXdirty, maxYdirty) fields keep track of the upper-left and lower-right limits of the squares that have changed - and are therefore "dirty" - since paint0ffScreen Image() drew them. The dirtySquare() method maintains the "dirty" fields. You use the "dirty" fields and methods to minimize the work done to redraw the screen. The "Displaying a 2-D Maze" section later in this chapter shows how the dirty fields help minimize redraw time.

If you don’t keep track of which part of the maze has changed, you have to draw the entire maze each time it changes, and the animation rate slows down dramatically under most Web browsers and Java runtimes.

You also add declarations to Maze for the following abstract methods that must be implemented by subclasses of Maze:

abstract boolean isOpen (int x, int y):
abstract void generate (boolean displaySearch);

The abstract method isOpen() defines a method to test a square in the maze grid in order to determine whether it’s a solid wall. Each type of maze implements a different test.

Maze defines the abstract method generate(), which the classes that extend Maze implement in order to generate the maze. generate() accepts a boolean parameter of true if you want the maze to display its progress as it builds itself. You pass false to generate() if you don’t want the progress of the maze displayed as it’s built.
Generating a wall maze

Here are the steps the `WallMaze` class uses to generate a maze:

1. Set all the squares in the maze grid to the `BLOCKED` state.
2. Randomly select a square on the left edge of the maze as the current square, clear the wall on the left side of the square, and set `nextSide` to `LEFT`.
3. Randomly choose a sequence to rotate through the remaining sides (the sides other than `nextSide`) of the current square.
4. Set `nextSide` to the next selected side in the rotation.
5. If a `BLOCKED` square lies adjacent to the `nextSide` of the current square, go to Step 9.
6. If more sides remain in the rotation sequence decided on in Step 3, go to Step 4.
7. You get to this step when the current square isn’t adjacent to any `BLOCKED` squares. Randomly select a non-`BLOCKED` square (a square already added to the maze) as the current square.
   This creates a new branch in the maze by starting a path at the newly selected square.
8. Set `lastSide` to one of the open sides of the selected square and go to Step 3.
9. You get to this step when you have found a square to add to the maze. Remove the wall between the current square and the square adjacent to its `nextSide`.
10. Set the new current square to the adjacent square and set `nextSide` to the wall of this square removed in Step 9.
    This sets `lastSide` to the side opposite `nextSide`. For example, if `nextSide` is `TOP`, the new `lastSide` is `BOTTOM`.
11. If any `BLOCKED` squares remain in the maze grid, go to Step 3, otherwise you’re done.

The `WallMaze` class implements the abstract methods `isOpen()` and `generate()` defined in the `Maze` superclass like this:

```java
public boolean isOpen(int x, int y)
    return inBounds(x, y) && sgr(x, y) != BLOCKED;
```

(continued)
public synchronized void generate
    (boolean displayConstruction)

    int xx, yy, sq, lastSide = LEFT;
    int count = mzWid * mzHyt, threshold = count
    // Step #1 - initialize the maze
    clearMaze();
    if (displayConstruction)
        showMaze(true);
    // Step #2 - select and set the starting square
    startX = xx = 0;
    startY = yy = rint(mzHyt);
    sq = (byte) ((BLOCKED | DIRTY) & LEFT);
    while ( --count >= 0) {
        // Step #3 - choose a sequence to rotate thru the sides
        int nextSide, nx = 0, ny = 0, nsq;
        int scnt = 3; // # of sides left to try in current sq
        int sideInc = rint(3); // offset from lastSide to try
        boolean branch = false, found = false;
        do {
            // Step #4 - set nextSide to direction to search
            nsq = 0;
            if ((nextSide = lastSide << (sideInc + 1)) > BLOCKED)
                nextSide >>= 4;
            switch (nextSide) { // get next square to add to maze
                case TOP:  // get next square to add to maze
                    if (yy > 0)
                        nsq = sgr(nx = xx, ny = yy)
                    break;
                case BOTTOM:
                    if (yy < mzHyt - 1)
                        nsq = sgr(nx = xx, ny = yy + 1);
                    break;
                case LEFT:
                    if (xx > 0)
                        nsq = sgr(nx = xx - 1, ny = yy);
                    break;
                case RIGHT:
                    if (xx < mzWid - 1)
                        nsq = sgr(nx = xx + 1, ny = yy);
                    else if (finishX < 0) // mark sq as maze exit
                        found = branch = true;
                    finishX = xx; finishY = yy;
                    break;
if (!found) {
    if (nsq == BLOCKED) {
        // unused square, use it
        found = true;
        // Step #5 - add the square
    } else if (--sent > 0) {
        // try next direction
        sideInc = (sideInc + 1) % 3; // Step #6 - new side
    } else { // dead end, start a new branch
        branch = true; // goto step #7 below
    }
}

if (found || branch) {
    // Step #9 - add the square to the maze
    // sq contains the current square which was either
    // initied in step #2 before the main loop, or set in
    // step #9a below. If found then the nextSide
    // wall is cleared in sq before setting maze[][]
    maze[xx][yy] = (byte) (found ? (sq & ~nextSide) : sq);
    dirtySquare(xx, yy);
    if (displayConstruction)
        showMaze(false);
}

if (branch) {
    // Step #7 - select a square to branch from
    if (count < threshold) {
        // exhaustively search for remaining squares
        sq = BLOCKED;
        SEARCH:
        for (xx = 0; xx < mzWid; xx++)
            for (yy = 0; yy < mzHht; yy++)
                if (sgr(nx, yy) == BLOCKED)
                    for (int i = 0; --i >= 0; ) {
                        nx = xx; ny = yy;
                        switch (dir = ++dir & 3) { //
                            case 0:
                                nx--; break;
                            case 1:
                                nx++; break;
                            case 2:
                                ny--; break;
                            case 3:
                                ny++; break;
                        }
                        if (inBounds(nx, ny) &&
                            (sq = sgr(nx, ny)) != BLOCKED) {
                            xx = nx; yy = ny;
                            break SEARCH; // found sq for branch
                        }
                    }
            }
    }
}

if (sq == BLOCKED)
    break; // maze done

(continued)
if (finishX(0)) // no exit square selected, do it now
    maze[(xx - mzWid - 1) + yy] = rint(mzHyt) & = (byte)-RIGHT;
    dirtySquare(xx, yy);
    if (displayConstruction)
        showMaze(false);
else I // randomly search for a square for new branch
    do i
        xx = rint(mzWid);
        yy = rint(mzHyt);
        sq = sqxx(yy);
    while (sq == BLOCKED);

    // Step #8 - set lastSide to an open side
    for (lastSide = 1; (lastSide & sq) != 0;)
        lastSide <<= 1;
        srcn = 3;
        sqInc = rint(3);
        branch = found = false;

    // Step ##8 - set lastSide to an open side
    while (!found);
    if ((lastSide = nextSide << 2) > BLOCKED)
        lastSide >>= 4;
        sq = sq & -lastSide; // Step ##9a
        xx = nx; yy = ny;
        if (!displayConstruction && (count & 0xFF) == 0)
            Thread.yield(); // give some time to other threads
        I/ Step ##11 - check for more squares (at top of loop)

    if (!displayConstruction)
        repaint();

Notice that each time the code modifies them a ze[] array, it calls
dirtySquare() to tell the display code that it needs to redraw the square.
Also, if displayConstruction is true, the code calls showMaze() to display
the maze after it adds a square to the maze. You call showMaze() to animate
the progress of the maze generation. (The "Displaying a 2-D Maze" section
later in the chapter discusses how showMaze() works.)
Step 7 in `generate()` employs two different strategies for picking a random square from which to start a new branch. When lots of `BLOCKED` squares remain, `generate()` randomly selects squares in the grid until it finds an unblocked square. Using random selection creates more interesting and varied mazes. When the number of `BLOCKED` squares remaining in maze[][] falls below the threshold level - arbitrarily set to 1/8 of the total grid squares - the code uses an exhaustive search to find the remaining `BLOCKED` squares. An exhaustive search quickly adds the remaining squares to the maze, additions that could take a long time to make randomly.

Generating a buck maze

The block maze algorithm maintains a list of squares that it must explore. The algorithm adds squares to the list as it adds them to the maze grid. Each square in the list keeps track of the unexplored directions from itself. Each square also remembers the direction in which the search was proceeding when the particular square was added to the list so that the algorithm can give a preference to continuing to search in the same direction. Each entry in the list is an object of the `Sqr` class shown here:

```java
class Sqr {
    private boolean t, b, l, r;  // top, bottom, left, right
    private int dir;              // direction square entered
    int x, y;                     // coordinates of square

    Sqr (int x, int y, int dir,
         boolean t, boolean b, boolean l, boolean r) {
        this.x = x; this.y = y; this.dir = dir;
        this.t = t; this.b = b; this.l = l; this.r = r;
    }

    int open () {            // open() returns a count of the unexplored directions
        int open () {
            return (t ? 1:0) + (b ? 1:0) + (r ? 1:0) + (l ? 1:0);
        }
    }
}
```

(continued)
The `open()` method in the `Sqr` class returns the number of unexplored directions from the square. `select()` returns an unexplored direction and then marks the direction as explored.

Here are the steps the `BlockMaze` class uses to generate a maze:

1. Set all the squares in the maze grid to `WALL`.
2. Randomly select a noncorner square on the left edge of the maze and set the square to `FLOOR`. Create a `Sqr` object with the `RIGHT` direction unexplored and add it to the list of available squares.
3. If the list of available squares is empty, you're done.
4. Select a square to explore from the list of available squares.
5. If less than two unexplored directions are available from the square, remove the square from the list of available squares. The single remaining direction is the last direction to explore, so you remove the square from the list.
6. Select a direction to explore from the square and mark the direction as explored.
7. Check to see whether you need to add the square in the selected direction to the maze grid.
   
   If the answer is "yes," add the square to the maze and to the list of available squares. When adding the square to the list of available squares, mark all of the directions to explore from the square to the directions that contain adjacent `WALL` squares.
8. Go to Step 3.
The **Block Maze** class implements the abstract methods `isOpen()` and `generate()` defined in the **Maze** superclass. `generate()` in turn uses the private method `tryDir()`. `tryDir()` uses the private methods `blocked()`, `noDiag()`, and `isOpen()`. (Remember, an overloaded version of a method is an alternate method that accepts different parameters.) Here are fields and methods you add to `BlockMaze` to generate mazes:

```java
private Vector pending; // list of available Sqr objects
private int strt = 70; // prob of exploring from same sqr
private int sdir = 60; // prob of exploring in same dir
private int thru = 90; // prob of blocking thru loop
private int side = 60; // prob of blocking wide area
private int diag = 100; // prob of blocking diag connection
private int dens = 15; // prob of leaving areas unexplored

public boolean isOpen (int x, int y)
    return inBounds(x, y) && sqr(x, y) == FLOOR;

private boolean isOpen (int x, int y, int allowProb)
    return (allowProb) && isOpen(x, y);

private boolean blocked (int x, int y) i
    return inBounds(x, y) && sqr(x, y) == WALL;

private boolean noDiag (int x, int y, int dx, int dy) {
    return blocked(x + dx, y) && blocked(x, y + dy) &&
           isOpen(x + dx, y + dy, diag);  

private boolean tryDir (int x, int y, int dir) {
    // Step #7 - check if adjacent square in direction dir
    // should be added to the maze

    switch (dir) {
    case TOP:
        if (isOpen(x, y-1, thru) &&
            isOpen(x-1, y, side) &&
            isOpen(x+1, y, side) &&
            noDiag(x, y, -1, -1) &&
            noDiag(x, y, 1, -1)) return false;
        break;
    (continued)
```
Part 11: Up to Speed

(continued)

    case BOTTOM:
        y++;
        if (isOpen(x, y+1, thru)
            isOpen(x-1, y, side)
            noDiag(x, y, -1, 1) || noDiag(x, y, 1, 1))
            return false;
        break;

    case LEFT:
        if (isOpen(x-1, y, thru)
            isOpen(x, y-1, side) && isOpen(x, y+1, side)
            noDiag(x, y, -1, -1) && noDiag(x, y, -1, 1))
            return false;
        break;

    case RIGHT:
        x++;
        if (isOpen(x+1, y, thru)
            isOpen(x, y-1, side)
            noDiag(x, y, 1, -1) && noDiag(x, y, 1, 1))
            return false;
        break;

    if (finishX < 0 && x == mzWid-1)
        finishX = x; finishY = y; // found exit
    else if (x <= 0 || x >= mzWid-1 || y <= 0 || y >= mZHyt-1)
        return false; // square on border or out of bounds
    else {
        Sqr sq = new Sqr(x, y, dir,
            blocked(x, y-1), blocked(x, y+1),
            blocked(x-1, y), blocked(x+1, y));

        // if pruning density, replace last pending Sqr
        if (pending.size() > 10 && prob(dens))
            pending.setElementAt(sq, pending.size() - 1);
        else // not pruning, add pending Sqr to list
            pending.addElement(sq);

        maze[x][y] = FLOOR;
        dirtySquare(x, y);
        return true;
public synchronized void generate
  (boolean displayConstruction)

  int free, idx;
  // Step #1 - initialize the maze
  clearMaze();
  if (displayConstruction)
    showMaze(true);
  // Step #2 - select and set the starting square
  pending = new Vector();
  maze[startX = 01[startY = rint(mzHyt - 2) + 11 = FLOOR;
  dirtySquare(startX, startY);
  pending.addElement(new Sgr(startX, startY, RIGHT, 
                             false, false, false, true));
  // Step #3 - loop until list of squares is empty
  while (!pending.isEmpty())
    // Step #4 - select a square to explore
    if (prob(sdir))
      idx = pending.size() - 1;
      // continue with last Sqr
    else
      idx = rint(pending.size()); // choose random Sqr
    Sqr next = (Sqr) pending.elementAt(idx);
    // Step #5 - remove square if no more sides to explore
    // Also randomly remove squares to reduce maze density.
    free = next.open();
    if ((free = next.open()) <= 1
        && pending.size() > 10 && prob(dens))
      pending.removeElementAt(idx);
    if (free > 0)
      // Step 4lsa - select a direction to explore
      if (tryDir(next.x, next.y, 
                  next.select(rint(free), prob(sdir))))
        if (displayConstruction)
          showMaze(false);
      // Step #8 - explore another square
      if (displayConstruction),
        repaint();
  // generate()
Part II: Up to Speed

Figure 7-3: Checking a square in tryDir(x, y, RIGHT).

Here is the code from the tryDir() switch statement:

```java
    case RIGHT:
        x++;
        if (isOpen(x+1, y, thru) && isOpen(x, y+1, side) && noDiag(x, y, 1, -1) && noDiag(x, y, 1, 1))
            return false;
        break;
```

The first thing the code does is increment x to adjust the (x, y) coordinates from the current square (square 1 in Figure 7-3) to the new square candidate (square 2). Working from the candidate square, the code calls isOpen(x+1, y, thru) to check whether the square to the right (square 3) is a floor square. Here is the isOpen() method:

```java
    private boolean isOpen(int x, int y, int allowProb)
    {
        return prob(allowProb) && isOpen(x, y);
    }
```

The thru parameter specifies the probability (from 0 to 100) that isOpen() returns true regardless of whether or not the square is open. Because thru is 90 and square 3 in Figure 7-3 is open, there is a 90 percent chance that prob(90) returns true and a 90 percent chance that the first call to isOpen() returns true. If isOpen() returns true, tryDir() returns false and doesn’t use the square.

Even if tryDir() decides not to use the square this time, the square could still be selected when evaluated from another direction. For example, square 2 in Figure 7-3 could be selected when moving to the LEFT from square 3.
Assuming that the first test beats the odds and isOpen() returns false, the next check is isOpen(x, y-1, side) looks to see whether the square to the top (square 4 in Figure 7-3) is open. The side probability is 80 percent, but because square 4 is a wall, isOpen() returns false regardless of the results of the probability test. The next check is for the other side at the bottom (square 5); square 5 is also a wall square, so isOpen() again returns false.

The last two checks are looking to see whether using square 2 creates two diagonally opposed open squares, as shown in Figure 7-4.

The test noDiag(x, y, 1, -1) checks to see whether using square 2 creates diagonally opposed open squares between squares 2 and 6 in Figure 7-3. The noDiag() test is

```java
private boolean noDiag(int x, int y, int dx, int dy) {
    return blocked(x + dx, y) && blocked(x, y + dy) &&
    isOpen(x + dx, y + dy, diag);
}
```

noDiag() first checks that both of the other opposing corners (squares 4 and 3 from Figure 7-3 in this case) are wall squares. If they are, it checks whether the diagonal square (square 6) is open. However, because BlockMaze sets the diag probability to 100, tryDir() never uses a square that creates diagonally opposed open squares.

Although diagonally opposed open squares don't create a faulty maze, they are aesthetically undesirable, at least when viewed from overhead.

Table 7-1 shows all the settings that you can tinker with to change the characteristics of the maze that BlockMaze generates. You change the settings by changing the initialization values of the private fields in the BlockMaze class.
sowing Mazes

You solve a maze by finding a path between two points in the maze. Games need to solve mazes in order to allow computer adversaries to navigate through the maze. This section shows how to implement algorithms to find a path through the maze. Chapter 8 discusses how to use the capability to navigate through mazes to instill "artificial intelligence" in your computer opponents.

Representing the solution

You declare the following two-dimensional array of bytes in the Maze class to keep track of a maze solution:

```java
protected byte[][] path;
```

Each entry in the array records the sides of the corresponding maze square through which the solution passes. You mark a path in the array by setting the bit for the corresponding side. The static final fields LEFT, RIGHT, TOP, and BOTTOM in the Maze class define the bits. For example, if the path enters the left side of the square and exits out the top, the entry in `path[i][j]` is `LEFT | TOP`.

You add the following methods to the `Maze` class to define the methods for traversing the maze:

<table>
<thead>
<tr>
<th>Field</th>
<th>Default</th>
<th>Increasing This Value Generates a Maze That...</th>
</tr>
</thead>
<tbody>
<tr>
<td>start</td>
<td>70</td>
<td>has longer uninterrupted passageways</td>
</tr>
<tr>
<td>sir</td>
<td>60</td>
<td>has straighter passageways with fewer turns</td>
</tr>
<tr>
<td>thru</td>
<td>90</td>
<td>has fewer loops and alternate paths</td>
</tr>
<tr>
<td>side</td>
<td>60</td>
<td>has fewer open areas and fewer areas wider than the passageway</td>
</tr>
<tr>
<td>diag</td>
<td>100</td>
<td>allows fewer diagonally opposed squares like the squares shown in Figure 7-4</td>
</tr>
<tr>
<td>dens</td>
<td>15</td>
<td>has more walls and a lower path density</td>
</tr>
</tbody>
</table>
The classes that extend `Maze` must implement the abstract method `traverse()`. You declare the overloaded `traverse(bool can displaySearch)` method to use as a shortcut for traversing from the start and finish squares of the maze.

**Keeping your left hand on the wall!!**

A relatively simple yet effective strategy for traversing a maze is to keep your left or right hand in contact with the wall as you move through the maze. Keeping your left hand on the wall causes you to take all the left-hand branches. When you reach a dead end, your hand sweeps along the dead-end wall and you start walking back down the path. Figure 7-5 shows traversing a maze while keeping the left hand on the wall.

The left-hand (or right-hand) rule only works reliably for mazes with a single solution between any two points. If a maze has more than one solution, you can end up traveling in an endless circle if you follow the left-hand rule. To see why you can move in an endless circle, consider a hallway with a column in the middle. Your objective in this simple maze is to get from one end of the hall to the other with two solutions: You can go around the column to the left or to the right. If you happen to start your search with your hand on the column, you perpetually walk around the column; your hand never leaves the column, and you never reach the goal of the maze.
The \texttt{W a 1 1 M a z e} class implements the left-hand rule traversal algorithm like this:

```java
public synchronized boolean traverse
(int xx, int yy, int fx, int fy, boolean displaySearch)

if (!inBounds(xx, yy) || !inBounds(fx, fy))
    return false;
int count = 0, sq = maze[xx][yy];
int side = LEFT, sx = xx, sy = yy;
boolean solve = (xx == startX && yy == startY &&
    fx == finishX && fy == finishY);
resetPath();
if (solve) { // mark path to enter maze
    path[sx][sy] = LEFT;
dirtySquare(sx, sy);
    side = TOP;
    while (xx != fx || yy != fy) { // search for direction to try
        if ((side <= 1) > BLOCKED)
            side = TOP;
        path[xx][yy] ^= side; // set exit from current square
dirtySquare(xx, yy);
        switch (side) { // set entrance to new square
            case LEFT: path[--xx][yy] ^= RIGHT; side = BOTTOM; break;
            case TOP: path[xx][--yy] ^= BOTTOM; side = LEFT; break;
            case RIGHT: path[++xx][yy] ^= LEFT; side = TOP; break;
            case BOTTOM: path[xx][++yy] ^= TOP; side = RIGHT; break;
        }
        sq = dirtySquare(xx, yy);
    }
    if (xx == sx && yy == sy && side == LEFT) {
        // we ve searched the entire maze and we re back at
        // the starting square, so there s no solution
        resetPath();
        if (displayPath)
            repaint();
        return false;
    }
    if (displaySearch)
        showMaze(false);
    else if ((++count & 0xFF) == 0)
        Thread.yield();
}
```

if (solve)
    mark path to exit maze
    path[xx][yy] J = RIGHT;
    dirtySquare(xx, yy);
    if (displayPath)
        repaint();
    return true;
}

ll traverse()

Notice that traverse() marks a square in the path using the exclusive-or operator (\(^=\)). Remember that you use the exclusive-or operator to toggle bits: A zero bit becomes a one and a one becomes a zero. This toggling means that the first time traverse() marks a path in a square, the code sets the bit for the path to one. As traverse() backtracks down a dead-end path, this same instruction toggles the one and clears the path to zero, effectively erasing the dead-end path from the path[][] array.

Using breadth-first searching to find the shortest path

You use a breadth-first search to find the shortest path between two points in a maze. Unlike the left-hand rule, a breadth-first search is designed to find the optimal solution in a maze with multiple paths between two squares. Breadth-first searching works by taking one step at a time on each possible path before taking a second step on any path. The search proceeds simultaneously along all paths that have not been pruned. You prune a path (eliminate it from future searching) when the search reaches a dead end or a square that has already been searched. Because all search paths are the same length, you know that when you find the destination square, you've also found the shortest path.

You use a two-dimensional array the same size as the maze grid to keep track of which squares have been searched. You declare and allocate the array like this:

```java
byte[][] graph = new byte[mzWid][mzHyt];
```

First, you initialize each entry in the graph[][] array to zero. Next, set the entry in graph[][] for the square where the search starts to -1. As you search, you record in graph[][] the direction to move in order to return to the previous square on the path. This record creates a backwards-pointing graph that you can follow to get back to the square where the search was begun. Figure 7-6 shows the breadth-first search path through a maze and the resulting graph[][] array.
While searching, you maintain a queue of squares waiting to be searched. After you add a square to the search path and set the corresponding entry in `graph[][]`, you add the square to the queue. Each iteration of the search pulls the next square out of the queue and checks each direction to see whether the search path can be extended to the square in that direction. You add any squares you find to the search path, set the corresponding entry in `graph[][]`, and place them in the queue. This cycle continues until either you find the destination square, in which case you've solved the maze, or the queue is empty, in which case the maze has no solution.

A **queue** is a first in, first out (FIFO) data structure, which means that the first item put into the queue is the first item removed from the queue and that you add items to the queue at one end and take them out at the other end. The term queue means "a waiting line," and like a line at the Department of Motor Vehicles, the first person in line is the first person served.

Here is the breadth-first `traverse()` method for the `BlockMaze` class:

```java
public synchronized boolean traverse(int sx, int sy, int fx, int fy, boolean displaySearch)
{
    if (!inBounds(fx, sy) || !inBounds(fx, fy))
        return false;
    int dir, xx = sx, yy = sy, count = 0;
    int qhead, gtail, qsize = mzWid * mzHyt - 1) * 2;
    short[][] queue = new short[qsize][2]; // 0 = x, 1 = y
    byte[][] graph = new byte[mzWid][mzHyt];
    boolean solve = (xx == startX && yy == startY && fx == finishX && fy == finishY);
    if (displaySearch) 
        resetPath();
    if (solve) 
        path[sx][sy] = LEFT;
    dirtySquare(sx, sy);
    showMaze(false);
```
graph[xl][yl]
queue[0][x0] = (short)xx; queue[0][y1] = (short)yy;
qtail = 0; ghead' = 1;

TRAVERSE:
for (;;) {
    if (qhead == qtail) I. // empty queue: unsolvable maze
        resetPath();
        if (displayPath)
            repaint();
        return false;
    xx = queue[gtail][0]; yy = queue[gtail][1];
    gtail = (qtail + 1) % gsize;
    int gstart = qhead;
    for (dir = TOP; dir <= LEFT; dir <= 1) I
        int ndir = 0, nx = xx, ny = yy;
        switch (dir) I
            case TOP: ny--; ndir = BOTTOM; break;
            case RIGHT: nx++; ndir = LEFT; break;
            case BOTTOM: ny++; ndir = TOP; break;
            case LEFT: nx--; ndir = RIGHT; break;

    if (inBounds(nx, ny) &&
        graph[nx][ny] == 0 && maze[nx][ny] == FLOOR)
        // extend the search path in direction dir.
        graph[nx][ny] = (byte)ndir; // point to prev square
        if (displaySearch) I
            path[xx][yy] += dir;
            dirtySquare(xx, yy);
            path[nx][ny] += ndir;
            dirtySquare(nx, ny);

    if (nx == fx && ny == fy) // found solution
        break TRAVERSE;
    queue[0][x0] = (short)nx;
    queue[0][y1] = (short)ny;
    qhead = (qhead + 1) % qsize;
}

if (displaySearch) I
    if (qhead == qstart) ( // dead end, backtrack
        while (path[xx][yy] == graph[xxl][yl]) I
            path[xx][yl] = 0;
            dirtySquare(xx, yy);
switch (graph[xx][yy]) {
    case TOP:    path[xx][--yy] = (byte)-BOTTOM; break;
    case RIGHT: path[++xx][yy] = (byte)-LEFT;  break;
    case BOTTOM: path[xx][++yy] = (byte)-TOP;  break;
    case LEFT:  path[--xx][yy] = (byte)-RIGHT; break;
}

dirtySquare(xx, yy);

showMaze(false);

else if ((++count & 0xFF) == 0) Thread.yield();

if (displaySearch)
    if (solve) {
        path[xx][yy] = RIGHT;
        dirtySquare(xx, yy);

        showMaze(false);
    }

/* reconstruct path by following graph */

// from finish to start
resetPath();

if (solve)
    path[fx][fy] = RIGHT;
while ((dir = graph[fx][fy]) != -1) {
    path[fx][fy] = (byte)dir;
    switch (dir) {
        case TOP:    path[xx][--yy] = (byte)-BOTTOM; break;
        case RIGHT:  path[++xx][yy] = (byte)-LEFT;  break;
        case BOTTOM: path[xx][++yy] = (byte)-TOP;  break;
        case LEFT:   path[--xx][yy] = (byte)-RIGHT; break;
    }
}

if (solve)
    path[fx][fy] = LEFT;
if (displayPath)
    repaint();
return true;
Notice in the preceding code that after you find a solution, you reconstruct the path by following `graph[][]` from the destination square to the starting square.

**Displaying a 2-D Maze**

The `Maze` class extends `java.awt.Canvas` to give it a display area. You draw the maze by overriding the `paint()` method inherited from `Canvas`. Because most of the work to draw a wall maze or a block maze is the same, the `Maze` class overrides `paint()` to do the drawing. As far as the `paint()` method is concerned, the only difference between the two `Maze` subclasses, `BlockMaze` and `WallMaze`, is in the actual drawing of a square. `paint()` calls `drawSquare()` to draw a single maze square. You have to implement the abstract method `drawSquare()` that `Maze` defines and that `paint()` calls in the classes that extend `Maze`. `BlockMaze` draws its squares as solid blocks. `WallMaze` draws walls between squares and leaves the middle of the squares open, except that `WallMaze` draws squares with walls on all four sides (BLOCKED squares) as solid blocks.

---

**Implementing a circular queue**

The `queue[]` array as a circular queue. A circular queue allows you to continually put squares into one end of the queue and remove squares from the other end without ever running into the end of the `queue[]` array. If you don’t use a circular queue, you have to periodically move the squares from the end of the `queue[]` array back to the beginning. You add squares in the `head` of the queue like this:

```java
4wi-head:0] = (short)nx;
zm-6 :head:1] = (short)ny;
l = (ghead + 1) % qsize;
```

You increment `qhead` to point to the next entry in the queue. The modulo operation `qsize` sets `qhead` to zero if `(qhead + 1)` equals `qsize`. (Remember, the modulo operator returns the remainder of dividing the left-hand operand by the right-hand operand - the remainder of dividing `(qhead + 1)` by `qsize` in this case.) You use similar code to remove the next square from the tail of the queue:

```java
xx = queue[gtail][0];
yy = queue[gtail][1];
qtail = (qtail + 1) % qsize;
```
Using the paint() method

You draw the maze to an offscreen image and then copy this image to the screen. (Chapter I shows how to use an offscreen image and override update() for smooth screen updates.) You add the following paint(): method and supporting fields to the Maze class:

```java
protected Image offscreenImage;
protected Graphics offscr;

public synchronized void paint(Graphics g) {
paintOffscreenImage();
g.drawImage(offscreenImage, 0, 0, this);
notifyAll();
}
```

paint() calls paintOffscreenImage() (another method in the Maze class to do the work of creating the maze image in an offscreen image. You do the work in paintOffscreenImage() rather than in paint() so that the subclasses can override paint() without replacing the code in paintOffscreenImage()). Keeping the offscreen image code out of paint() allows the subclasses to modify how paint() works and still reuse the code in paintOffscreenImage() to generate the maze display-(Chapter 8 shows an example of overriding paint() to display sprites on top of the offscreen maze image.)

The first thing paintOffscreenImage() does is check whether it needs to create the offscreen image and then creates it if necessary.

You allocate the offscreen image from paint() rather than in the Maze() constructor because createImage() can't create the offscreen image unto you place the maze canvas on the screen by adding it to a container. And because you can't place the canvas on the screen until you create it, you port the code that allocates the offscreen image in paint(). The Abstract Window Toolkit (AWT) can't call paint() until you add the component to the screen hierarchy, so createImage() is guaranteed to work when paint() calls it.

Next, paintOffscreenImage() loops through all the squares in the range of dirty squares and draws any squares that have the DIRTY bit set in the square. For each DIRTY square, paintOffscreenImage() calls drawSquare() to draw the maze square and then calls drawPathSquare to draw any path through the square. (The "Displaying a solution" section later in this chapter shows how to implement drawPathSquare().) If the square is the maze's start or finish square, paintOffscreenImage() calls drawTarget() to draw colored circles to mark it.
Finally, you copy the offscreen image to the screen in `paint()` with this statement:

```java
'g.drawImage(offscreenImage, 0, 0, this);
```

The complete `Maze` class, including the `paintOffscreenImage()` and `drawTarget()` methods, is included on the CD.

### Repainting the maze in a thread-friendly manner

Because Java calls `paint()` from a different thread than the thread in which your applet runs, you use `wait()` and `notifyAll()` to wait for `paint()` to draw the maze. (CD Chapter 2 discusses the AWT Interface thread that calls `paint()` and shows how to use `wait()` and `notifyAll()` to control the timing of an animation.) This section shows how you use these techniques to control the animation of a generating maze.

First, notice in the preceding section that before exiting, `paint()` calls `notifyAll()` to wake up any threads that are waiting for the maze to repaint itself. The code waiting to wake up is in `showMaze()`. You call `showMaze()` to repaint the maze and wait for `paint()` to finish. `showMaze()` sleeps long enough after displaying the maze to produce a consistent 30 frames-per-second (fps) animation frame rate. Here is the `showMaze()` method and supporting fields you add to the `Maze` class:

```java
protected long timer, maxFrameRate = 30L; // 30 fps
protected void showMaze (boolean allDirty)
    if (allDirty || offscreenImage == null)
        repaint();
    else
        repaint(leftOffset + minXdirty * sgWid,
                topOffset + minYdirty * sgHyt,
                sgWid * (maxXdirty - minXdirty + 1) + lineWid,
                sgHyt * (maxYdirty - minYdirty + 1) + lineHyt):
            try wait(); catch (InterruptedException e) {} 
            long t = System.currentTimeMillis();
            if ((t = t - (1000L / maxFrameRate)) > 0)
                try (Thread.sleep(t)); catch (InterruptedException e) {} 
            timer = System.currentTimeMillis();
```
Part II: Up to Speed

**Calculating where the pixels go**

You draw the maze squares at pixel locations determined by the size of the Maze canvas and the grid dimensions of the maze. The pixel locations are the same for wall mazes and block mazes, so you place the code to calculate the pixel offsets and sizes in the Maze class. The `drawSquare()` methods implemented by the classes that extend Maze use these pixel locations to draw the maze squares.

You calculate the pixel values in `resetMaze()`, which adjusts the pixel width and height of each rectangle in the grid to fit the screen size at the Maze canvas. Maze overrides the `reshape()` method inherited from Canvas to resize the maze whenever the size of the canvas changes. If `reshape()` changes the size of the canvas, it calls `resetMaze()` to calculate the new size of the grid squares. Figure 7-7 and Table 7-2 show the fields that `resetMaze()` calculates.

![Figure 7-7: The pixel dimensions of a maze square.](image)

### Table 7-2

<table>
<thead>
<tr>
<th>Field</th>
<th>What It Contains</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sqWid, sqHyt</code></td>
<td>The width and height of a grid square in pixels. These include the <code>lineWid</code> and <code>lineHyt</code>, respectively, of the line on one side of the square.</td>
</tr>
<tr>
<td><code>lineWid, lineHyt</code></td>
<td>The pixel width and height of the lines separating grid squares. You set these to zero for block mazes.</td>
</tr>
<tr>
<td><code>sgLnWid, sgLnHyt</code></td>
<td>The pixel width and height of a grid square plus the separating lines.</td>
</tr>
<tr>
<td><code>leftOffset, topOffset</code></td>
<td>The pixel offsets to center the displayed maze within the canvas. These are the canvas pixel offsets of the upper-left corner of the square.</td>
</tr>
</tbody>
</table>
Knowing that block mazes are simple is half the battle

The squares in a block maze are simple colored rectangles. The type of maze square determines the color of the rectangle. Figure 7-1 earlier in this chapter shows an example of a block maze. You implement the drawSquare( ) method for the BlockMaze class like this:

```java
protected void drawSquare(int xx, int yy) {
    offscr.setColor(maze[xx][yy] == WALL ? Color.gray : Color.white);
    offscr.fillRect(leftOffset + (xx * sqWid),
                    topOffset + (yy * sqHyt), sqWid, sqHyt);
}
```

The expression `leftOffset + (xx * sqWid)` calculates the pixel offset of the left edge of the maze square and `topOffset + (yy * sqHyt)` calculates the top pixel offset of the square.

Displaying a wall maze

Each square in a wall maze has 16 different possible combinations of walls, and drawing one such square takes several steps. The first step is to check whether the square has all the walls set, in which case you treat the square as a solid wall. You draw a solid wall by filling the entire square with black. If the square isn’t a solid wall, you start by erasing the square to white and then drawing the four sides and four corners depending on which walls the square has set. Figure 7-8 shows the 8 wall sections that you draw and the 16 resulting wall maze squares.

![Wall Maze Diagram](image)
Notice that in Figure 7-8 all 16 squares draw all 4 corner sections of the maze, which happens because Wa 11 Maze only generates dense mazes with no open areas. If you allow wall mazes with open areas, you only draw a section if the square on the opposite side of a corner that has any walls set that share the corner, or if the corner of the square is on the edge of the maze.

Figure 7-2 earlier in this chapter shows an example of a wall maze.

The complete WallMaze class, including the drawSquare() method, is included on the CD.

To reduce the number of fillRect() calls that drawSquare() has to use in order to draw the maze square, the fillRect() calls that draw the wall sides draw both corners as well. For example, the following code draws the top wall and both top corners (sections 1, 2, and 3 in Figure 7-8) in the square:

```java
if (top - (sq & TOP) != 0)
    offscr.fillRect(xoff, yoff, sgl.nuid-, LineHyt);
```

This code also sets the local variable top to true if it draws the top corner. If drawSquare() doesn't draw the LEFT side of the square, it checks to determine if it needs to draw the top-left corner of the square. If top is false, drawSquare() didn't draw either the TOP or LEFT wall, so it draws the corner. You repeat the TOP check for the BOTTOM wall of the square and repeat the LEFT check for the RIGHT wall.

Customizing the appearance of a wall maze

You can change the look of a wall maze by either setting the pixel width and height of the grid squares to different values or by changing the thickness of the lines that define the walls. For example, the figure shows a maze with squares that are 16 pixels wide, 8 pixels tall, and have walls that are 6 pixels wide and 6 pixels tall.

You control the pixel sizes of the squares by calling resize(width, height) to set the dimensions of the maze canvas, and calling setDimensions(squaresWide, squaresHigh) to set the maze dimensions. To change the default line sizes, you call setLineSizes().
Displaying a solution

A maze stores the current solution path in the path[] array. The bytes in path[] have a bit set for each side of the corresponding square in the maze[][] array that has a solution path. The “Solving Mazes” section earlier in this chapter discusses how you set these bits.

You display the solution by drawing each path segment set in path[]. paintOffscreenImage() calls the drawPathSquare() method in Maze to draw a single path square. Because some subclasses of Maze may want to calculate maze solutions but not display them, paintOffscreenImage() only calls drawPathSquare() if the boolean field displayPath is set.

drawPathSquare() uses pixel sizes and offsets that you initialize in resetMaze(). pWid and pHyt contain the pixel width and height of the displayed path. You set pxoff to the pixel offset within a square of the left edge of a vertical path segment and pyoff to the offset of the top edge of a horizontal segment. You declare these fields in Maze like this:

```java
protected int pWid, pHyt, pxoff, pyoff;
```

You initialize these fields by adding the following code to resetMaze():

```java
int pw = sgWid - lineWid;
pWid = (pw & 1) == 0 ? Math.max(2, (pw >> 1) & -1) : Math.max(1, (pw >> 1) + 1);
int ph = sgHyt - lineHyt;
pHyt = (ph & 1) == 0 ? Math.max(2, (ph >> 1) & -1) : Math.max(1, (ph >> 1) + 1);
pxoff = (sgLnWid - pWid) >> 1;
pyoff = (sgLnHyt - pHyt) >> 1;
```

To center the path in the square, the width of the path must be even if the square width is even and odd if the square width is odd. The following instruction calculates the pixel width of a square:

```java
int pw = sgWid - lineWid;
pWid = (pw & 1) == 0 ? Math.max(2, (pw >> 1) & -1) : Math.max(1, (pw >> 1) + 1);
```

You use the pixel width pw to calculate the path width like this:

```java
pWid = (pw & 1) == 0 ? Math.max(2, (pw >> 1) & -1) : Math.max(1, (pw >> 1) + 1);
```

(pw & 1) == 0 is true if the path width is even and false if it is odd. You set an even path width approximately half the width of the square and at least two pixels wide by using Math. max(2, (pw >> 1)). You calculate an odd path width at least one pixel wide using Math. max(1, (pw >> 1) + 1).
Figure 7-9 shows what the completed path for a wall maze looks like.

**Figure 7-9:** Displaying a solved wall maze.

**Putting the maze on the screen**

You place a Maze canvas on the screen by adding it to a container, such as an applet. This section shows how to implement the MazeApplet class to display the maze and how to use threads to animate the generation and solving of multiple mazes simultaneously. In fact, the applet can even solve the maze while it is still generating it. Figure 7-10 shows a block maze being solved while it is still being generated.

**Figure 7-10:** Solving a partially generated maze.

**Using a thread to animate, generate, and solve a maze**

You spawn a thread to animate, generate, or solve the maze. Because each maze or applet can have more than one thread, you create a thread class to handle the different thread operations.

Because the only function of the class is to execute a thread, you extend the Thread class directly from Thread instead of implementing the Runnable interface.

Depending on the solve parameter passed to the MazeThread constructor, the MazeThread class either generates or solves the maze. Here is the complete MazeThread class:
class MazeThread extends Thread {
    private Maze maze;
    private boolean show, solve;

    MazeThread (ThreadGroup tg, Maze maze,
           boolean show, boolean solve)
        super(tg, solve ? Solve thread : Generate thread
             this.maze = maze;
             this.show = show;
             this.solve = solve;
             start();

    public void run () {
        if (show)
            setPriority(Thread.MIN_PRIORITY + 1);
        if (solve)
            maze.traverse(show);
        else
            maze.generate(show);
    } // class MazeThread

Notice that if the operation is being animated, the run() method sets the thread priority to MIN_PRIORITY + 1. Lowering the thread priority makes the user input and screen updating more responsive.

**Reviewing parameters in the MazeAppl et class**

To make the features of the maze accessible to HyperText Markup Language (HTML), the MazeApplet class accepts certain HTML parameters. CD Chapter 1 discusses how to pass parameters to an applet from HTML. MazeApplet accepts the following parameters:

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>What It Specifies</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 NEWIDTH</td>
<td>The width and height of the lines (L1:newLd, L1:newLy). You only want to specify line width for wall mazes. The line width defaults to zero for block mazes and one for wall mazes.</td>
</tr>
<tr>
<td>MAZEWIDTH</td>
<td>The number of grid squares wide to make the maze. The maze width defaults to 30 squares.</td>
</tr>
</tbody>
</table>
Parameter Name | What It Specifies
--- | ---
MAZEHEIGHT | The number of grid squares tall to make the maze - "maze height defaults to 20 squares."
MAZE | The Maze subclass to use. The default maze class "WallMaze".

For example, you add the following applet tag to your HTML document block maze that is 20 squares wide and 15 squares tall:

```html
<applet code=MazeApplet width=242 height=182>
  <param name= MAZE value= BlockM'aze >
  <param name= MAZEWIDTH value= 20”>
  <param name= MAZEHEIGHT value= 15”>
</applet>
```

This HTML produces a block maze with squares that are 12 pixels wide and 12 pixels tall with a 1-pixel border around the maze. MazeApplet derives these pixel dimensions from the HTML tag like this:

```plaintext
square width = (width / MAZEWIDTH) = (242 / 20) = 12
square height = (height / MAZEHEIGHT) = (182 / 15) = 12
leftOffset = (width % MAZEWIDTH) / 2 = (242 % 20) / 2 = 1
topOffset = (height % MAZEHEIGHT) / 2 = (182 % 20) / 2 = 1
```

MazeApplet generates a new maze when the user clicks the maze with the mouse button. If the user holds the Shift key down while clicking the mouse on the maze and the maze has added the solution square to the maze, MazeApplet solves the maze. `mouseDown( )` spawns a thread to generate or solve the maze. (CD Chapter 2 shows how to spawn threads from event handler `mouseDown( )`.)

The MazeApplet class is included on the CD.
Chapter 8

2-D Sprite Maze

This chapter covers:
- Modeling game elements with sprites
- Managing sprites with a sprite engine
- Displaying and animating sprites
- Detecting and handling sprite collisions
- Meting a 2-D maze game using sprites

A sprite is an arbitrarily-shaped (not necessarily rectangular) graphic object that moves nondestructively across a background. A familiar example of a sprite is the mouse cursor - it can be any shape, and it moves around the screen without changing the screen background. You use sprites when you want to minimize redrawing the background.

Sprites are most useful for 2-D games - particularly arcade games where you have a background and various objects moving over it. You use sprites to represent game elements that move around the screen, although you can also use sprites for stationary game elements. Game elements can be any object in your game: spaceships, bullets, explosions, little men, obstacles, walls, vicious blobs of slime, or a plumber named Mario.

Movable sprites contain code to move across a game background - which brings up the questions of where and how to move the sprite - so we show you how to give your sprites enough intelligence to answer these questions. Of course, moving sprites can run into the boundaries of the background and other moving and stationary sprites, so we show you how to detect and resolve collisions when they occur.

In short, this chapter shows how to create, display, animate, and most importantly, keep track of and manage sprites with a sprite engine. Finally, this chapter puts all the sprite stuff together to make a simple game using sprites and the Block Maze class from Chapter 7.
Gentleman, Start Your Sprite Engines!

Sprite engine is just a fancy term for a data structure that keeps track of sprites and tells them when to perform certain operations such as drawing, moving, or animating. When applied to software, the term engine identifies code that stands on its own and is general enough to be used in a variety of applications. A well-constructed sprite engine (which of course includes the one we present in this chapter) can be extended and used in many games.

A sprite engine manages the sprites in a rectangular play field. You can give your sprite engine all kinds of bells and whistles, but the four primary duties of a sprite engine are to:

- Maintain a list of all the sprites under its control and their position in the play field
- Draw the sprites from back to front
- Move the sprites
- Detect and resolve collisions between sprites or between a sprite and the edge of the play field

Actually, detecting and resolving collisions isn’t a requirement for a sprite engine. In some of your games, sprites may need to occupy the same space in the play field without triggering a collision. However, for many games, collision detection is the most important service the sprite engine provides, so we include it in our list of primary duties. After all, a shoot-'em-up game wouldn’t be much fun if the bullets never hit anything.

To leverage the power of object-oriented programming, your Java sprite engine doesn’t actually draw or move the sprites; it simply tells the sprites when to move or draw themselves. Because collision detection either involves more than one sprite or involves the sprite and the edge of the play field, the sprite engine takes care of detecting collisions. However, the engine just tells the colliding sprites what happened and lets the sprites determine how to resolve the collision. This division of responsibilities makes the code for both the sprites and the sprite engine fairly simple, yet allows you to build games with hundreds of moving and animating game elements that interact with each other.

Implementing a sprite

A sprite has only a few responsibilities:

- It draws itself.
- It updates its state. This usually involves moving and/or animating the sprite, but it could be anything the sprite needs to do periodically.
It defines its collision box. The collision box is a rectangular area that moves with the sprite and functions as the sprite’s area of influence, determining where the sprite can collide with other sprites.

It handles collisions with other sprites and with the edge of the play field.

Because game elements that are sprites may need to extend classes that aren’t sprites, you use an interface to define the sprite methods. Using an interface allows any class that implements the interface to function as a sprite.

An interface is a definition of methods that a class implements in order to do the job that the interface defines.

The sprite interface you use for the game in this chapter, as well as for many other games, is quite simple. You declare the Sprite interface in the file Sprite.java like this:

```java
import java.awt.*;
public interface Sprite {
    void setSpriteEngine(SpriteEngine se);
    boolean updateSprite();
    Rectangle drawSprite(Graphics g);
    Rectangle collisionBox();
    Rectangle collideWith(Object obj);
}
```

The sprite engine uses the Sprite interface methods to manage sprites. Table 8-1 shows the responsibilities for each of the five methods the Sprite interface defines.

<table>
<thead>
<tr>
<th>Method</th>
<th>What the Sprite Engine Expects It To Do</th>
</tr>
</thead>
<tbody>
<tr>
<td>setSpriteEngine()</td>
<td>The sprite engine passes this method a reference to the engine when the sprite is added to the engine, and calls it with null when the sprite is removed from the engine. The sprite uses the reference to call methods in the sprite engine. This method is the sprite's heartbeat. The sprite engine calls this method periodically to tell the sprite to update its state. You move, animate, and initiate actions in updateSprite(). updateSprite() returns true if you change the sprite's collision box, false if you don't.</td>
</tr>
</tbody>
</table>
Part II: Up to Speed

Table 8-1  (continued)

<table>
<thead>
<tr>
<th>Method</th>
<th>What the Sprite Engine Expects It To Do</th>
</tr>
</thead>
<tbody>
<tr>
<td>drawSprite()</td>
<td>The sprite engine passes this method a graphics context in which to draw the sprite. It returns a <code>Rectangle</code> representing the region drawn to the screen or <code>null</code> if it didn't draw anything.</td>
</tr>
<tr>
<td>collisionBox()</td>
<td>This method returns a <code>Rectangle</code> containing the sprite's collision box. The sprite engine uses the collision box to determine whether the sprite collides with other sprites. <code>collideBox()</code> returns <code>null</code> to indicate that this sprite doesn't collide with other sprites.</td>
</tr>
<tr>
<td>collideWith()</td>
<td>In the event of a collision, the sprite engine passes this method the object with which the sprite collided. After resolving the collision with whatever action is part of your game, <code>collideWith()</code> returns the possibly changed collision box for the sprite.</td>
</tr>
</tbody>
</table>

The object parameter that the sprite engine passes to the sprite's `collideWith()` method is usually another `Sprite`, but can also be one the `SpriteBorder` constants- NORTH, SOUTH, EAST, or WEST-that `SpriteEngine` defines to represent collisions with the appropriate edge of the play field. `(SpriteBorder is an empty class that `SpriteEngine` uses to define these object constants so that `collideWith()` can test for a border collision using the test `obj instanceof SpriteBorder`). `collideWith()` resolves the collision, which may include changing the position of the sprite. `collideWith()` returns the new collision box or `null` if the sprite engine man doesn't check for any more collisions with a given sprite during the current update.

Putting sprites in their place

Although some `Sprite` classes may need to extend a non-`Sprite` class, the main function of a `Sprite` class is usually just to be a sprite. In addition, the methods defined by the `Sprite` interface all have logical default implementations. These two features of sprites often enable you to encapsulate the common code shared between sprites and reduce the amount of code you have to write for each new sprite you create by giving these sprites a common superclass.

The `SpriteObject` class is exactly this superclass; it implements the `Sprite` interface. The code to keep track of a sprite's position and collision box is pretty standard for all sprites, so you can implement it in `SpriteObject`. Here is the `SpriteObject` class:
import java.awt.*;
public class SpriteObject implements Sprite I
    protected double x, y; // center of sprite
    protected int width, height;
    protected SpriteEngine spriteEngine;
    public SpriteObject (double x, double y, int w, int h) I
        this.x = x; this.y = y; width = w; height = h;
    
    public double centerX () { return x; }
    public double centerY () { return y; }
    public int spriteWidth () { return width; }
    public int spriteHeight () { return height; }

    public void setSpriteEngine (SpriteEngine se)
        spriteEngine = se;

    public boolean updateSprite () I return false; I
    public Rectangle drawSprite (Graphics g) I return null; I
    public Rectangle collisionBox () I
        return new Rectangle((int)(x - width / 2.0),
                            (int)(y - height / 2.0),
                            width, height);

    public Rectangle collideWith (Object obj)
        : return collisionBox());
// end class SpriteObject

Chapter 8: 2-D Sprite Maze

Sprite objects position sprites with an anchor point at the center of the sprite image. Using the center of the sprite image for the anchor point is a decision you need to make based on the kinds of actions your sprites need to be able to perform. You can position sprites from any of the nine locations shown in Figure 8-1. The width and height of a sprite may change as it is animated, but the anchor point continues to dictate how the sprite positions itself. You need to choose the anchor point based on how the sprite anchors itself to the background. We chose the center position as the default because without any other selection criteria, the center is the best choice.

However, some sprites may require anchor points at locations other than the center of the sprite image. For example, an explosion sprite animates from a small explosion image to a large explosion image. You want the explosion to grow out from the center, so you use a center anchor point. On the other hand, if your sprite represents a side view of a worm on the
Part II: Up to Speed

Figure 8-1: The nine ways to position a sprite's anchor point.

ground, you probably want to use a bottom anchor point to anchor the worm to the ground. Using a bottom anchor point, the sprite expands and shrinks from the center as its width changes, but keeps the bottom of the worm anchored to the ground as the height changes. Conversely, you probably want to use a top anchor point when you have a sprite that crawls across a ceiling.

Moving sprites around the play field

Notice that SpriteObject uses double values for the x and y position the sprite. Given that you can only draw images at integer pixel locations, you may wonder why x and y are floating-point values. The answer lies in the fact that using floating-point values results in much smoother movement. Chapter 1 shows how you use floating-point coordinates and floating-point delta x and y values to smoothly move objects at any speed and in any direction.

You give your sprites motion by changing their position in the method updateSprite(). Here is an example of how you add simple vector motion to your sprite:

```java
protected double deltaX, deltaY; // the vector deltas
public boolean updateSprite () { 
    x += deltaX;
    y += deltaY;
    return true;
}
```

Notice that the movement code doesn't need to do any checking to see whether the object moves out of bounds or runs into something because the sprite engine handles all the collision detection. All you have to do is move the sprite.
Resolving collisions

The sprite engine takes care of detecting collisions, but the sprite itself is responsible for handling what happens as a result of the collision. When the sprite engine detects a collision, it calls the `collideWith()` methods for each sprite involved in the collision. You implement `collideWith()` to resolve the collision. (The "Implementing a sprite" section earlier in this chapter discusses how the sprite engine calls `collideWith()`.)

The `ObjectDetector` class extends `SpriteObject`. It detects when a specific object has collided with it and then notifies the sprite engine's observers. (The observers are other objects, such as the `SpriteMaze` game presented in the "Building on the `BlockMaze` class" section later in this chapter, that receive messages from the sprites in the engine.) You can trigger an event when a sprite reaches a location in the play field by adding an `ObjectDetector` sprite for that location to the sprite engine. The section "Sprite events and handling them" later in this chapter discusses how the notification process works. Here is the `ObjectDetector` class:

```java
import java.awt.*;

class ObjectDetector extends SpriteObject {
  private Object target;
  ObjectDetector(int x, int y, int w, int h, Object target) {
    super(x, y, w, h);
    this.target = target;
  }

  public Rectangle collideWith(Object obj) {
    if (obj == target) {
      spriteEngine.notifyObservers(this);
      return collisionBox();
    }
  }
}
```

You can use sprite collision detection to do proximity detection. Proximity detection is when you want a sprite to know when something is close to it before it collides. You use proximity detection as an early warning system to allow your sprite to change course, initiate defensive maneuvers, or launch an attack. You give a sprite proximity detection by adding a slave sprite with a `collisionBox()` that defines the detection perimeter around the master sprite. The slave sprite only needs to implement the `collisionBox()` and `collideWith()` methods; you leave the other methods empty like the corresponding methods in the `SpriteObject` class.
Displaying sprites

You display a sprite by implementing the Sprite method `drawSprite()`. The sprite engine passes `drawSprite()` the graphics context in which to draw the sprite. As an example, the following `RoundSprite` class draws a colored oval:

```java
import java.awt.*;

class RoundSprite extends SpriteObject {
    protected Color color;

    RoundSprite (double x, double y, int w, int h, Color c) {
        super(x, y, w, h);
        color = c;
    }

    public Rectangle drawSprite (Graphics g) {
        g.setColor(color);
        g.fillOval((int)(x - width / 2.0),
                   (int)(y - height / 2.0), width, height);
        return collisionBox();
    }
}
```

Notice in the call to `fillOval()` that `drawSprite()` translates the sprite's center anchor point to the upper-left corner of the sprite before drawing to match the upper-left coordinates that `fillOval()` expects.

Drawing plain old colored geometric shapes has its place, but what you really need in order to give your game visual appeal are some colorful images zipping around the screen. Here's an `ImageSprite` class that you use to create sprites from loaded images:

```java
import java.awt.*;

class ImageSprite extends SpriteObject {
    protected Image image;

    ImageSprite (Image image, double x, double y) {
        super(x, y, 0, 0);
        setImage(image);
    }

    public void setImage (Image img) {
        image = img;
        width = img.getWidth(null);
        height = img.getHeight(null);
    }
}
```