
Next Wave °f java-Fueled
Catch the
Internet Gaming
45
MILLION
DUNRUES BOORS
IN PRINT
JAVA™ GAME PROGRAMMING FOR DUMMIES®
Explains
More Than
Ten Games!
A Reference for the Rest of Us!'
by Wayne Holder
& Doug Bell
The Fun and Easy Way' to Create Your Own Games and Put Them on Your Web Page
Your First Aid Kif for Adding Pizzazz to Boring Web Sites
Creating Cool Games in Java - Explained in Plain English
IDG
BOOK
WORLDNN"ID

maxiwE

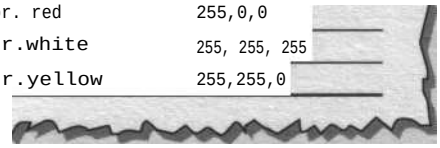
`ova wtne Programming For Dummies`

<APPLET> Tag Attributes

Attribute	*Value*
CODE	***Required:*** **The name of the Java class file containing the compiled** Appl et **subclass to execute.**
WIDTH, HEIGHT	***Required:*** The ***suggested*** **pixel width and height of the area the browser should reserve for the applet in the Web page.**
CODEBAS E	**The uniform resource locator (URL) of the directory or folder that contains the applet code.** If CODEBAS E is **not specified, then the Web browser viewing the document defaults to the location of the HTML document.** COD EBAS E **allows the applet code to be place in a different location than the HTML.**
NAME	**The applet name that other applets on the Web page can use to find it and communicate with it.**
ALT	**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.**
ALIGN	**The alignment of the applet relative to the text line containing it. This attribute works like the** ALI GN **attribute for the** I MG **tag. The possible values are** top, mi ddl e, bottom, 1 eft, and ri ght. **The alignment is** bottom **by default.**
HSPAC E , VSPACE	**The number of pixels of space the browser should leave around the applet on the left and right** (HSPAC E) **and top and bottom** (VSPACE).

Built-in Java Colors

Color	*RGB values*
Color.black	0,0,0
Col or.blue	0,0,255
Color.cyan	0,255,255
Color.darkGray	64,64,64
Col or. gray	128, 128, 128
Color.green	0,255,0
Color.lightGray	192,192,192
Color.magenta	255,0,255
Color.orange	255,200,0
Col or. pi nk	255, 175, 175
Col or. red	255,0,0
Color.white	255, 255, 255
Color.yellow	255,255,0



Commonly Overridden Applet Methods

Applet Method	*Override It To . . .*
void init(	**Perform any one-time initialization the applet needs before it runs.**
void start()	**Begin animations, processing, or threads.**
void paint (Graphics g)	**Draw the applet to the screen.**
void stop()	**Suspend animations, processing, or threads initiated in** start( ).
void destroy()	**Clean up after the applet before it quits.**

Cheat Sheet $2.95 value. Item 0168-2.

For more information about IDG Books, call 1-800-762-2974.

IDG BOOKS WORLDWIDE

...For Dummies: #1 Computer Book Series for Beginners

Computer BOOK SERIES FROM IDG

ava?M Some ProgramM 4

For Dstmmiei

Cheat Sheet

Drawing Outlined Shapes and Lines

Shape Outline	Method and Parameters
Rectangle	drawRect(int x, int y, int width, int height)
3-DRectangle	draw3DRect(int x, int y, int width, int height, boolean raised)
Rounded Rectangle	drawRoundRect(int x, int y, int width, int height, int arcWidth, int arcHeight)
Oval	draw0val(int x, int y, int width, int height)
Arc	drawArc(int x, int y, int width, int height, boolean raised, int startAngle, int arcAngle)
Polygon	drawPolygon(int[] xPoints, int[] yPoints, int nPoints) or, drawPolygon(Polygon poly)
Line	drawLine(int x1, int y1, int x2, int y2)

Drawing Filled Shapes and Text

Filled Shape	Method and Parameters
Rectangle	fillRect(int ~, int width, i , - ~~ -
3-D Rectangle	fi113DRect(int x, int int width, int -eigih^I- boolean raised)
Rounded Rectangle	fillRoundRect(ir- y, int width, int int arcWidth,int arcHeight)
Oval	fillOval(int x, int width, int heigh
Arc	fil]Arc(int x, int y, int width, int height, boolean raised, int startAngle, int arcAngle
Polygon	fillPolygon(int[] xPoints, int[] yPoints, int nPoints) or: fillPolygon(Polygon poly)
Text String	drawString(String str, int x, int y)

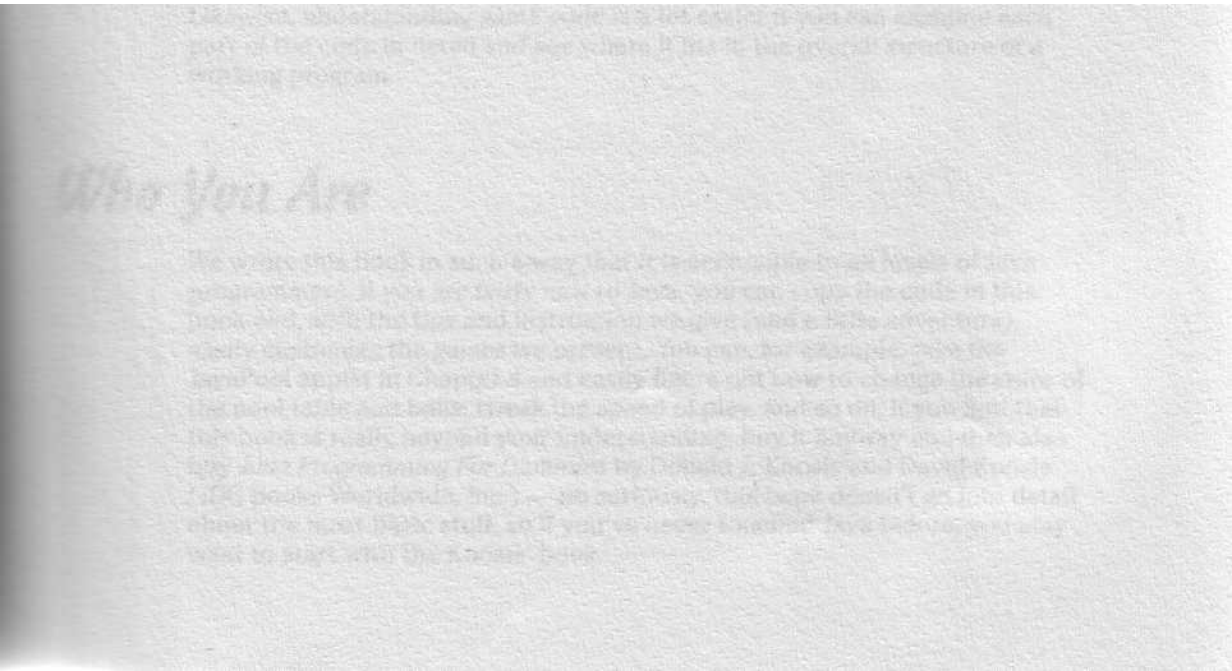
Useful HTML Tags

Tag	Example Usage	Description
	<A -;FF=http: //www. id c~^ -ks.co m>IDGBooks</A>	The anchor tag creates a link to another document or Web page, in this case the IDG Books Web site.
	APPLET <APPLET CODE=MyAppl et WIDTH=80 HEIGHT=50></APPLET>	Insert a Java applet, in this case an applet with the filenameMyApplet.
	<IMG SRC="image.gif">	Insert aGIForJPEGimage. IMGdoesn't require anendtag.
	<P>This is anew paragraph< / P>	Starts anew paragraph. An end tag </P> is not required, but is good practice.
==',T	<FONT SIZE=5 COLOR=RED> Big Red Text< / FONT>	Set the font size and/or color of the contained text.
	<TT>Monospaced text< /TT>	The teletype tag displays the contained text using monospaced text.
	< I >Italic text< / I >	Italicize the contained text.
	< B>Bold text< / B>	Display the contained text with a bold face font.
	<U>Underlinedtext</U>	Underline the contained text.

. . For Dummies: #1 Computer Book Series for Beginners

Table of Contents

Table of Contents
I.....*0000,000*0.*..*000.0.0 *&0*.0000



Java Game Programming For Dummies

Introduction

.....,.......,..0..0.*0..*.00..x.0*

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 anyplace 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.

1cans used in This Book

-e_ k

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.

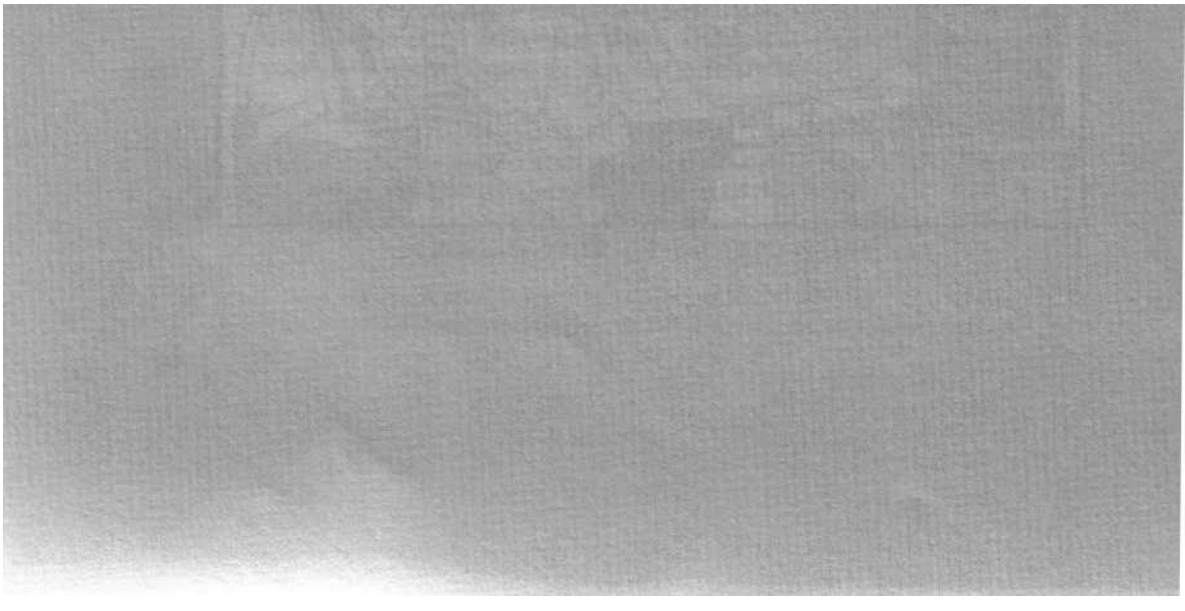
Part I

Steppin' Out

WE. SHOULD NAVE T+ i5 ~iED IN VERSION 2"

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.



Chapter 1

Follow the Bouncing Ball

0*0*0o0 0 0 00000#*0 0 ®0 s0*0 s00 ss0 **0 0 **0 0s0000 s0 0 0 0** 0

hr *This Chapter*

Making things animate

Modeling motion

Handling boundary collisions

p Reducing flicker with double buffering

i 0 0 0 0 0 0 . . . • 0 . . 6 . . . • 0 0 0 . 0 4 . 0 . • • b t . • 0 0 . • ! . • 0 0 . 0 0 .

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 '/so 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 fl-me Programming For Dummies CI)ROM.

Trchia

O f f the Time

Java's Th read class lets you easily construct a program that slices time into tiny intervals using method s 1 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

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

   public void run () {
     while (running) {
        repaint();
```

(continued)

(continued)

```
        try I
          ticker.sleep(1000 / 15);
          catch (InterruptedException e) I }


    public synchronized void start () ['
      if (ticker == null   |·   !ticker.isAlive()) i
        running ° true;
        ticker = new Thread(this);
        ticker.setPriority(Thread.MIN_PRIORITY + 1);
        ticker.start();


    public synchronized void stop () [
      running = false;
    I
```

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.

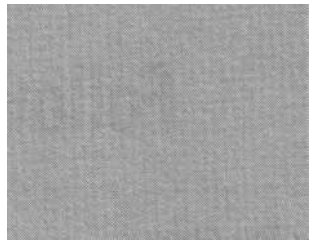
Your animation code needs to respect the applet's life cycle as described in the previous paragraph; otherwise the animation can continue to run - even after the user leaves the page containing your applet - and waste CPU cycles.

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.

(continued)

```
        try
          ticker.sleep(1000 / 15);
        catch (InterruptedException e) { I
      |
```



```
    public synchronized void start () f
      if (ticker == null  ¡.  !ticker.isAlive()) I
        running = true;
        ticker = new Thread(this);
        ticker.setPriority(Thread.MIN_PRIORITY + 1);
        ticker.start();


    public synchronized void stop t)
      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 *delta x* and *delta y* values (the Greek letter delta [0] is used in math and physics to indicate the *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 `float`-type rather than `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 `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 (`float`) values for fractional calculations is quicker.

Encapsulating the essence of a ball

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:

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

(continued)

(continued)

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


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

The constructor for Ba 1 1 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, coI or, and si z e, respectively.

B a 1 1 also defines a d r a w ( ) method that you can call to make the ball draw itself to a G r a p h i c s context. The code is really not much more than calls to s e t C o l o r ( ) and f i l l 0 v a l ( ), but note that the f l o a t values x and y must be cast to i n t in the call to f i 1 10 v a 1 ( ) 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 B a 1 1 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 W I DT H and H E I GHT attributes in the < A P P L ET> 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()`

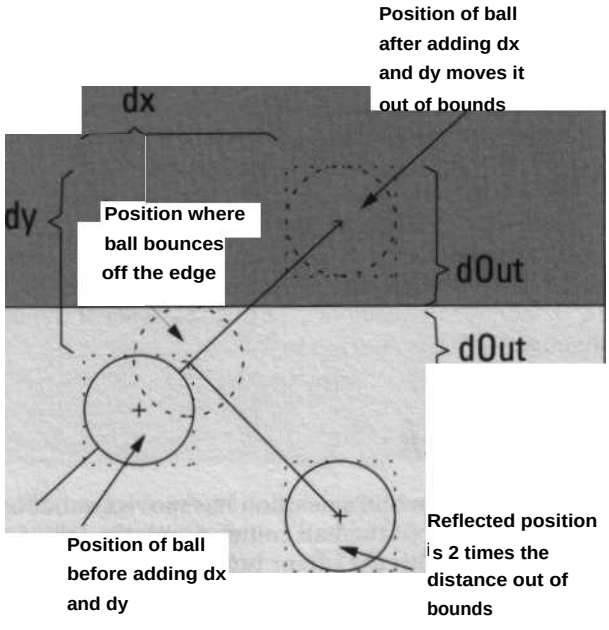


Figure 1-1: After detecting a collision at the top of the boundary, you move the ball back in bounds by moving it twice the distance it traveled out of bounds.

```
public void move (Rectangle bounds) I                    get ,
  // Add velocity values dx/dy to position to
  // 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) &&
           d x > g' )
    dx = -dx;
    x -= 2 * ((x + size) - (bounds.x + bounds.width));

  // Check for collision with top edge
  if (y < bounds.y && dy < 0) I
    dy = -dy;
    y -° 2 * (y - bounds.y);

  // Check for collision with bottom edge
```

```
else if ((y + size) > (bounds.y + bounds.height) &&

  dy = -dy;
  y -= 2 * ((y + size) - (bounds.y + bounds.height));
```

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 B a 1 1 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:

```
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);
```

The new instance of Ba11 is saved in the variable ba11. The starting position for ba11 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 ba11 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 ba11 move. Here's the code for a paint() method that draws a 2 x 2 checkerboard pattern for the background, animates ba11 by calling move(), and draws ba11 into the background by calling draw()

```
public void paint (Graphics g) f
  if (offscr == null) {
    offscreenImage = createImage(width, height)
    offscr = offscreenImage.getGraphics();
  I
  1/ 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.drawlmage(offscreerlmage, 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 offsereen

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 `fill1Rect()` calls that aren't prefixed with the `Graphics` context `g` that you normally use. Instead, the `paint()` method starts by creating an *offscreen Image,* like this:

```
if (offscr == null) i
  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:

```
offscreenimage = createImage(width, height);
```

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

```
offscr - offscreenlmage.getGraphics(l;
```

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:

```
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 I m a g e, the u p d ate () method screen clear is unnecessary.

To remove the screen clear caused by the default version of u p d a t e ( ), you write a new update ( ) method in your applet. Your new u p d a t e() omits the screen clear code and, instead, simply calls p a i n t ( ), like this:

```
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 y 2 that are the center points of the offscreen I ma g e. The code then uses the values for x2 and y2 to draw the upper-left and lower-right sections in g r a y, and the upper-right and lower-left sections in w h i t e.

Next, the code calls the methods in B a l l to move b a l l to its new position and draw ball onto the offscreen Image, like this:

```
ball.move(bounds);
ball.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:

```
g.drawimage(offscreenlmage, 0, 0, null);
```

Figure 1-2 shows your completed applet in action.

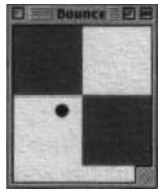


Figure 1-2: The completed Bounce applet.

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.

Chapter 2

Ponglet

a . . . a . 0 . . . 0 . & . a . . . . . a . . . 0 . 0 0 0 . a a a 0 0 v a

This Chapter

Designing with state

Using the mouse

Keeping score

Creating a computer opponent

a a a 0 0 a.O *o **asaa0 ssa0 asa0** 0 0 0 0 9 a0 0 0 9 0 a. 0 0 .s. 0 a0 s0 * w

d Often, the hardest thing to do when creating a computer program is to decide how to organize all its different actions. You know that each separate action is really quite simple by itself, but making them all work smoothly together can be confusing. Fortunately, managing all those program actions is actually fairly easy.

And, to prove it, in this chapter you're going to create your own Ping-Pong applet (Ponglet), complete with mouse controls, score display, and a computer opponent. The techniques you use to work with and organize the different actions in the Ponglet game are equally useful for many other games you may create.

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 Came 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, playerl scores, player2 scores, playerl 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 p a i n t ( ) and the control code in r u n ( ) 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 r u n ( ) is going to be the most complex, so you want to think out a clean way to organize it. Using a swi tch 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 w h i 1 e (r U n n i n g) loop that uses s l e e p ( ) to set the animation frame rate. Here's the complete r u n ( ) framework for Ponglet:

```
public void run () I
  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 PW0N:
    case GWON:
      break;

    repaint();
    try I
      ticker.sleep(1000 / 30):
    } catch (InterruptedException e) { ; 1
```

Note that there is no break between the PW0N and GW0N states because you want your code to do the same thing for both states. Therefore, when the switch statement selects the PW0N 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 c a s e statements in this code. The key to dealing with these c a s e statements is in the code that you add to complete the c a s e 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 c a s e 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:

```
private static final int WAIT = l;
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 `run()` 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:

```
case WAIT:
   if (!mouse in)
     delay = 20;
   else if ( delay < 0) {
     // Serve the ball
     int sLoc = rndlnt(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_in)` 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 `else 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 Bal 1. This code is nearly identical to the Ba11 class in Chapter 1, except that it only checks for bounces off the left and right bounds. Here's the complete code:

```
class Ball t
  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) i

    this.y = x;
    this.y = y;
    this.dx = dx:
    this.dy = dy;
    this.color = color;
    this.size = size;
    radius = size >> 1;


  public void move (Rectangle pd) f
    // Add velocity to position to get new position
    x += dx;
    y += dy:
    // Check for collision with bounding Rectangle
    i f ((x < bd.x && dx < Of)   ||
        ((x  + size) > (bd.x + bd.width) && dx > Of))
      x += (dx = -dx):


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

The code in WA I T that serves the ball also calls two new methods rndInt() and rnd() that generate random int and float values. WAI T 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 rndInt(nn) method generates a random int (integer) between 0 and nn. The rnd(nn) method generates a random float (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 rndlnt (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 P a dd 1 e 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
    th'is.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 )> l)))
      this.x = x;


  public int checkReturn (Ball ball, boolean plyr,
                          int rl, int r2, int r3) I
    if (plyr && ball.y > (y - ball.radius)
        !plyr  &&  ball.y < (y + ball.radius))
      if ((int) Math.abs(ball.x - x) < (width 1 2 +
                                        ball.radius)) I
        ball.dy = -ball.dy;
        1! Put a little english on the ball
        ball.dx += (int) (ball.dx * Math.abs(ball.x - x) /
                          (width / 2));
        return r2;
      I
      else
```

```
        return r3;
    }
    return rl;
}

public void draw (Graphics g) f
    g.setColor(color);
    g.fillRect(x - (width >> 1), y, width, height);
```

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

```
ball.y > (y - ball.radius)
```

and the code for the computer's paddle is

```
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

```
(int) Math.abs(ball.x - x) ( (width / 2 + 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:

```
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:

```
              ▪ ~)a , r.0,~- to )
      · n C1 -1· / 2))
```

You can play with the (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.

```
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:

```
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

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:

```
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:

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

He shoots, he scores!

After scoring a point, the PSC0RE 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 PW0N. 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:

```
case PSCORE:
  // Increment player s score and check if she has won
  gstate = (++pScore >= MAX_SCORE ? PW0N : 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 PW0N, else it sets gstate to WAIT.

If the computer scores, the code in the GSC0RE case is nearly identical:

```
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 win_show to a value of 100. The code also counts down wi n show's value in the code for the PW0N and GWON states. The code is the same for both states, so the case statements fall through, like this:

```
case PW0N:
case GW0N:
  // Delay while we show who won
  if ( win show < 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:

```
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()`

```
Public boolean mouseExit (Event evt, int x, int y) l
  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:

```
 ublic boolean mouseMove (Event evt, int x, int y)
  pPaddle.move(x, table);
  return true;
```

1Jispla ying the state

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:

```
public void paint (Graphics g) I
  if (offscr == null) I
    offscreenImage = createlmage(width, height)';
    offscr = offscreenImage.getGraphics();

  / Fill offscreen buffer with a background B/W checkerboard
  nt x2 = table.width >> 1;
  nt y2 = table.height >> l;
  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(offscreenlmage, 0, 0, this);
```

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

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()

```
public void init() {
  width = size().width;
  height = size().height;
  // Set up table and mouse control area dimensions
  table = new Rectangle(width, width);
  msePad = new Dimension(width, height - width);
```

```
pPaddle = new Paddle(width » l, table.height -  fi, 20.
                    3, 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 code on the screen locations given by the `Point` objects `game` and `player`. 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:

```
if ((gstate & (PW0N | GW0N)) != 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 e 1 s e statement that follows the code to declare a winner:

```
else
     // Draw ball
    if ((gstate & (SERVE | RETURN | PGUTTER)) != 0)
       ball.draw(offscr);
    I/ Draw player s paddle
    if (mouse-in  ¦¡  |gstate & (SERVE | RETURN.`  PGUTTER | GGUTTER)) 1= 0)
      pPaddle.draw(offscr);
    fl Draw computer s paddle
    ¡f (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);
¡f (!mouse-in) I
  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.

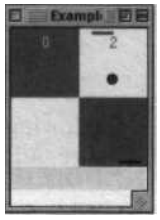


Figure 2-1: Here's how the completed Ponglet game looks.

Chapter 3

Hole In One

6 Eius Chapter

- dating golf
- Making a click-and-drag putt interface
- l g the physics of a ball

erhaps 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 `Ho1eIn0ne` applet.

This chapter describes all the techniques used in the `Ho1 e I n0ne` applet. The complete code and ready-to-use applet is included on the *Java Came 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 land 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

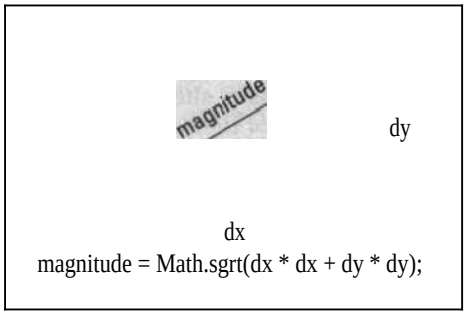
A vector (not to be confused with the j a v a . u t i 1 class of the same name) is another name for a pair of dx and dy values. You can think of a vector as representing the difference between two points; the dx and dy values form a

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

math.sgrttdx * dx + dy * dy)

The length of a vector's diagonal is commonly called the `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.

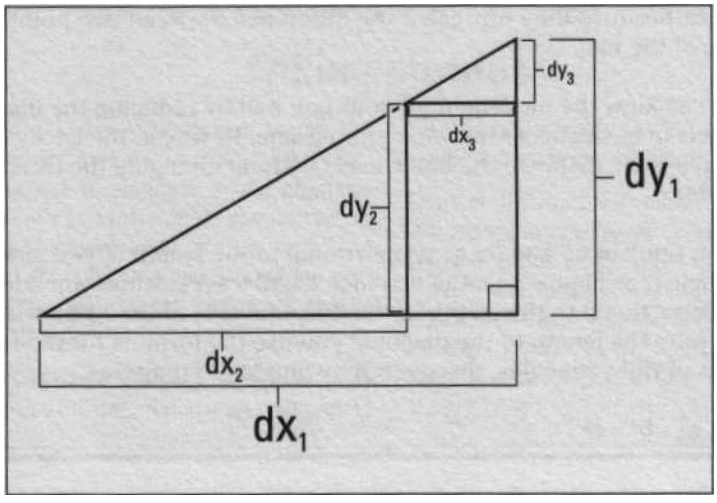


Figure 3-2: To increase or decrease the magnitude of a vector, proportionally change the vector's dx and dy values.

The biggest triangle, with sides dx,/dy,, 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 dx2/dye, you have to subtract the lengths of the sides of the smallest triangle, dx3/dy3, from the sides of dx,/dy,.

However, notice that you subtract more from dx, than you do from dy,. If you were to subtract the same amount from dx, and dy ℮, 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 and dy.

The secret to keeping the ratio of dx and dy the same is to divide dx and dy by the magnitude of the vector formed by dx and dy:

```
dx = dx / Math.sqrt(dx * dx + dy * dy)
dy = dy / Math.sqrt(dx * dx + dy * dy)
```

Dividing by the magnitude changes dx and dy into a *unit vector* (see the following techno term icon), with values for dx and dy 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 and dy by the same number, you maintain the same ratio between dx and dy and therefore the same direction for the ball. For example, when the ball's movement is completely vertical (dx = 0), the value of dy 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 V e c 2 D, like this:

```
public class Vec2D I
  public float  dx, dy;

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


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


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


  public void subVec (Vec2D vec) I
    dx -= vec.dx:
    dy -= vec.dy;
```

(continued)

(continued)

```
    public void unitVec () f
      float mag = mag();
      setVec(dx / mag, dy / mag);


    public void mulVec (float scale) t
      setVec(dx * scale, dy * scale);
    1
```

Vec2D defines internal dx and d y values so that you can create a V e c 2D object whenever you need to keep track of a vector quantity. For example, you can create a V e c 2D object called v e 1 to control the motion of your golf ball. Then, after you have this V e c 2D object, you can call the methods on it, such as SetVec() to set the direction and speed of the ball by setting the d x and dy values in vel.

The methods addVec( ) and subVec ( ) in 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 unitVec() converts a vector into a unit vector. You can use this method in combination with m u 1 Vec() to proportionally scale the d x and dy values in a vector by the f 1 oat parameter passed to it by m u 1 V e c ( ) . Notice that u n i t V e c ( ) is coded to use the method m a g ( ) 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 C i r c 1 e to contain the code that is common to both a hole and a ball?

Creating the C i r c 1 e class

Your C i r c 1 e class needs to define x, y, and d i a m 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 Ba11 class, you need the constructor to initialize a value for the r a d i u s 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 d i s t ( ) . Here's the complete code for Ci rcl e:

```
class Circle i
  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.sgrt((xSq * xSq) + (ySq * ySq));
```

Building a B a 11 by extending C i r c 1 e

Next, you can extend C i r c 1 e to create a new class called B a 11. You put code in B a11 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 Vec2D 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();
   ,'vate Vec2D       tvec = new Vec2D();
  _._- = boolean      sunk = false;

  ::    ;'nt x, int y, int diam) [
    . :-r(x, y, diam);
```

In addition to the constructor, this code defines two Vec2D objects: vel and t v e c. The code for the Golf game uses v e 1 to hold the ball's d x and dy values and t v e c as a temporary variable to do your deceleration calculations. You use the boolean flag s u n k to keep track of when the ball falls into the hole. In addition, you use s u n k in both B a 1 1 's d r a w ( ) 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 d e c e 1 ( ). d e c e 1 ( ) takes a single parameter called v a 1 that specifies the amount you want to subtract from the vector's magnitude.

Your dece1 ( ) code needs to start by checking that the magnitude of the ball's v e 1 vector isn't less than v a 1. If v e 1 is less than v a 1, the ball has slowed so much that if you were to further slow down the ball by subtracting the amount in v a 1 from v e 1, 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 v a 1 (meaning that the ball has not yet rolled to a stop), you can go ahead and do the deceleration calculation, like this:

```
public void decel (float val) f
  if (val >-- 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 t v e c. It **initializes by calling s** e t V e c ( ) and passing v e 1 's d x and dy values, which gives you a copy of v e 1 in t v e c.

Next, you call the u n i t V e c ( ) method to convert t v e c into a unit vector. Then you call muI Vec ( ) to shrink this vector down to the same magnitude as v a 1. Finally, you subtract this scaled vector from v e 1 by calling subVec().

In addition to the constructor, this code defines two V e c 2 D objects: v e 1 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 s u n k to keep track of when the ball falls into the hole. In addition, you use s u n k in both B a 1 1 's d r a w ( ) 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 d e c e 1 ( ). d e c e l ( ) takes a single parameter called v a 1 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 v e 1 vector isn't less than v a 1. If v e 1 is less than v a 1, the ball has slowed so much that if you were to further slow down the ball by subtracting the amount in v a 1 from v e 1, the ball would start to roll backward. To avoid having the ball roll backward, you can simply set the ve 1 vector to zero. However, if the magnitude is greater than or equal to v a 1 (meaning that the ball has not yet rolled to a stop), you can go ahead and do the deceleration calculation, like this:

```
public void decel (float val)
  if Oval >= vel.mag())
    vel.setvec(o, 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 V e c 2 D object called t v e c. It initializes by calling s e t V e c ( ) and passing v e 1 's d x and dy values, which gives you a copy of vel in tvec.

Next, you call the u n i t V e c ( ) method to convert t v e c into a unit vector. Then you call mu 1 Vec ( ) to shrink this vector down to the same magnitude as v al. Finally, you subtract this scaled vector from v e 1 by calling subVec().

Moving the ba«

Now that you've conquered deceleration, you can write the code to implement `Ba11'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 `dece1()` in order to update the ball's deceleration, like this:

Calling `dece1()` updates v_{e1} to account for deceleration, after which you can add v_{e1} 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 `Ba11` 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:

```
p~hlic void addPos (Vec2D  vel) (
    +=  ve l. 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:

```
lean    hitEdge:
'hitEdge = (x < bd.x + radius
            (x + radius) > (bd.x + bd.width)))
```

(continued)

```
(continued)
    x += (vel.dx = -vel.dx);
 if (hitEdge |= (y < bd.y + radius ()
                 |y + radius) > (bd.x + bd.height)))
    y += (vel.dy = -vel.dy);
 if (hitEdge)
    decel(vel.mag() * .Sf);
```

You use the boolean flag `hi tEdge` to signal that the ball has bounced off a vertical or horizontal edge. Then, you use the `dece1()` method to reduce the ball's speed by 80 percent after a rebound by multiplying `ve I's` current magnitude by .8.

Putting the bait

You also need to add code to `Bal 1` 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-

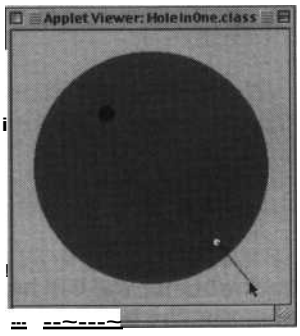


Figure 3-3: **The golf game interface uses a rubber-band style display to control the direction and force of a putt.**

Seleeting the ball

The `Ba11` class needs several methods to support this interface. First, you need a method called `touches()` to detect when the user clicks the ball:

```
public boolean touches (int mx, int my) t
  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.

freeuting 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():

```
public void putt (Point ptr) {
  vel.setvec((x - ptr.y) 120, (y - ptr.y) f 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:

```
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.

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 s u n k 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( ):

```
public void draw (Graphics g)
  if (!sunk)
    g.setColar(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 B a 1 1 class) to simulate the golf ball, the next step is to create code for the hole in the form of a H o 1 e class. H o 1 e is a little trickier to create than Ba 1 1 because the physics of how a real golf ball interacts with a real hole are far from simple. However, by using your new V e c 2 D 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:

```
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.

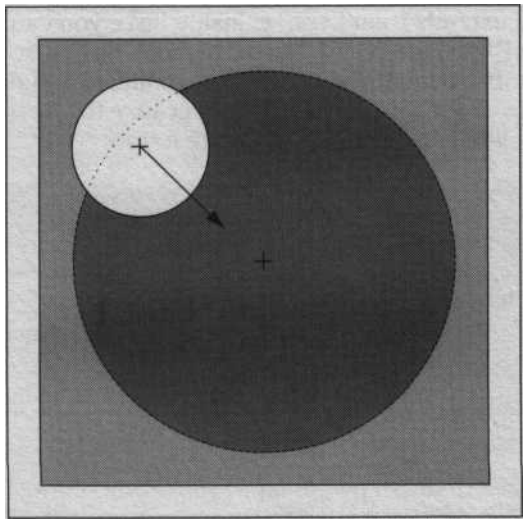


Figure 3-4: Gravity pushesthe ball down against the edge of the hole, which acts like a fictitious force pushing the ball toward the center.

Vectoring in

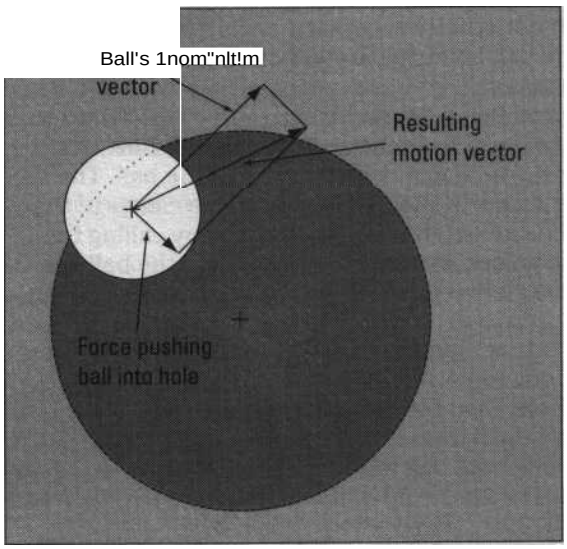
When you animate the ball by adding the v e 1 vector to the ball's position, you use v e 1 to simulate the force of the putter acting on the ball and the ball's resulting momentum pushing it in a particular direction. The fictitious force pushing the ball to the center of the hole can also be represented 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

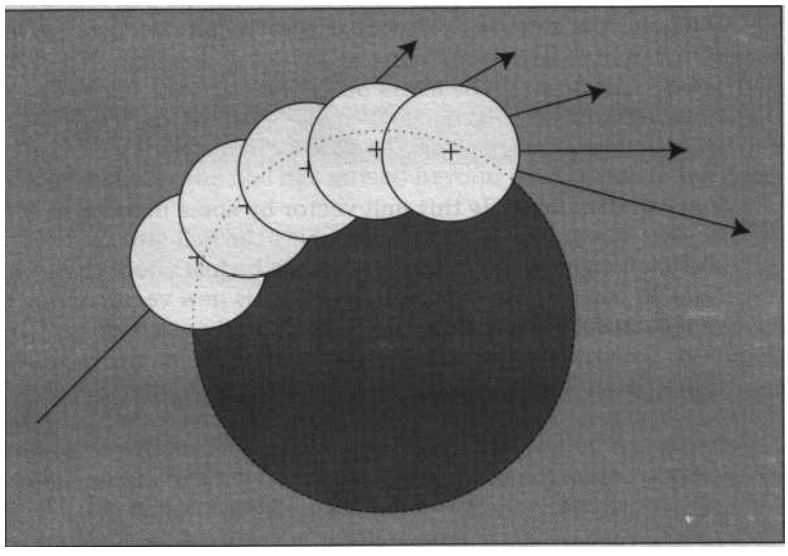
In the case of a fast moving ball that only grazes the edge of the hole, the fictitious force acting to push the ball into the hole is much weaker than the force of the ball's momentum. So the combined force only manages to deflect the ball's path. However, because the fictitious force deflects the ball 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.



Rpre 3-5: The result of adding am rectors is a new awtur that iconbines effects of the s wo.

e 3-6: **the sasxance ,and speed mw .zrrect, the ammammmmraoon aumirte force the sm the ftk and s an lie W to it{l w a Path Toe.**



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 H o 1 e called i n f l u e n c e ( ) 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 i n f l u e n c e ( ) to detect when the ball has been captured by the hole, and in turn set the s u n k flag to t r u e. 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 i n f l u e n c e ( ) code starts by computing two values to which you need to refer in several places in the code. The first value, d i s t I n , must be set to the r a d i u s of the hole minus the r a d i u s of the ball. The second value, h b D i s t , 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 C i r c I e method d i s t ( ). 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 h b D i s t is less than the r a d i u s of the hole and greater than d i s t I n , then the fictitious force should act on the ball. If h b D i s t 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 h b D i s t is less than d i s t I n , 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 V e c 2 D object t v e c to do the computation, like this:

```
tve~. et'Jec(x - ball.x, y - ball.y);
tvec. ;ni~Ve_();
```

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 a d d V e c ( ) to add this new vector to the ball's momentum vector v e 1 , like this:

```
ball.vel.addVecff" ~
```

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 (h b D i s t) is less than the radius of the hole minus the radius of the ball (d i s t I n). 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 v e 1 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 s u n k flag, like this:

```
ball. sunk |= ball.vel.naq() < radius && hbDist < 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 Hol e 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 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:

```
ball.sunk && hbDist > 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 v e 1 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.

Step one in faking it is to update the ball's velocity vector `vel` 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:

```
tvec.setVec(x - ball.x, y - ball.y);
tvec.mulVec((hbDist - distln) / 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:

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

Coding the Ho1 e 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 `Th read` 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 `Poi nt` object to record the position of the mouse and pass it to the `p u tt ( )` method in `B a 1 1`. And you need to override the applet methods `mouseDown ( )`, `mo useDrag( )`, and `mo useUp( )` to implement the full mouse interface. Here's the complete code for these three methods:

```
public boolean mouseDown (Event evt, int x, int y) i
  if (ball.sunk)
    ball = new Ball(x, y, BALLSIZE);
    repa i nt
  i
  if (lball.moving()  &&  select = ball.touches(x, y))) i
    putt = new Point(x, y);
    repaint();
  !
  return true;


public boolean mouseUp (Event evt, int x, int y) I
  i f (select)
    ball.putt(putt);
    repaint();

  select= false;
  return true


public boolean mouseDrag (Event evt, int x, int y)
  i f (select)
    putt = new Point(x, y);
    repaint():

  ret.irn  ra?:
```

Drawing the green

You can customize the code you write for the applet's p a i n t ( )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:

```
-,blic void paint (Graphics g) I
  i f (offscr == null) I
    offscreenImage = createImage(width, height);
    offscr = offscreenlmage.getGraphics();

  offscr.setColor(roughColor);
  Offscr.fillRect(0, 0, width, height):
  _ffscr.setColor(greenColor);
```

(continued)

(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);
        }
        g.drawImage(offscreenImage, 0, 0, this);
```

The complete code for Hole In One is included on the *Java Game Programming For Dummies* CD-ROM at the back of this book.

Chapter 4

JavaPool

6....0* OOOOO OOO 0 *OOOOOOOOO*OOOO6O #

h This Chapter

w The mathematics of detecting collisions

r Simulating pool

-> Modeling billiard ball physics

.....*OOOOOO OOOOOO*4OOOOOO . O *

he game of billiards certainly appeals to barflies and pool hustlers still ɣein' 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 ¡n 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 Came Programming For Dummies* CD-ROM.

calculating 13att-to-46att cottisions

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 presen•_ 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.

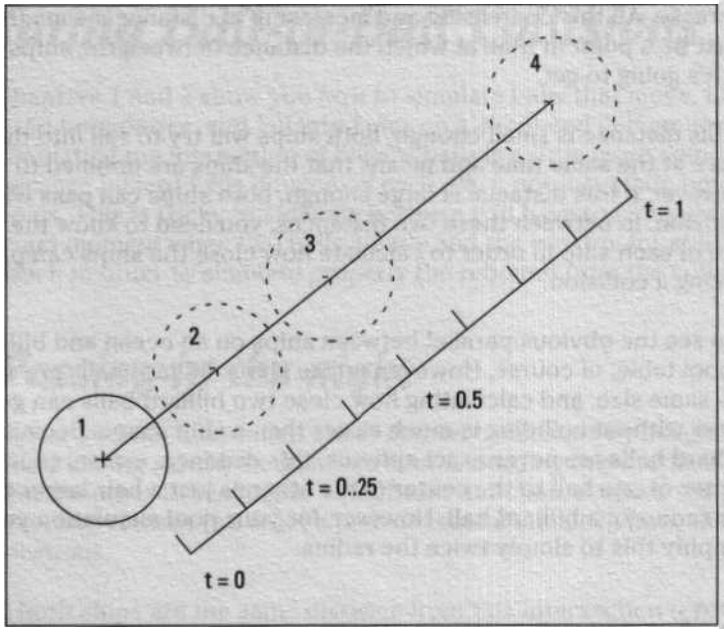


Figure 4-1: Because distance is proportional to time, you can calculate the position of a ball moving at a constant speed if you know two points and the time it takes to travel between them.

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 x dx, y + 3 x 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:

Calculating the distance to a co«ision

Chapter 3 covers using the relationship a2 + b2 = c2 (the Pythagorean Theorem) to calculate the distance between two points. You may recall that the distance between two points pointl and point2 is the square root of (point Lx - point2.x)2 + (point Ly - point2.y)2. 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, bl and b2, over time.

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))z+

sqrt((bl.y + bl.dy x t) - (b2.y + bl.dy x t))z

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 its 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 Lision.

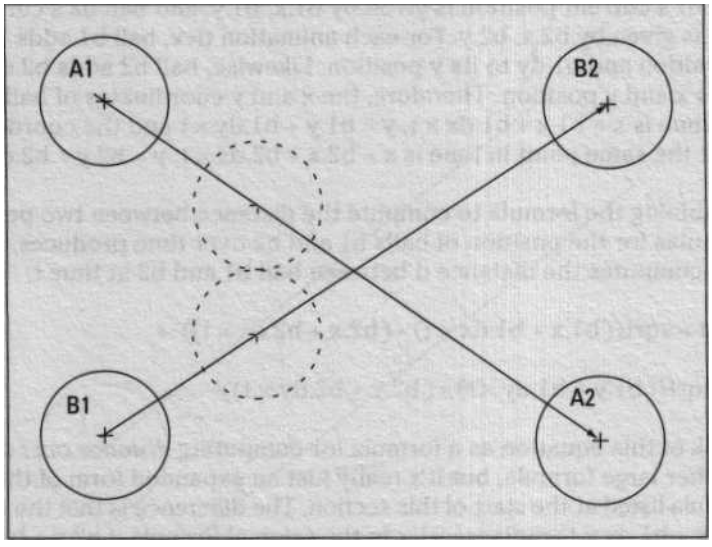


Figure 4-2: Some potential collisions would take place between the time intervals used to animate movement, which results in the collision being missed.

Solving for time

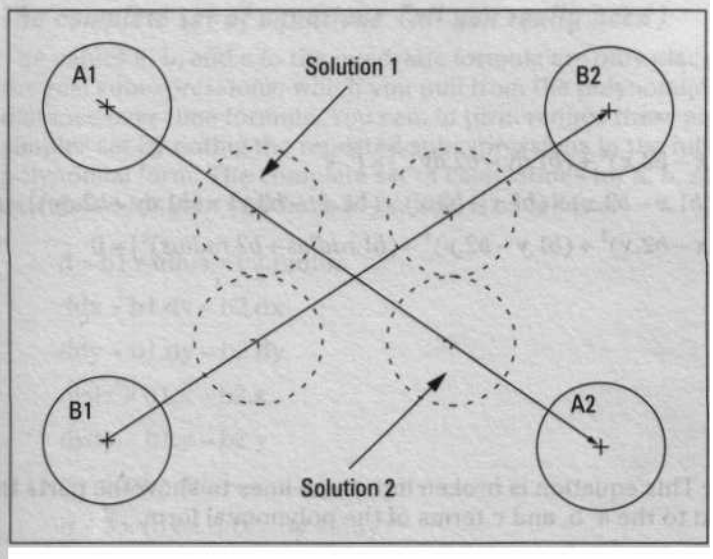
If you still remember any high-school algebra, you probably recall that formulas - like the one presented in the previous section for computing distance over time - can be rearranged to solve for specific values. From the position, direction, and velocity information for two objects at a specified time, 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 = radius x 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. ma themati ca . 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 = radius x 2, as demonstrated in Figure 43.



In Figure 43, ball A is moving from A1 to A2, and ball B is moving from 131 to B2. As the balls approach the intersection point, the distance between the balls becomes equal to radius x 2 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 2. 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 quation

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

General form of a polynomial equation

$$a \times t^2 + b \times t + c = 0$$

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.

$$((bl.x - 62.x)^2 + (bl.dy - b2.dy)^2) \times t2 +$$
$$2 \times ((bl.x - b2.x) \times (bl.x - b2.x) \times (bl.dx - b2.y) \times (bl.dy - b2.dy)) \times t +$$
$$((bl.x - b2.x)' + (b1.y - b2.y)' - (bl.radius + b2.radius)') = 0$$

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

The quadratic formula.

$$t = \frac{-b \pm \sqrt{b' - 4ac}}{2a}$$

The strange ± notation shows how you get two solutions to the equation. Solution 1 results when you *subtract* the subexpression

The subexpression in the quadratic formula that gives you the two possible solutions.

$$\sqrt{b^2 - 4ac}$$

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.

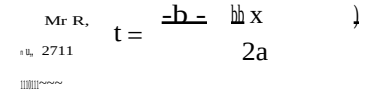
The complete set Of *equations (all you really need)*

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:

```
d = bl.radius + bl.radius
ddx = bl.dx - bl.dx
ddy = bl.dy - bl.dy
distx = b l.x - b2.x
disty = b1.y - b2.y
a=ddx'+ddy'
b=2x(dxxddx+dyxddy)
c=dx'+dy'-d'
```

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:

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

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 [01 to ball Ell and bal 1 [11 to ball [0]. Here's a more efficient way to arrange your loops:

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

By changing the first loop to start at 1 and by changing the second loop's comparison test to **j j < i** 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 u p d a t e B a 1 1 s ( ) method in the listing on the *Java Came 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 bull 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 bit- 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 leit edge. Likewise, it the of dy is positive, the ball can hit the bottom edge but not the top edge.

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.

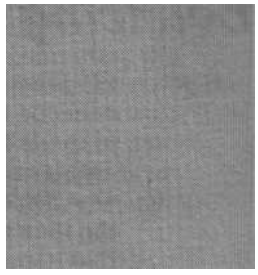
Ball-to-wall collisions are different from ball-to-ball collisions, but your pool simulation code will have to watch for both types of collisions at the same time. At any given moment, the code needs to know what type of collision will happen next and how to handle it - the next section shows you how.

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():

```
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:

```
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;
```

```
if (vel.dy >= 0)
  vCol = (bd.height + bd.y - y - radius) /vel.dy;
else
  vCol = (bd.y - y + radius) / vel.dy;
return Math.min(hCol, vCol);
```

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()

```
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() that 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 conservatimm 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?).

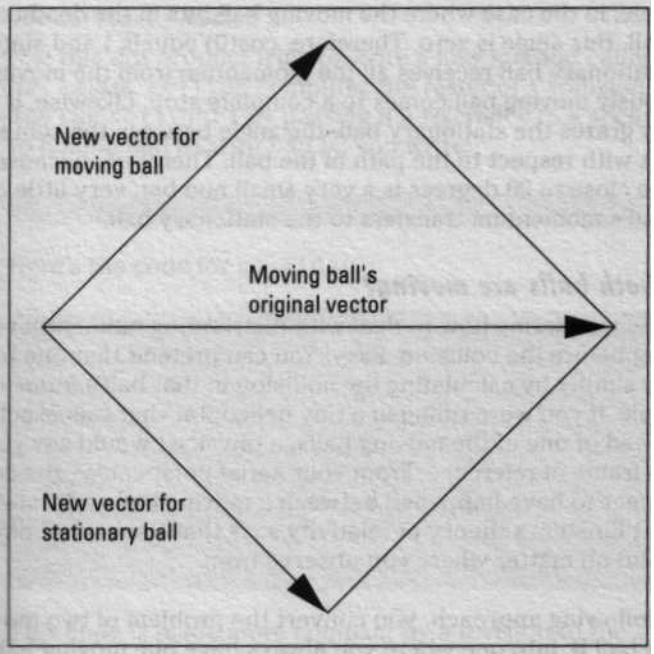
lmagine that a moving ball strikes a stationary ball and that at the point of collision, the stationary ball is exactly 45 degrees to the right of the path of the moving ball. After the collision, the previously stationary ball moves off at the 45 degree angle. Conversely, the path of the moving ball 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

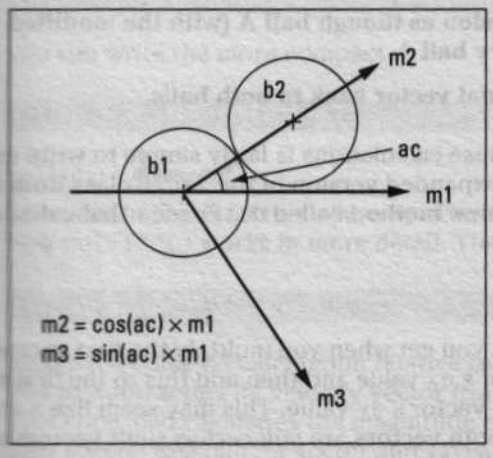
Chapter 3 demonstrates that the result of adding two vectors is a new vector that combines the effects of the original two. This same principle also applies in reverse. If you add the velocity vectors for the two balls after the collision, you must get a value that exactly equals the velocity vector 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.



Figm 4-4: After a ,=lesion baeen a ball and a aa ry ing vectors of the ma bob

Ie wector of the monly iuiioic~ ball



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) equals 1 and sin(0) equals t;„ I 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 new.*, 90 degrees with respect to the path of the ball. Therefore, because the cosime 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 i f *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 As 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 d of P r od ( ) 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.

The col l i de( ) *method*

Use this dot product trick to write anew method called col 1 i de ( ) for your B a 1 1 class. c o 1 1 i d e computes the result of a collision between two balls. You call c o 1 1 i d e ( ) by passing it a reference to a second ball. For example, to collide ball bl with ball b2 you write:

```
ba.collide(b2)
```

Here's the code for coI 1 i de( ):

```
public void collide (Ball b) I
   // calculate collision in b s reference frame
   float my = 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 V e c 2 D class (see "Creating a vector class" in Chapter 3) which changes any method that previously returned v o i d to instead return a reference to the same object. This trick means that you can combine several calls to successive V e c 2 D methods into a single statement. For example, instead of writing

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

you can write the more compact

```
float my - vel.subVec(b.vel).mag();
```

col 1 i de( ) *dissected*

After you understand this new way of writing vector code, you can examine how c o 1 1 i d e ( ) works in more detail. The first line of code

```
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 V e c 2 D objects: v 12 and v 1 c. The code that creates v 12 uses a new V e c 2 D constructor, takes references to two P o i n t.2'1 objects, and creates a vector that is the difference between the two Poi nit. objects. The code then converts this difference to a unit vector. The foIIoVF-ing line of code accomplishes these steps:

```
Vec2D vl2 = (new Vec2D(this, b1?.unitVec(?;
```

Next, the code creates the v 1 c V e c 2 D object by copying the current ball's vel vector and converting the copy to a unit vector, like this:

```
Vec2D  vlC = vgl.copy          . unitVec();
```

The code then calculates the cosine of the angle between v 12 and v 1 c by computing the dot product, like this:

```
float cos ° v1c.dotProd(vl2);
```

Next, the code sets the magnitude of v 12 to equal my * cos. Before this calculation, v 12 is a unit vector that points along a line from b to the current ball. Adjusting the magnitude to cos * my converts v12 into a vector that represents the amount of momentum to be transferred from the current baB to ball b.

In the next step, the code subtracts v 12 from the current ball's v e 1 vector. Doing so removes the momentum from the current ball - the same momentum that the code later transfers to ball 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 b. All these steps are accomplished in this line of code:

```
ve I. subVec( V12.mulyecfco,  • mv)).addVec(b.vel);
```

Finally, the code transfers the momentum taken from the current ball to ball b and restores its original frame of reference by adding v 12 to ball b's original vel vector, like this:

```
b.vel.addVec(vl2);
```

Putting A« the Pieces Together

Much of the code for the J a v a P o o 1 applet is copied directly from the H o 1 e I n 0 n e 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.

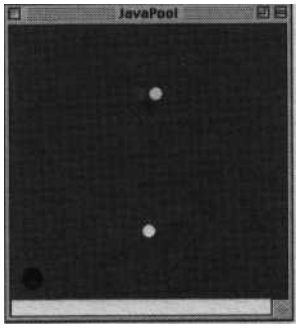
The controlling code for JavaPool applet is the code in the method updateBallS( ). The code in updateBallS( ) is based on the ideas discussed earlier in the "Timing and order" section. For each tick of the animation, updateBalls( ) calls each ball's edgeIntercept( ) and pathIntercept( ) methods to see 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.

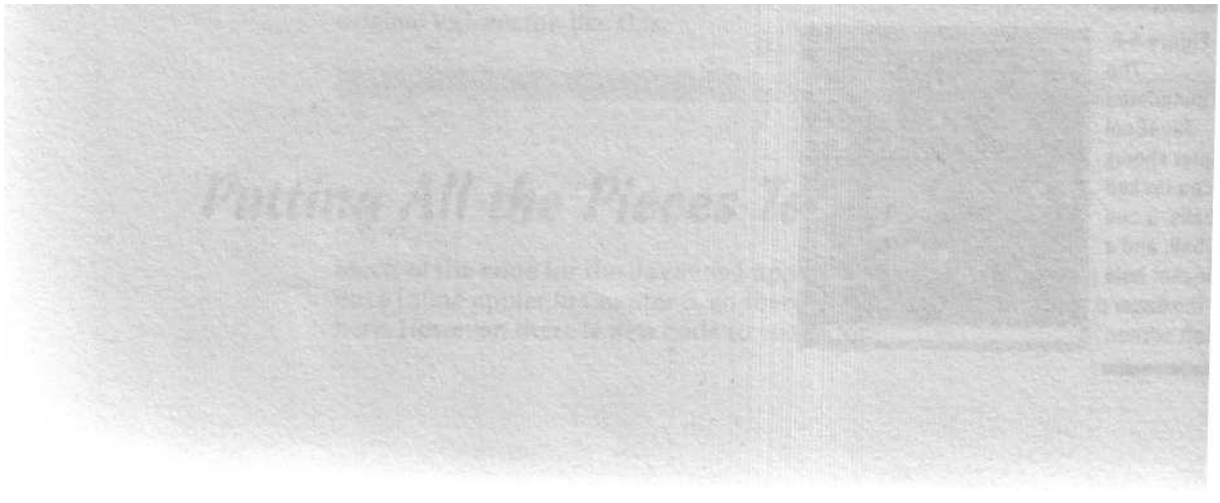
The interface for shooting a ball is identical to the click-and-drag interface described in Chapter 3, except that with the JavaPool interface you can select and shoot any of the four balls. The applet's init( ) method also creates a pocket hole in the lower-left corner of the applet. You can move this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



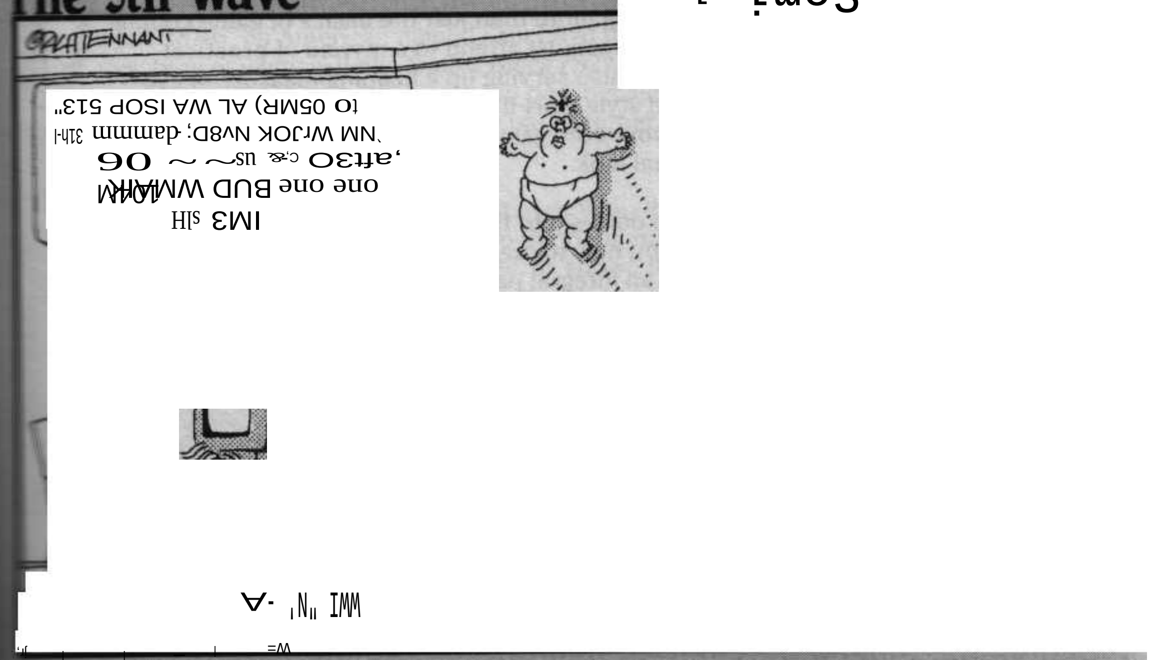
;~igmre 4-6: The completed JavaPool et shows >lhm racked balls, a cue hall, and a ket hole ijr ^nie lower- -- -orner.



Part II

Up to Speed

The 5th Wave



Ire this part . . .

Froducing 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 11 and Part III.

Chapter 5

Sliding Blocks Brain Teaser

In Tkis Chapter

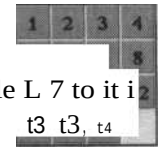
- Spicing up your games with images
- USing the Medi aTracker class
- fogramming puzzle logic
- Ilmmplementing a *click and drag* interface

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.

5-1:



immssible L 7 to it i
sie.

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.

Figure 5-2: The starting position for the Sliding Block Brain Teaser.

This chapter covers all the techniques used in the Sliding Blocks Brain Teaser applet. The complete code and ready-to-run applet is on the *Java Game Programming For Dummies* CD-ROM included with this book.

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 f i 11 3 D Re c t ( ) in the G r a p h i c s 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.

iytitat stamp Pads

Using I ma ge in Java is like having a digital stamp pad that you can use to stamp down copies of a picture onto a G r a p h i c s context. In this case, the stamp pad is an I m a g e object, and you stamp it using the d r a w I m a g e ( ) method provided in the G r a p h i c s class. However, before you can call d r a w I m a g e ( ), you first have to load an image file and create a Java I m a g e object. You can create an Image object by loading files from a Web server or from your hard disk.

To create an I m a g e object, you use a method called g e t I m a g e ( ) that is provided in the A p p 1 e t class. When you call g e t I m a g e ( ), you pass it a URL parameter (Universal Resource Locator, or more simply, Web address) that tells g e t I m a g e ( ) 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. g e t I m a g e ( ) loads the data from this image file and uses it to construct an I ma ge object.

You can also call get Image ( ) 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 g e t I m a g e ( ) can be conveniently used with two other App I et methods called getCodeBase( ) and get DocumentBase( ). Calling g e t C o d e B a s e ( ) returns a URL that points to the directory on the Web server from which the applet was loaded. Calling g e t D o c u m e n t B a s e ( ) returns a U R L 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()`:

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

public class Example extends Applet I
  Image  coffee;
  public void init() {
    coffee = gettmage(getCodeBase(), coffee.gif );
  I

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

Choosing GIF or JPEG

You can use `getImage()` tofetch 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 J PEG 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 wantto convert it to a GIF file, simply select Image-:>Modee 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 (n u l l) 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 n u 1 1 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 Media Tracker is to create anew Media Tracker object, like this:

```
MediaTracker tracker =  new MediaTracker(this);
```

The this parameter passed to the MediaTracker constructor is a referenot 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 = getlmage(getCodeBaseC), board.gif );
tracker.addlmage(boardImage,   board.gif ), 0);
```

Notice that you still need to call getImage() to request the Image you wauL 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 waitForAI1(), which doesn't return until all the images you added with addImage() are loaded. If you use waitForAl1(), it doesn't matter what ID value you pass to addImage().

MediaTracker.waitForAIIO

After you add alI the images to your MediaTracker object, you call the method waitForAI1() 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 waitForAl1() method has one more detail you have to handle.

The waitForAI1() 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 f
   tracker.waitForAll();
I  catch (InterruptedException e
```

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`:

```
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:

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

  piecelmages[ii] = getImage(getCodeBase(),
                             piece  + ii + gif );
  tracker.addImage(piecelmages[ii], 0);

try' I
  tracker.waitForAll();
catch (InterruptedException e) I
```

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:

```
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`.

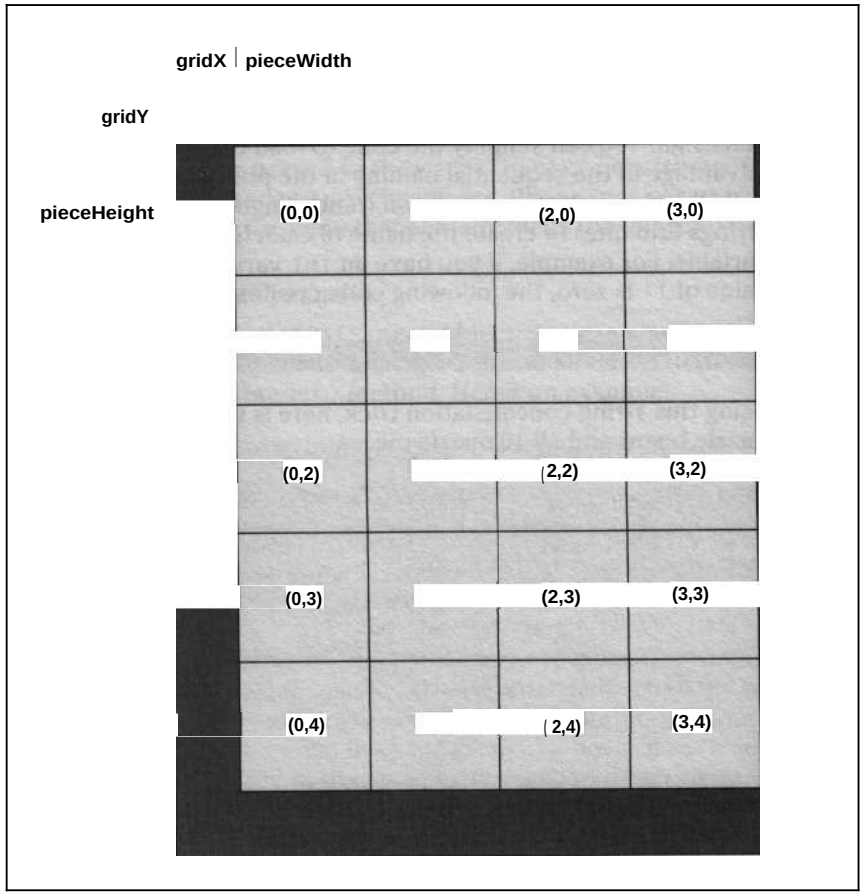


Figure 5-3: All the puzzle pieces on the game board align to an invisible 4x5 grid centered in the middle of the board.

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:

imageX = gridX x pieceWidth + gridX

imageY = gridY x pieceHeight + gridY

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.

Reading the width and height of *an* Image

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

```
int wi dt^ -  boardImage.getWidth(na113;
int height = boardlmage.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 gri dX, gri dY, pi eceWi dth, *and* pieceHeight

Using `getWi dth( )` and `get Height()` 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 p i `eceImages[` 51 to initialize `pi eceWi dth` and pi `eceHei ght,` like this:

```
nt pieceWidth - pieceImages[5],getWidth(null);
int pieceHeight = pieceTmages[5l.getHeight(null);
```

You still haven't initialized g `ridX` or g `ridY,` but you can easily calculate these values from the width and height of the game board image `boardImage` and pi `eceWi dth` and pi `eceHei ght.` 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 `pi eceWi dth,` so you calculate `gridX` like this:

```
int ridX =  (boardlmage.getWidth(null) -
            (pieceWidth * 4)) / 2;
```

You calculate `gridY` using nearly identical code:

```
:•idY = (boardlmage.getHeight(null) -
        (pieceHeight * 5)) / 2;
```

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 platy-. 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 tha 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 `Rectang1e` that is designed to represent a movable rectangular area. `Rectang1e` includes cod 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. `Rectang1e's` built-in features make it the perfect superclass for your new Pi `ece` class.

Starting with `Rectangl e, you` can easily code the beginnings of your new `Pi ece` class, including code to draw the image of the puzzle piece, like this

```
class Piece extends Rectangle {
  private  Image  pic;
  Piece (int bx, int by, Rectangle grid, Image img) i
    super(bx * grid.width + grid.x,
          by * grid.height + grid.y,
          img.getWidth(null), img.getHeight(null));
    pic = img;
```

```
;blic void draw (Graphics g)
 g.drawImage(pic, x, y, null);
```

The first two parameters to `Pi ece, 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 `gri dX, gri dY, pi eceWi dth,` and `pHyt. You` could pass these parameters to the constructor for `Piece` in four `int` parameters. However, it's easier to create a `Rectang1e` 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 p i c, which is used by `Piece's draw( )` method. The constructor calls i `mg . getWi dth( )` and i `mg . getHei ght( )` 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, `Pi ece'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 p i c 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:

```
public class Puzzle extends Applet I
  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 );
```

(continued)

(continued)

```
:rivate String      winMsg = Win!
 ivate  Point       selectedPiece, winLocation;
/ The pcs[] array specifies the starting position of the
/ different puzzle pieces on the 45 grid.
private int[][]     pcs = {11,01.10,01,(0,21,(3',01;{3,21,
                           (0,41,t1,31,t2,31,13,41,t1,2)1;public void init_:,!
 MediaTaacker tracker = new MediaTracker(this);
 boardImage = getImage(getCodeBase(), board.gif );
 tracker.addlmage(boardImage, 0);
 for (int ii = 0; ii < 10; ii++)

                            piece  + ii + GIF );
   tracker.addImage(piecelmages[ii], 0);

 try t

   tracker.waitForAll();
 catch (InterruptedException e) I ; }
 nt pieceWidth = pieceImages[5].getWidth(null);
 nt pieceHeight = piecelmages[51.getHeight(null);
 int gridX = (boardImage.getWidth(null) -
             (pieceWidth * 4)) / 2;
 nt gridY = (boardimage.getHeight(mull) -
             (pieceHeight * 5)) / 2;
 grid = new Rectangle(gridX, gridY,
                      pieceWidth, pieceHeight);
 winLocation = new Point(pieceWidth + gridX,
                         3 * pieceHeight + gridY);
 clickArea = new Rectangle(gridX, gridY,
                           4 * pieceWidth,
                           5 * pieceHeight);
 for (int ii = O; ii < 10; ii++)
   pieces[ii] = new Piece(pcs[ii][0], pcs[ii][1],
                          grid, pieceimages[ii]);
 offscreenImage = createlmage(size().width,
                              size().height);
 offscr = offscreenImage.getGraphics()
 repaint();
```

The puzzle pieces are instantiated inside the last f o r 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, p i e c e W i d t h, 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), `paint()` 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.

Moving 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:

```
public boolean mouseDown (Event evt, int x, int y) I
  if (clickArea.inside(x, y)) (
    for (int ii = 0; ii < pieces.length; ii++) {
      if (pieces[ii].inside(x, y)) I
        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 f o r loop to iterate through the list of pieces and, again, uses the i nside() method to check whether the user clicks the mouse inside this piece. If the mouse is clicked inside a piece, the reference for that P i e c e is copied to the p i c ked variable, and the location where the user clicked is recorded by creating the P oint object s e1ectedPiece and passing the x and y location of the click to the P oint constructor.

A Poi nt is an AWT class that holds the values of an x,y pair.

Moving the pieces

Next, you override the mo u seD r a g ( ) 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 p i c ked!=nu11 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:

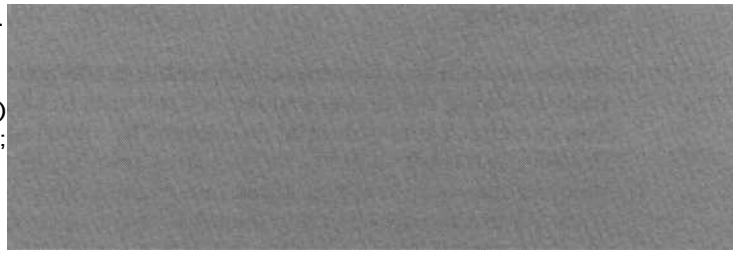
```
public boolean mouseDrag (Event evt, int x, int  y) f .
  if (picked != null && selectedPiece != null) I
    int  dx, dy;
    while ((dx = limit(y - selectedPiece.x)) != 0 &&
           picked.slide(pieces, dx, 0, clickArea))
      selectedPiece.translate(dx, 0);
    i
    while ((dy = limit(y - selectedPiece.y)) != 0 &&
           picked.slide(pieces, 0, dy, clickArea))
      selectedPiece.translate(0, dy);
    !
    repaint();

  return true;
```

Taking the horizontal movement first, the code calls a small helper method called 1 i m i t ( ) 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:

```
private int limit (int val) {
  return (val) > 0 ? 1 : (val < 0 ? -1 : 0));
}
```

If you're not familiar with it, 1 i mi t ( )'s use of the conditional operator(? :) in the return statement may look strange. However, this is just a more compact way of writing:

```
i f (val > 0) .
   return I;
else 1
   i f (val f 0)
      return -I;
   else
      return 0;
```



Inside m o u s e D r a g ( ), the code computes the difference between the original location where the user clicked the mouse to select the piece, which is held in the P o i n t value s e 1 e c t e d P i e c e, and the current position of the mouse, which is passed to mo u seDrag( ) in the x and y parameters. Then, mouseDrag( ) uses two w h i 1 e 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 wh i 1 e loop handles moving the piece left or right, and the second wh i 1 e loop controls movement up or down.

The wh i 1 e 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 s 1 i d e ( ).

S 1 i d e C) ing around

Writing s 1 i d e ( ) 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 Rectang1e.

Checking for pieces that block the slide path With `Rectangle.intersects()`

Given two rectangles, the Rectang1e method i n t e r s e c t s ( ) returns t r u e if one Rectang1e overlaps or *intersects* the other. You can use i n t e r s e c t s ( ) to determine whether the player is trying to move one puzzle piece on top of another by creating a new Recta n g 1 e 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 `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 `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 `Rectangle.union()` *and* `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 `Rectangle` class provides a simple way to check for the piece's position in the form of two additional methods: `union()` and `equals()`.

The `union()` method starts with two `Rectangles` and creates a new `Rectangle` that is as large as the smallest `Rectangle` you can draw that would contain both the original rectangles. The `equals()` method compares two `Rectangles` and returns `true` if both `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 bd is the `Rectangle` that describes the bounds of the board and `np` is a `Rectangle` object that describes the new position for the piece:

```
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:

```
int factorial (int n)
  if (n > 1)
    return n * factorial(n - 1);
  else
    return 1;
```

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()`:

```
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:

```
D    is boolean mouseUp (Event evt, int x, int y)
  -r (int ii = 0; ii < pieces.length; ii++)
    pieces[iil.snap(grid):
    cked =  null;
  paint();
r Aurn true;
```

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:

```
public void snap (Rectangle grid)
  move(((x - gridx + grid.width 1 2) / grid.width)
       grid.width + gridx,
      (y - grid.y + grid.height 1 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) I
    offscr.drawlmage(boardlmage, 0, 0, null);
    for (int ii = 0; ii C pieces.length; ii++)
        aces[iil.draw(cffscr);
    g.-:'r - lmage(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) I
  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[CI.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 `winLocation` check, is best added to the `paint()` method just after the `for` loop that draws the pieces. Here's the code:

```
if (pieces[0].x == winLocation.x &&
    pieces[0].y == winLocation.y) I
  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 *Java Game Programming For Dummies* CD-ROM included with this book.

Figure 5-4: The puzzle solved.

Chapter 6

Blackjack

In Tbis Chapter

- Building a complete game of Blackjack
- Programming the fundamentals of card games
- Eitracting graphics from composite images
- Creating a user interface using AWT components
- Tanging 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 omponent 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.

Understandiaq the Blackjack Game

®ackjack (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 =excellent choice for a computer card game: You can make the computer y 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 dealer's hand without going over 21. A hand's score is the sum of the ***Vahm*** 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 herselL 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:

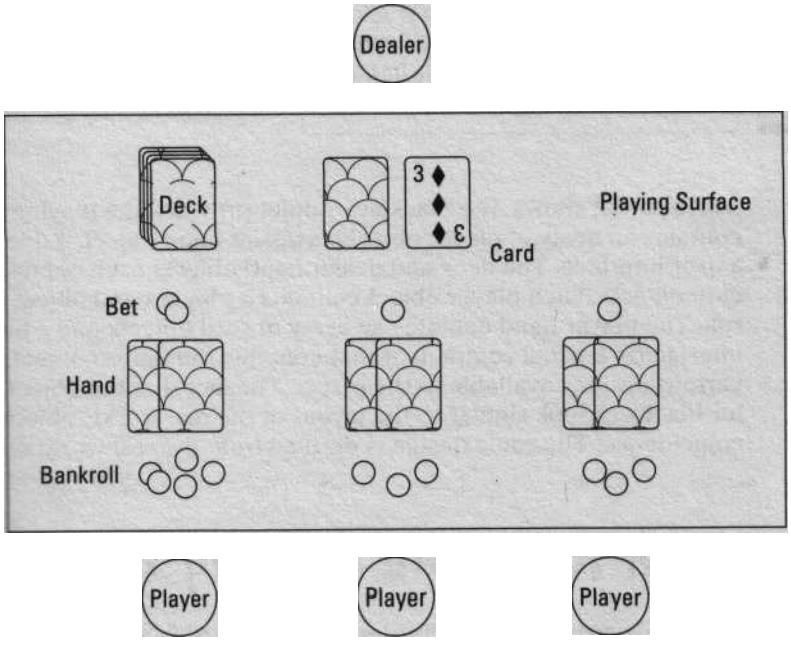
Blackjack's Special Situations

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.

Split — 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.

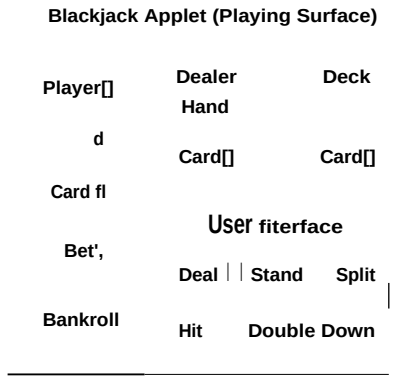


Figure 6-2: Blackjack object relationships.

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 bank-roll. 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 af 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 tc 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 Ca rd 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:

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

  public int getSize t)   (return top; I    /Ilf undealt cards
  public int getPacks () ( return packs;        packs in deck
  public int packSize () I return 52; }         cards in pack
  public Deck () C this(1)   I
  public Deck (int packs) i
    if ((this.packs = packs) > 0)
      top = packs * packSize();
      cards = new Card[topl;
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[rndl = temp;
```

(continued)

(continued)

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

1/ 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)1; l
  catch (lndexOutOfBoundsException e) {
    throw new NoSuchElementException();
  l

// 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.

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

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

    public class Card {
    public   static final int
         CLUB = 0, DIAMOND = l, HEART = 2, SPADE = 3:
```

(continued)

(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 I;
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; I
public final int getSuit () ( return suit; I
public Card (int rank, int suit) I
  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 () I
  return rank == ACE !! rank == EIGHT ? an    , a


public final boolean isRed () I
  return suit == DIAMOND !! suit == HEART;
¦

public' final boolean isFace ()
  return rank >= JACK;
¦
```

```
public static int getCardWidth ()  { return cardWidth;
public static int getCardHeight () i 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      A 2 3 4 5 6 7 8 9 10 J        OK   | CLUBS)
 *   CARDBACK A 2 3 4 5 6 7 8 9 10 J 0 K   | DIAMONDS)
 *   BLANK     A2345678910J0K   | HEARTS)
 *   BLANK     A2345678910J0K   | SPADES)

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



public static void drawCardBack (Graphics g,
                                 int x, int y) f
  // 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) I
  if (cardsImage != null) {
    Graphics gcopy = g.create();
    gcopy.clipRect(x, y, cardWidth, cardHeight);
    gcopy.drawimage(cardslmage, x - xoff, y - yoff, null);
    gcopy.dispose();



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

(continued)

(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;,
)

   end, class  Card
```

The Ca rd class supports Jokers even though Blackjack does not use a Joker. Ca rd 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 0 b j e c t. t o S t r i n g ( ) so that it implicitly converts an object to a string when using the + string concatenation operator. A Ca rd object has methods for returning the plural and singular names of its suit and rank. C a r d uses these methods in t o S t r i n g ( ) 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 equal s ( )

Notice that Card overrides both Object. equals() and Object. has hCode(). The Vector, Hashtabl e, and other data structure classes use equal s( ). A Vector is like an array that automatically grows as new objects are added to it. A Hashtabl e is a vector where the objects are located by using a key object, such as a string, instead of an index. Ha s ht a bl e uses a *hash code* (an integer) -calculated by calling the hashCode ( ) method for the key object-to find the object ~n the hash table. You need to maintain certain elationships between equals(and =zhCode( ) in order for an object to work orrectly with these data structures. While the -d objects work fine with the array used by -ie Deck class, the games that use the Ca rd :^ass may place Card objects in Vectors or s-tab1 es. So whenever you override -_~ 's() or hashCode( ),you must make Z-a that the following expressions remain _ Q

a,equals(b) == b.equals(a)

i f (a,equals(b))

a.hashCode() == b.hashCode()

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 equal s( ) or hashCode( ) changes, the ob-ject 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 r a n k attributes in both eq u a 1 s ( ) and ha s In Code ( ) but never changes their values after the C a rd ( ) constructor runs.

Extracting card graphics from a composite image

Ca rd 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 C a r d 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.

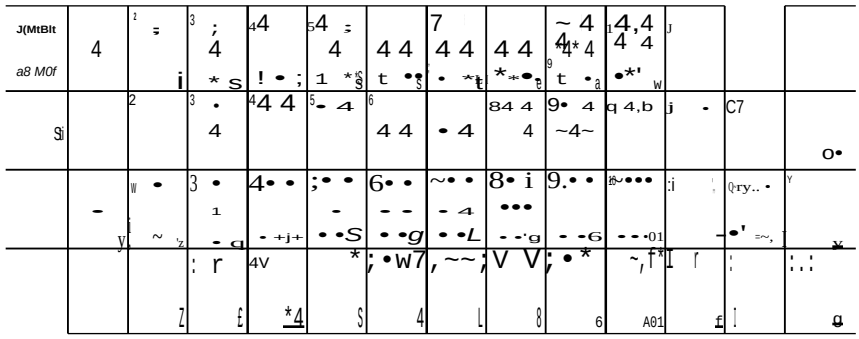


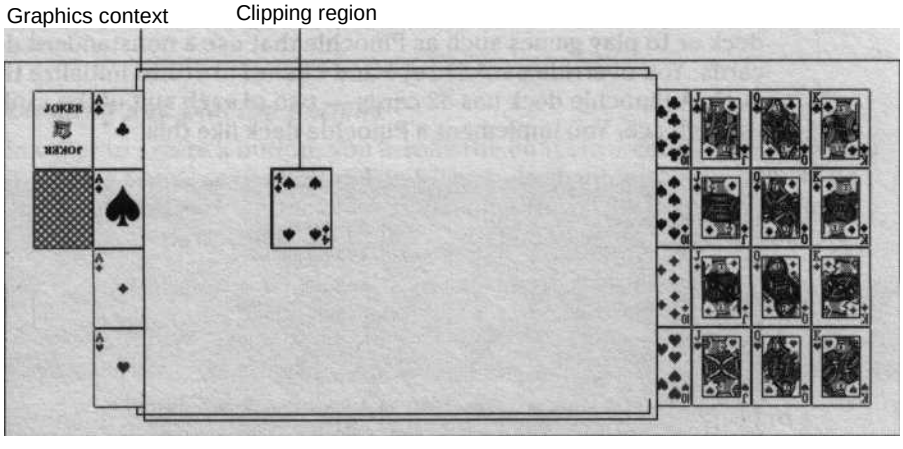
Figure 6-3: The "cards.gif" composite image.

Four rows of 14 card images - one row for each suit - combine to form the composite image. You call the static method i n i t G r a p h i c s ( ) to load the combined image. i n i t G r a p h i c s ( ) uses the Medi aTracker class discussed in Chapter 5 to load the cardsImage image. i n i t G r a p h i c s ( ) calculates the width (ca rdWi dth) of an individual card by dividing the composite image's width by 14 and the height (c a rd H e i g h t) by dividing the composite's height by 4.

To draw a single card rather than the entire c a rd s I ma g e 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 c a r d s I m a g e graphic, you make the clipping region the size of a single card, then position c a r d s I ma g e to line up the desired card with the clipping region and call d r a w I m a g e ( ) to draw the card. Figure 6-4 shows what this clipping image looks like if c a r d s I m a g e is a third piece of paper and you place it between the other two sheets of paper.

The draw( ) and drawCardBack( ) methods use card Wi dth and c a r d H e i g h t to calculate the x,y pixel coordinates of the upper-left corner of the image to extract from c a r d s I ma ge. These methods pass the x,y pixel coordinates to d o D r a w ( ), which uses them as the offsets for positioning c a rd s I ma g e so that the card to be drawn is in the correct place relative to the clipping region. d o D r a w ( ) sets the clipping region and draws the card using the following code:

```
Graphics gcopy = g.create();
  gcopy.clipRect(x, y, cardWidth, cardHeight);
  gcopy.drawImage(cardsImage, x - xoff, y - yoff, null);
  gcopy.dispose();
```



Using a clipping region to draw the four of spades.

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 t.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 p a c k S i z e ( ) and r e s h u ff 1 e ( ) 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:

```
public class PinochleDeck extends Deck I
  public PinochleDeck ()           ( super(); f
  public PinochleDeck (int packs) ( super(packs); I
  public int packSize (;)          ( return 48; }
  public synchronized void reshuffle () (
    top = 0:
    for tint packs = this.packs * 2; --packs >= 0; )
      for (int suit = Card.C'LUB; 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():
  !
l  II end class PinochleDeck
```

Creating a User Interface with *Components*

You create a user interface in Java from AWT components. Component is an abstract class in the j ava. awt package that embodies all the common functions of user interface elements. You build your interface from the subclasses of Component: Button, Canvas, Checkbox, Choi ce, Label, Li s - ScrolIbar,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

- W 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.

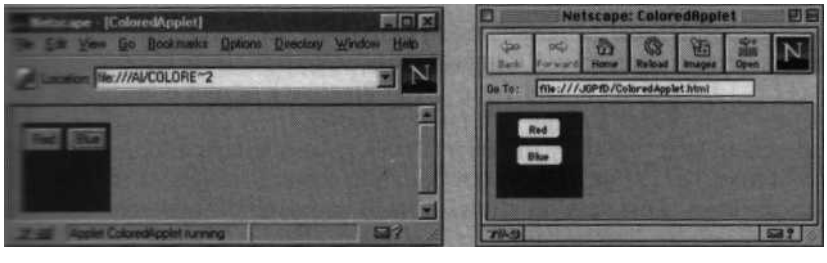
```
Button dealButton = new Button( Deal );
Button cancel  = new Button();
cancel.setLabel( Cancel   + operation);
```

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

You don't usually subclass `Button` or most other 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 component has a platform-specific *component peer*.

A component peer is a class that connects an AWT component such as a button to the OS's native implementation. The peers then draw the component and handle the component events. So for example, Microsoft Windows implements the `ButtonPeer` using a Windows button, and Apple Computer's MacOS implements the `ButtonPeer` 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 `ACT I ON-EVENT. You` override `Component. a c t i on()` in the parent container in order to handle action events. You identify the button that generated the event in `a c t i o n ( )` by comparing the `Event. t a r g e t` field to the `Button` objector, alternately, by comparing the `Event. a r g` field to the button label string.

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

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


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

  // end class ColoredApplet
```

Notice that the `a ct i on ( )` method in Col `oredApp I et` uses the `evt . to rget` field to find the `red` button and uses the `a rg` parameter to identify the `bl ue` button. We include both techniques in the example just to show both ways of checking. However, the `evt. t a r g e t` way is faster, because comparing object references is less work than comparing strings.

Reading and displa ying 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: `Text Fi el d`, `TextArea`, and `Label`. The Blackjack applet uses labels to display each player's name and current bankroll. `Text Fi el d` and `TextArea` also accept text typed by the user. `Text Fi el d` 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 L a b e 1 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:

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

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

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

      end cla    s ~, - • Label
```

This example initially sets the s c o r e D i s p 1 a y 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.

Ope 6-5. The 'SwreLabel applet.

Getting a few words from the user

Use a Text Fi el d component to collect a single line of text from the user. As with other components, you add a text field to a C o n t a i n e r component. You override a c t i o n() in the container to detect when the user enters text.

`Text Fi el d is` a subclass of `TextComponent. A Text Comp onent` supports th4 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 TextComponent Methods

Method	*Use This Method to ...*
`getText()`	Retrieve the current text from the component.
`setText(str)`	Set the value of the component's text to the string `st r.`
`getSelectedText()`	Retrieve the currently selected text in the component.
`select(start, end)`	Set the component's selection to the characters from offset `sta rt` to offset `end,` inclusive.
`setEditable(edit)`	If the boolean parameter `edi t is fal se,` user editing is disabled for the component; otherwise editing is enabled.

The offset of a character is the position in a string of that character, startinc 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 Fi el d, TextArea is` a subclass of `TextComponent` and has all the text editing features listed in Table 6-1. `Text Area` also has the three additional methods for working with the text, listed in Table 6-2.

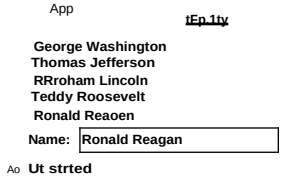
Table 6-2 Additional TextArea Methods

Method	*Use This Method to ...*
`appendText(str)`	Append the string `st r` to the end of the area's current text.
`i nsertText(str, pos)`	Insert the string `str` at the character offset pos.
`replaceText(str, start, end)`	Replace the text from character offset `sta rt` to offset `end,` inclusive, with the string `str.`

The following example uses a `Label, Text Field,` and `TextArea:`

```
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( Iist);
    add(new Label( Name: . Label.RIGNT));
    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;

  ll end class ListEntry
```

`L i s t E n t ry` creates a `Text Area` that holds five lines of approximately 20 characters each and a `Text Fi el d` that holds a single line of approximately 20 characters. The AWT calls `L i s t E n t ry . action()` when the user presses the Enter or Return key in the `e n t ry` text field. `a c t i o n ( )` appends the `e n t ry` text to the 1 i s `t` text area and selects the text in `e n t ry so` that the next text that the user types replaces the current text. Figure 6-6 shows the program running.



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 follo by a linefeed (\ r \ n).

Using C a n v a s to create new components

A C a n v a s 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 C a n v a s 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 I m a g e B u t t o n

The AWT components are functional but not particularly colorful. You can design your own custom components by subclassing the C a n v a s 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:

```
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[l

  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[Ol.getHeight(null),
                     img[1l.getHeight(null)):
    return new Dimension(x < 0 ? 10 : x; y < 0 ? 10 : y).;
```

```
  public Dimension preferredSize () I
    return minimumSize();


  public void paint (Graphics g) t
    if (img[imgndx] != null)
      g.drawImage(img[imgndx1, 0, 0, this);


  public void update (Graphics g) 1
    paint(g);
  I

  public synchronized void run ()
    imgndx = l;
    repaint();
    Component p = getParent{};
    if (p != null)
      p.postEvent{new Event(this,  Event.ACTION EVENT,
                            img[01));
try I Thread.sleep(200); } // press for 1/5 second
    catch (InterruptedException e) I}
    imgndx = 0;
    repaint();


  public synchronized boolean mouseDown (Event evt,
                                         int x, int y) I
    (new Thread(tg, this)).start();
    return true;
  I
  /f end class ImageButton
```

The following are the key points for implementing `ImageButton`:

- **ImageButton overrides Component. preferredSize() and Component. mini mumSi ze ( ) in order to have the button automatically size itself to the size of the largest image.**
- **mo u s e D own ( ) 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 synchroni zed 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 E v** `ent.arg` **field.**
- **The button animation is state-driven;** `run ( )` **sets the i** `mg ndx` **variaY to change the state, and** `pa i nt( )` **uses i** `mg 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 o f *cards*

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

Figure 6-7: A BlackjackHand with four cards.

R0 

The `B1ackjackHan` **d 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 `Bl ackjackHand` **class is**

```
i mport java.awt.*;
i mport java.util.*;

class BlackjackHand extends Canvas I
  protected int      horizInset;
  protected boolean soft;
  protected Vector hand = new Vector(6);
  private boolean    exposed, active;
  private Image      offscreenImage;
  private Graphics offscr:
  public Dimension preferredSize () I
    return new Dimension((Card.getCardWidth() + horizInset)
                         * 2, Card.getCardHeight());
```

```
BlackjackHand () I
  resize(preferredSize());


int cardCount ()              return hand.size(); I
protected boolean isDealer () I return true; I
boolean blackjack () ( // is a blackjack?
  return hand.size() == 2 && value() == 21;


boolean isSoft ()"( // has an 11-point ace in the hand?
  value();. // sets <soft> field
  return soft;


void setActive (boolean on) f // highlight this hand
  active = on;
  repaint():


void expose () I // expose the dealer s hole card
  exposed = true;
  repaint();
I

void clearHand () I // remove all cards from hand
  hand.removeAllElements();
  exposed = active = soft =,false;
  repaint();
  Thread.yieldf7:'


void deal (Card card) ( // deal a card to the hand
  hand.addElement(card);
  repaint();


public void paint (Graphics g) f
  if (offscreenImage == null) I
    offscreenImage = createlmage(size().width,
                                 size().height);
    offscr = offscreenlmage.getGraphics();
```

(continued)

(continued)

```
offscr.setColor(aetive ? 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) l 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;

l
g.drawlmage(offscreenImage, 0,     this);


.,ublic void update (Graphics g) I paint(g); }
/ returns the point value for the hand
nt value () I
  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;,

! returns base point value of card <cardNum> in the hand
[ ^otected int value (int cardNum)
  int rank = ((Card)hand.elementAt(cardNum)).getRank();
  return Math.min(Card.TEN, rank);   // face cards are 10's

I  // end class BlackjackHand
```

Here are the key elements in B 1 a c k j a c k H and:

- `BlackjackHand` extends `java.awt. Canvas so` that it can display itself on the screen.
- The `B1ackjackHand()` constructor sets the size of the canvas - necessary because by default a canvas is zero pixels wide and zero pixels tall.
- The `java.uti1.Vector` field `hand` stores the cards. (Remember, a `Vector` works like an array that automatically reslzes itself as more objects are added to it) Using a vector allows a hand to accept any number of cards dealt to it. `dea1()` adds cards to the vector and `clearHand()` removes all the cards from the vector.
- `B1ackjackHand` 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.
- `Wpaint()` uses the graphics context `offscr` to draw to the offscreen image. `offscreen Image. 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. `setActi ve()` 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 j `ava . awt . LayoutManager` ¡nterface to determine what size to make AWT components and where to place them. Each container -a `Panel , Appl et, Wi nd ow, Frame, Di al og. or Fi 1 eDi a 1 o g` -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 `mo v e ( )` method to position the component, calls `r e s i z e ( )` tf set the component's size, and calls `r e s h a p e ( )` to set both the size and position. Most layout managers use the component's p `r e f e r r e d S i z e ( )` ar `mi ni mumSi ze( )` methods to determine its dimensions, but some layout managers ignore these suggested sizes (see Table 6-3).

You can subclass `L a b e 1, B u t t o n,` 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 - `B o r d e r Layout, C a r d Layout, F1 ow Layout, Gri d Layout, Gri dBagLayout.` Table 6-3 lists types of layout managers assigned by default to each type of container.

Table 6-3 Default Layout Managers

Container	Layout Manager	Comments
`Panel`	`FlowLayout`	
`Applet`	`FlowLayout`	
`Window`	`BorderLayout`	(ignores suggested size)
`Frame`	`BorderLayout`	(ignores suggested size)
`Di al og`	`BorderLayout`	(ignores suggested size)
`Fi IeDialog`	`null`	

A container can have a `n u 1 l` 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 b background graphic - may be best with a `n u 1 1` layout manager.

If you want to use a different layout manager than the default, pass the nei layout manager to the container's `set Layout ( )` method. For example, if y want to have an applet use a `B o r d e r L a y o u t,` place the following in the applet's `i n i t ( )` method:

```
setLayout(new BorderLayout());
```

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:

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

Figure 6-8: applet using -ut.

`BorderLayout` arranges 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:

```
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 )3;
    add( Center , new java.awt.Button( Center ));
  }
```

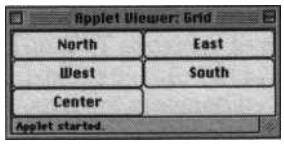


Figure 6-9: An applet using BorderLayout

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

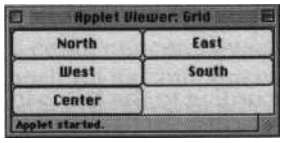


Figure 6-10: An applet using GridLayout.

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.

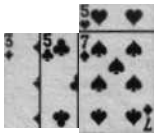


Figure 6-11: Laying out split hands.

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

```
import java.awt.

class PlayerHandLayout implements layoutManager i
  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) t)
  public Dimension preferredLayoutSize (Container parent) S
    Dimension ds = ((BlackjackHand)parent.getComponent(0)). preferreaSize l;
    int numHands = parent.countComponents();
    if (numHands <-- 2)
      return new Dimension(ds.width, ds.height * numHands);
```

(continued)

(continued)

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


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


  public void layoutContainer (Container parent) I
    Dimension hsize = ((BlackjackHand)parent.getComponent(O)).preferredSize();
    int numHands = parent.countComponents();
    int h, y = D;
    for (int ii = 0; ii < numHands    i++)
      if (ii == player.getHandlndex()   !!
          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 ;
  )
] // end class PlayerHandLayout
```

The prevalent aspects of P1 `ayerHandLayout` are

- The `B 1 a c k j a c k P 1 a y e r` **passes a reference to itself and the vertical** `overl ap` **ratio to the P1** `ayerHandLayout ( )` **constructor.** `1 ayoutConta i ner ()` **uses pl** `ayer` **to access information about the hand.**
- `1 a y o u t C o n t a i n e r ( )` **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.**
- `V 1 ayoutContai ner( )` **calls** `reshape( )` **to set the size of overlapped hands to the size of the exposed area of the hand. Using** `r e s h a p e ( )` **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.**

- `preferredLayoutSize( )` calculates the dimensions needed to display `numHands` hands by assuming two exposed hands: the active hand and the last hand. If the active hand is the last hand, only one hand is exposed, but the calculation still sets the size of the hands as if two hands were exposed. The result of this calculation allows the container to size itself based on the maximum number of hands a player can have.
- Layouts such as `BorderLayout`, which separate the components into different named groups, use the `addLayoutComponent( )` and `removeLayoutComponent( )` methods to organize the components into groups. However, `PlayerHandLayout` doesn't use these methods and just needs to include empty methods to complete the `LayoutManager` interface.

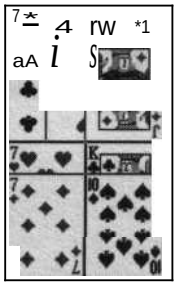


Figure 6-12: Exposing active hand.

Dividinq the screen with panels

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 `GridLayout(0, 1)` layout manager. Add the radio button `Checkbox` 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.

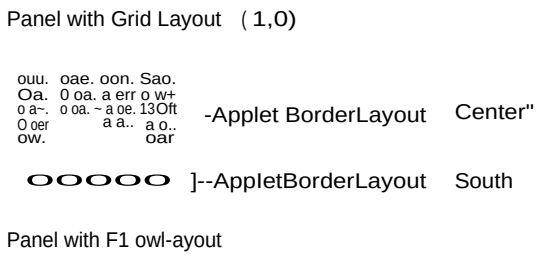


Figure 6-13: Nested panels.

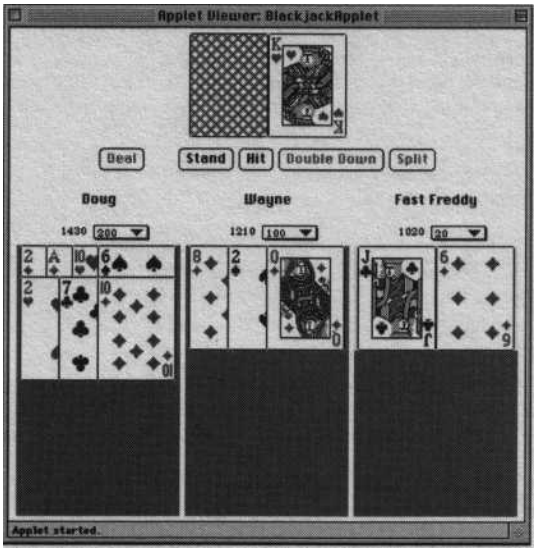
You can also arrange this example with a `GridBagLayout`. A `GridBagLayout` is a powerful layout manager for creating complex arrangements. Unfortunately, using a `GridBagLayout` is a fairly complicated process, and nested panels, as in this example, are usually easier to create and maintain.

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.

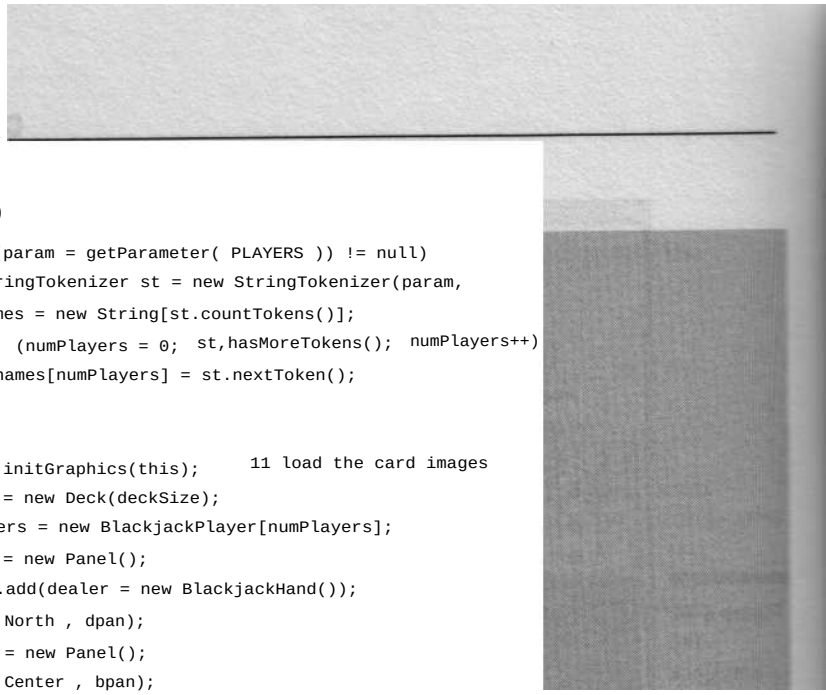
The top-level applet

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 = l, bankroll = 0;
    String     param;
    String[]   names = null;
    Thread curT = Thread.currentThread();
    appTG = curT.getThreadGroup();
    curT.setPriority(curT.getPriority() - 1);
    setLayout(new BorderLayout());
```

(continued)



(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);     11 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 = getPlayert);
    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) I
        hand.deal(dea1());
        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());
        splltHand.deal(deal()):
        setButtons(player);
      ]
      else
        return super.action(evt, what):
```

```
public synchronized void run () {
  newDeal():
  newdeal.disable();
  try I
    BlackjackPlayerHand hand;
    for (int card = 2; --card 7= 0: )
      for (int pp = 0; pp < players.length; pp++)
        if ((hand = players[ppl.getHand()) != null) l
          hand.deal(deal());
          Thread.sleep(500);
        ]
      dealer.deal(deal());
      Thread.sleep(500);
    ;
    if ((hand = nextHand(true)) != null) I
      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);
```

(continued)

(continued)

```
    ¦
    catch (InterruptedException e) 11
    for (int pp = 0; pp < players.length;   pp++)
      players[ppl.resolveDeal(dealer);
    newdeal.enable();
  ¦

  BlackjackPlayer getPlayer () I
    return curPlayer )- players.length ?
             null : players[curPlayerl;


  BlackjackPlayer nextPlayer () I
    curP]ayer++;
    return getPlayer();


  BlackjackPlayerHand nextHand (boolean firstHand) {
    BlackjackPlayer      player = null;
    BlackjackPlayerHand hand = null:
    ¡f (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();
    ¡f (player == null) I
      setButtons(null);
      synchronized (this) I notify(); I // deal to dealer
      return null;
    ¦
    ¡f (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();
  |
    players[iil.clearHands();
    players[iij.newHand(0);

  newdeal.enable();
  setButtons(null);
  return shuffled;


private void setButtons (BlackjackPlayer player) I
  |f (lastActive != null) I
    |astActive.setActive(false);
    |astActive = null;

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




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

  |
}  Jl end class BlackjackApplet
```

The HTML that loads the applet

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

```
<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. `dealCard()` 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 1 to display the player's Blackjack hand(s)**

B1 a c k j a c k P1 **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 1 a c** k j a c k P 1 a y e r:

```
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
              I l,  5, 10, 20, 30, 50, 100, 200, 500, 10001;
  private static int       playerCount;
  private String           name;
  private int              l astBet, bankroll, curHand,
                           numHands;
private final BlackjackPlayerHand[]
                         hands = new BlackjackPlayerHand[4];
  private Label            bankrollLabel;
  private Choice           betEntry;
  private Panel            handsPanel;

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

  int getHandlndex () i
    return curl-and >= numHands ? 1 : curl-and;
  I

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

  int getBankroll () ( return bankroll; }
  int getActiveHandCount () I return numHands; l
```

(continued)

(continued)

```
boolean canSplit () I return numHands < hands length; I
BlackjackPlayer (Container parent) I
  this(parent, null, STANDARD_BANKROLL);
I

BlackjackPlayer (Container parent, String name,
                 i nt bankroll) I
  parent.add(this);
  setLayout(new BorderLayout()):
  playerCount++;
  i f (name == null)
    name = Player   + playerCount;
  add( North ,  new  Label (this.name = name,  Label. CENTER));
  i f (bankroll <= 0)
    bankroll = STANDARD_BANKROLL;
  String bankStr = Integer.toString(bankroll);''
  Font font = new Font( Courier , Font.PLAIN, 10);
  bankrollLabel = new Label(bankStr, Label.RIGNT);
  betEntry = new Choice();
  for (int ii = 0; ii < betAmount.1ength; ii++)
    betEntry.addItem(Integer.toString(betAmount[iil));
  bankrollLabel.setFont(font);
  betEntry.setFont(font):
  Panel pbank = new Panel();
  add( Center , pbank);
  pbank.add(bankrol]Label);
  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)
  i f (evt.target == betEntry)
    setBet(Integer.parselnt(betEntry.getSelectedltem()));
```

```
  else
    return super.action(evt, what);
  return true;


synchronized void clearHands () {
  while (numHands > 0)
    hands[--numHandsl.clearHand();
  curHand = 0;
  handsPanel.layout();


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


synchronized BlackjackPlayerHand newHand (int bet) I
  BlackjackPlayerHand result = null;
  if (bet == 0)
    bet = lastBet;
  if (numHands < hands.length && (bet = setBet(bet)) > 0)f
    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) I
  this.bankroll += amount;
  bankrollLabel.setText(Integer.teString(this.bankroll));

I  /I end class BlackjackPlayer
```

The responsibilities of a `BlackjackPlayer` are

- A `BlackjackPlayer` **can have up to four hands active at once.** The `hands[]` **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:**

```
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(0) == value(1);
```

```
void clearHand () f
   super.clearHand();
   hasSplit = false;
1

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


int winnings (BlackjackHand dealer) i
  int hval = value();
  if (hval <= 21) I
    if (blackjack()) I
      if (dealer.blackjack())
        return bet;
      else
        return (bet << 1) + (bet » 1);

    else i
      int dval = dealer.value();
      if (rival ? 21 () rival < hval)
        return bet <( 1;
      else if (dual == hval && !dealer.blackjack())
        return bet;


   return
```

Chapter 7

2-D Maze

o • • o ! o • t • o o o o o • o • o ! • o o o o o 6 o o o o • o o • o o Y ` 4 o o 8

W *This Chapter*

Creating block and wall mazes

Generating random mazes

p. Solving mazes using the right-hand rule

Solving mazes using breadth-first searching

em. Displaying 2-D mazes

0*00:*0*sa*si*sesoss*a0s00so*00a0000s*os®s09*0**

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:

V 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.

r 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 `Wall Maze` 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` sub classes. 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:

```
in, pcrt  java.awt.

abstract class Maze extends Canvas {
  static final byte TOP     = 0x01;
```

```
static final  byte  RIGHT   = 0x02;
static final  byte  BOTTOM = OxO4;
static finial  byte  LEFT    = OxO8;
static final  byte  DIRTY   = (byte) OxBO;

protected byte[][]  maze;

protected: abstract byte ¦nitSq  ();
```

Maze also contains a c 1 earMaze() 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 i nit S q ( ) to return the initial value for the squares with this line:

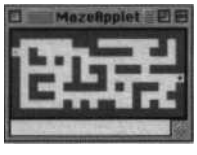
```
protected abstract byte initSq ();
```

Declaring i nit S q ( ) as an abstract method requires that any classes that extend the Maze class must implement i nit S q ( ).

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

The B 1 o c kM a z e subclass

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



Fqprre 1-t: block – aze.

To extend Maze to implement a block maze, you implement the abstract method i nit Sq ( ) that Maze defines. The portion of the B 1 ockMaze class responsible for representing the maze looks like this:

```
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 `| WALL | DIRTY)` "ORs" the `DIRTY` flag with the `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.

The `Wall Maze` 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.

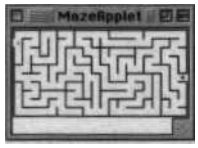


Figure 7-2: A wall maze.

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 `mazeE1E1` grid square. You set the bit when the wall exists and clear the bit when the wall is open. The `LEFT`, `RIGHT`, `TOP`, and `BOTTOM` static final variables in `Maze` define the bit values used in `maze E ] E 1`. The portion of the `Wa 1 1 Ma ze` class responsible for representing the maze looks like

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

  protected byte initSq () l
    return (byte) (BLOCKED | DIRTY);

      end class WallMaze
```

You display a square with all walls set as a solid wall. The static final `BLOCKED` variable holds the value with all the walls set - a solid wall is `BLOCKED` on all sides. i ni `tSq( )` creates its return value from `BLOCKED`.

Note that because squares share walls with adjacent squares, `mazeE1E1` 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

checking for walls easier than if ma ze C] C 7 were to record the walls in only one square. Given that a game typically changes the maze much less frequently than it checks the state of the maze, making checking the maze easier than changing it is a good trade-off.

Working with bits

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 *base* 10because 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 ast not yet- and instead use a series of on/ off values as a counting system. Every value ¦ored in a computer is composed of bits. Each is either an off value or an on value. By c--nvention, an off bit represents the value 0 ar d an on bit represents 1. The computer represents values larger than 1 by combining bits t= form *base* 2 or binarynumbers.

lava doesn't have a representation for using msnary numbers in your code, but it does have _-e for *hexadecimal (base* 16)numbers. In the -=xadecimal counting system, each digit is icr- of the values from 0 to 9 or from A to F rr-ere A = 10, B = 11, and so on. A hexadecina digit represents exactly four binary digits sc converting between hexadecimal and bi-rary numbers is relatively easy. You indicate a hexadecimal number in Java by beginning the member with a zero followed by an x, like this:

```
,x10      OxAlB2C3      OxDEADBEEF      0x17
```

example, the decimal number 23 is 10111 a 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.

```
20+3 = 23 10  = 16+4+2+1 = 10111      = 16+7 = 17 16
```

You use the "bitwise" operators &, |, A,-,<<, >>, and >>> to work with individual bits and the "or" (|) operator to set or combine bits:

```
x |= 1  // sets the ones bit in x
```

You use the "and" (&) operator to clear or test bits:

```
(x & 1) != 0 // true if the ones bit in
    x is set
x &= 1 // clears all bits in x except the
    ones  bit.
```

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

```
x ^= 1 // toggle the ones bit in x (0-
   >1, 1->0)
```

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

```
x &= -1 // clears the ones bit in x
```

You use the "left shift" operator («) to move all the bits in the value to the left:

```
x «= 1 // equivalent to x *= 2 (10111 <<
   1 = 101110)
```

You use the "right shift" operator (> >) to move all the bits in the value to the right:

```
x >>= 1 // equivalent to x i=
   1 = 1011)
```

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, pva 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?

- oO 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 `Wall Maze` and `Block Maze` subclasses. Here are the fields and methods you add to the `Maze` superclass to keep track of which squares am "dirty" and need to be redrawn by `paint 0ff screen 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[xl[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 `paint 0ff screen 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 `Wall Maze` 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 `lastSide` to LEFT.
3. Randomly choose a sequence to rotate through the remaining sides (the sides other than `lastSide`) 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 `lastSide` 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:

```
public boolean isOpen (int x, int y)
  return inBounds(x, y) && sqr(x, y) != BLOCKED:
```

(continued)

(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 sqr"
    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:
          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) I // mark sqr as maze exit
            found = branch = true;
            finishX = xx; finishY = yy;

          break;
```

(ail

```
    }
    if (!found) f
      if (nsq == BLOCKED)      // unused square, use it
        found = true;          // Step #5 - add the square
      else if (--sent > 0)     J/ 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
      l/  sq  contains the current square which was either
      1/ inited 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][yyl = (byte)
                    (found ? (sq & -nextSide) : sq);
      dirtySquare(xx, yy);
      if (displayConstruction)
        showMaze(false);
      if (branch) f
        // Step #7 - select a square to branch from
        if (count < threshold) I
          // exhaustively search for remaining squares
          sq = BLOCKED;
SEARCH:   for (xx = 0; xx < mzWid; xx++
            for (yy = 0: yy < mzHyt; yy++)
              if (sgr(nx, yy) == BLOCKED)
                int dir = rint(4);
                for (int ii = 4; --ii >= 0; ) f
                  nx = xx; ny = yy;
                  switch (dir = ++dir & 3)'I
                    case 0: nx--; break;
                    case l: nx++; break;
                    case 2: ny--; break;
                    case 3: ny++; break;
                  ]
                  if (inBounds(nx, ny) &&
                      (sq = sqr(nx, ny)) != BLOCKED) {
                    xx = nx; yy = ny;
                    break SEARCH; // found sqr for branch

      if (sq _= BLOCKED)
        break;   // maze done
```

(continued)

(continued)

```
}
else I // randomly search for a sqr for new branch
  do i
    xx = rint(mzWid);
    yy = rint(mzHyt):
    sq = sqr(xx, yy);
  } while (sq == BLOCKED);

// Step ##8 - set lastSide to an open side
for (lastSide = l: (lastSide & sq) != 0;.)
  lastSide <<= l;
scnt = 3;
sideInc = rint(3):
branch = found = false;


} while (!found);
// Step ##10 - init the new square and set lastSide
if ((lastSide = nextSide << 2) > BLOCKED)
  lastSide >>= 4;
sq = nsq & -lastSide; // Step ##9a
xx = nx; yy = ny;
if (!displayConstruction   &&  (count & OxFF) == 0)'
  Thread.yield():   // give some time to other threads
I/ Step ##11 - check for more squares (at top of loop)


if (finishX ( 0) f // no exit square selected, do it now
  maze[xx - mzWid - 1l[yy = rint(mzHyt)[ &= (byte)-RIGHT:
  dirtySquare(xx, yy);
  if (displayConstruction)
    showMaze(false):

if (!displayConstruction)
  repaint();
} // generate()
```

Notice that each time the code modifies them `a ze[ ]` C] array, it calls `d i r ty Sq u are()` 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 `s h owMa z e ( )` works.)

Step 7 in g e n e r a t e ( ) employs two different strategies for picking a random square from which to start a new branch. When lots of BLOCKED squares remain, g e n e r a t e ( ) 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 t h r e s h o 1 d 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 Sq r class shown here:

```
class Sqr
  private boolean t, b, 1, 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.] = 1; this.r ° r;


  // open() returns a count of the unexplored directions
  int open () i
    return (t ? 1:0) + (b ? 1:0) + (r ? 1:0) + (1 ? 1:0);
  ]

  int select (int n, boolean sameDir)
    // Step #6b - select dir to explore and mark as explored
    if (sameDir) // try to expand in dir square was entered
      switch (dir) i
        case Maze.TOP:     if (t) { t = false; return dir; }
                           break;
```

(continued)

(continued)

```
        case Maze.BOTTOM: if (b) I b = false; return dir; 1
                          break;
        case Maze.LEFT;   if (1) 1 1 = false; return dir; 1
                          break;
        case Maze.RIGHT: if (r) I r = false: return dir; }
                          break;

    // return the n th unexplored direction
    if      (t && --n < 0) ( t = false; return Maze.TOP; I
    else if (b&& --n < 0) 1 b = false; return Maze.BOTTOM;l
    else if (r && --n < O) (r  = false; return Maze.RIGHT;
    else                   { 1 = false; return Maze.LEFT:  !
  |

| /l end class Sqr
```

The open ( ) method in the Sq r class returns the number of unexplored directions from the square. s e l e c t ( ) returns an unexplored direction and then marks the direction as explored.

Here are the steps the B 1 o c k M a z e 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 BlockMaze 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 an overloaded version of 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:

```
private Vector pending; // list of available Sqr objects
private int strt = 70; // prcb 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; /I 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 al lowProb)
  reti~rn _ , (allowProb)  &&  isOpen(x, y);


private boolean blocked (int x, int y) i
  return inBounds(x, y) && sgr(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) {
  ll Step #7 - check if adjacent square in direction dir
  //            should be added to the maze
  switch (dir) f
    case TOP:
      Y  ;
      if (isOpen(x, y-1, thru)
           isOpen(x-1, y, side)  |' isopen(x+l, y, side)
           noDiag(x, y, -1, -1) ~~ noDiag(x, y, 1,  -1))
        return false;
      break;
```

(continued)

(continued)

```
    case BOTTOM:
      y++;
      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;
    case LEFT:

      if (isOpen(x-1, y, thru) ||(
          isOpen(x, y-1, side) ~~ isOpen(x, y+l, side)
          noDiag(x, y, -1, -1) ~~ noDiag(x, y, -1, 1))
        return false;
      break;
    case RIGHT:
      x++:
      if (isOpen(x+l, y, thru) ||
          isOpen(x, y-1, side) || isOpen(x, y+1, side) ~~
          noDiag(x, y, 1, -1) || n,oDiag(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 <= D || y >= mZHyt-1)
    return false;   /I square on border or out of bounds
  else {
    Sqr sq = new Sqr(x, y, dir,
                     blocked(x, y-1), blocked(x, y+l),
                     blocked(x-l, 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 lit - select and set the starting square
  pending = new Vector();
  maze[startX = 01[startY = rint(mzHyt - 2) + 11 = FLOOR;
  dirtySquare(startX, startY);
  pending.add[lement(new Sgr(startX, startY, RIGHT,
                             false, false, false, true));
  // Step #3 - loop until list of squares is empty
  while (!pending.isEmpty()) i
    // Step #4 - select a square to explore
    if (prob(sdit))
      idx = pending.size() - l;    // continue with last Sqr
    else
      idx = rint(pending,size()); // choose random Sqr
    Sqr next = (Sqr) pending.elementAt(idx);
    ll Step #5 - remove square if no more sides to explore
    // Also randomly remove squares to reduce maze density.
    if ((free = next.open()) <= 1   ii
        (pending.size() > 10 && prob(dens)))
      pending.remove[lementAt(idx);
    if (free > 0)
      // Step 4i6a  - select a  direction to explore
      if (tryDir(next.x, next.y,
                 next.seleet(rint(free), prob(sdir))))
        if (displayConstruction)
          showMaze(false);

       Step #8 - explore another square

  1'  (ldisplayConstruction),
    repaint();
1   // generate()
```

The key to generating an interesting block maze is how Step 7 in the method try Dir ( ) decides whether to add a square to the maze. try Dir ( ) looks at the squares surrounding the new square candidate. Figure 7-3 shows the operations try Dir ( ) performs when the dir parameter is RIGHT.

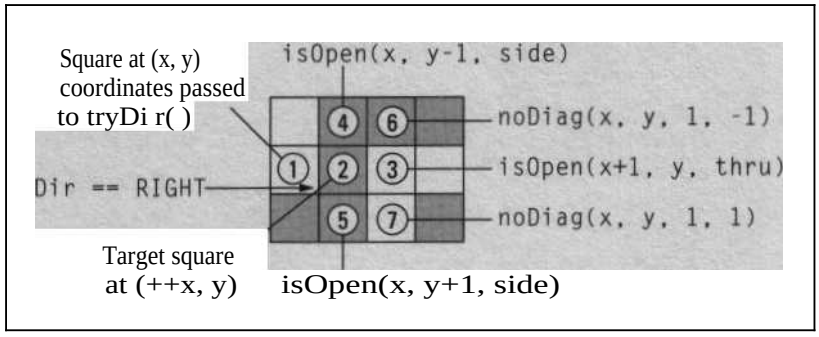


Figure 7-3: Checking a square in tryDi r (x. Y, RIGHT).

Here is the code from the t ry D i r ( ) s w i t c h statement:

```
case RIGHT:
  x++;
  if {isOpen(x+1, y. thru) ''
      i sOpen(x, y-1, side) '' 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 i s0pen (x+1. y , t h r u) to check whether the square to the right (square 3) is a floor square. Here is the i s0pen ( ) method:

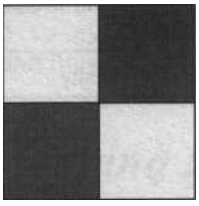
```
private boolean isOpen (int x, int y, int allowProb) I
  return prob(allowProb)  &&  i sOpen(x, y);
I
```

The thru parameter specifies the probability (from 0 to 100) that i s0pen( ) returns t r u e regardless of whether or not the square is open. Because t h r u is 90 and square 3 in Figure 7-3 is open, there is a 90 percent chance that p rob (90) returns t r u e and a 90 percent chance that the first call to i sOpen( ) returns true. If i sOpen() returns true, tryDi r( ) returns fal se and doesn't use the square.

Even if t ry D i r ( ) 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 i `s0pen()` returns `fa1se`, the next check i `s0pen(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, i s `Open()` returns `fa1se` 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 i `s0pen()` again returns `fa1se`.

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

```
private boolean noDiag (int x, int y, int dx, int dy) I
  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 `B1ockMaze` sets the `diag` probability to 100, t ry D i r ( ) 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 `B1ockMaze` generates. You change the settings by changing the initialization values of the private fields in the `Bl ockMaze` class.

Table 7-1	B1 ockMaze Maze Generation Settings	
Field	*Default*	*Increasing This Value Generates a Maze That...*
s t rt	*70*	has longer uninterrupted passageways
s d i r	60	has straighter passageways with fewer turns
thru	*90*	has fewer loops and alternate paths
side	60	has fewer open areas and fewer areas wider than the passageway
diag	100	allows fewer diagonally opposed squares like the squares shown in Figure 7-4
dens	15	has more walls and a lower path density

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:

```
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 L E FT, R I GHT, 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 p a *t h [I [I* is (LEFT | TOP).

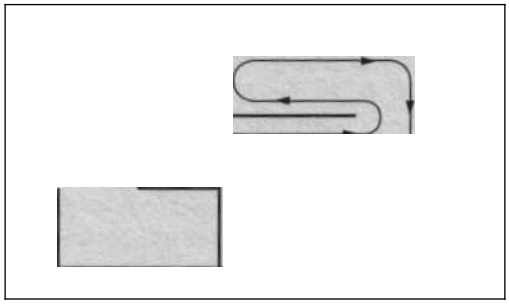
You add the following methods to the M a z e class to define the methods for traversing the maze:

```
abstract boolean traverse (int startX, int startY,
                           int finishX, int finishY,
                           boolean displaySearch);

public boolean traverse (boolean displaySearch) I
  return traverse(startX, startY, finishX, finishY,
                  displaySearch);
```

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 !eft hand on the wa!!

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.



dpe 7-5: Reverse maze using the left-hand rule.

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 W a 1 1 M a z e class implements the left-hand rule traversal algorithm like this:

```
public synchronized boolean traverse
  (int xx, int yy, int fx, int fy, boolean displaySearch)

  if (!inBounds(xx, yy)   ||   !nBounds(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[xxl[yy] = LEFT;
    dirtySquare(xx, yy);
    side = TOP;
  }
  while (xx != fx   ||   yy != fy) I
    while ((sq & side) != 0) // search for direction to try
      if ((side <f= 1) >  BLOCKED)
        side = TOP;
    path[xx][yy] ^= side; // set exit from current square
    dirtySquare(xx, yy);
    switch (side) f // set entrance to new square
      case LEFT:    path[--xx][yy]  ^= RIGHT;    side = BOTTOM;
                    break;
      case TOP:     path[xxl[--yy] ^= BOTTOM;   side =  LEFT;
                    break;
      case RIGHT:   path[++xx][yy]  ^= LEFT;     side = TOP;
                    break;
      case BOTTOM:  path[xxI[++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
      II the starting square, so there s no solution
      resetPath():
      if (displayPath)
        repaint();
      return false;

    if (displaySearch)
      showMaze(false);
    else if ((++count & OxFF) == 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 t r a v e r s e ( ) 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 t r a v e r s e ( ) marks a path in a square, the code sets the bit for the path to one. As t r a v e r s e ( ) 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 p a t h [ ] [ ] 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:

```
byte[][] graph = new byte[mzWid][mzHyt];
```

First, you initialize each entry in the g r a p h [ ] [ ] array to zero. Next, set the entry in g r a p h [ ] [ ] for the square where the search starts to -1. As you search, you record in g r a p h [ ] [ ] 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 g r a p h [ ] [ ] array.

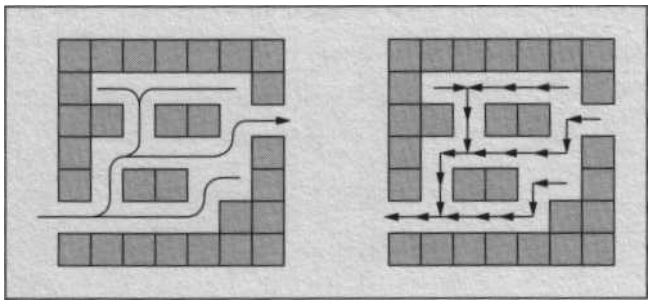


Figure 7-6: A breadth-first search path and corresponding graph array.

While searching, you maintain a queue of squares waiting to be searches. After you add a square to the search path and set the corresponding erao , in `graph[][]`, you add the square to the queue. Each iteration of the sea 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 UN 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 oth® The term queue means "a waiting line," and like a line at the Department 4 Motor Vehicles, the first person in line is the first person served.

Here is the breadth-first `traverse()` method for the `Blockmaze` class:

```
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[gsize][21; // 0 = x, 1 = y
  byte[][]  graph = new byte[mzWid][mzHyt];
  boolean    solve = (xx == startX && yy == startY &&
                      fx == finishX && fy == finishY);
  if (displaySearch) l
    resetPath();
    if (solve) 1
      path[sx][sy] = LEFT;
      dirtySquare(sx, sy);

    showMaze(false);
```

```
  graph[xxl[yyl
  queue[0]E0] = (short)xx;   queue[0]E1] = (short)ny;
  qtail = 0: ghead'= 1:
TRAVERSE:
  for (;;) {
    if (qhead == gtail) I. // empty queue: unsolvable maze
      resetPath();
      if (displayPath)
        repaint();
      return false;-
    f
    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++;   rdir = 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[nxl[nyl = (byte)ndir; // point to prev square
        if (displaySearch) I
          path[xx][yy] |= dir;
          dirtySquare(xx, yy);
          path[nx][nyl |= ndir;
          dirtySquare(nx, ny);

        if (nx == fx && ny == fy) // found solution
          break TRAVERSE;
        queue[ghead][0] = (short)nx;
        queue[ghead][1] = (short)ny;
        qhead = (qhead + 1)% qsize;
      l

    if (displaySearch) I
      if (qhead == qstart) { // dead end, backtrack
        while (path[xx][yy] == graph[xxl[yyl) I
          path[xxl[yyl = 0;
          dirtySquare(xx, yy);
```

(continued)

(continued)

```
            switch (graph[xx][yy]) I
              case TOP:      path[xx][--yy] &_ (byte)-BOTTOM;
                             break:
              case RIGHT:    path[++xxl[yy] &_ (byte)-LEFT;
                             break;
              case BOTTOM:   path[xx][++yy] &_ (byte)-TOP;
                             break;
              case LEFT:     path[--xxl[yy] &_ (byte)-RIGHT;
                             break;
            l
            dirtySquare(xx, yy);

        l
        showMaze(false);

      else if ((++count & OxFF) _= 0)
        Thread.yield();

    if (displaySearch) I
      if (solve) {
        path[xx][yyl ↓= RIGHT;
        dirtySquare(xx, yy);

      showMaze(false):'


    /l reconstruct path by following graph
    // from finish to start
    resetPath();
    if (solve)
      path[fx][fy] = RIGHT;
    while ((dir = graph[fxl[fy]) != -1)
      path[fx][fy] ↓= (byte)dir:
      switch (dir) I
        case TOP:      path[fx][-- fy] = BOTTOM;   break;
        case RIGHT:    path[++fx][fy] = LEFT;      break;
        case BOTTOM:   path[fx][++fy] = TOP;       break;
      } case LEFT:     path[--fxl[fy] = RIGHT;     break;

    }
    if (solve)
      path[fx][fy] y= LEFT;
    if (displayPath)
      repaint();
    return true;
```

Notice in the preceding code that after you find a solution, you reconstruct the path by following g r a p h [ ] [ ] from the destination square to the starting square.

Displaying a 2-D Maze

The M a z e class extends j a v a . a wt . C a n v a s to give it a display area. You draw the maze by overriding the p a i n t ( ) method inherited from C a n v a s.

Because most of the work to draw a wall maze or a block maze is the same, the M a z e class overrides p a i n t ( ) to do the drawing. As far as the p a i n t ( ) method is concerned, the only difference between the two Maze subclasses, Bl ockMaze and Wal l Maze, is in the actual drawing of a square. paint( ) calls d raw Squa re ( ) to draw a single maze square. You have to implement the abstract method d r a w S q u a r e ( ) that Maze defines and that p a i nt( ) calls in the classes that extend M a z e. B 1 o c k M a z e draws its squares as solid blocks. Wa 1 1 Maze draws walls between squares and leaves the middle of the squares open, except that W a 1 1 M a z e draws squares with walls on all four sides (BLOCKED squares) as solid blocks.

Implementing a circular queue

breadth-first traverse( ) code imple- ! thequeue[] array as a circular queue. ¢ai mular queue allows you to continually put fires into one end of the queue and remove Wires from the other end without ever run- ing into theendofthequeue[] array. If you ⇑on't use a circular queue, you have to peri- sAcally move the squares from the end of the r back to the beginning. You add squares in -r_ nead of the queue like this:

```
4ei-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 gsizesets qhead to zero if(qhead + 1) equals q s i ze. (Remember, the modulo operator returns the remainder of dividing the left-hand operand by the right-hand operand - the remainder of dividing (ghead + l) by qs i ze in this case.) You use similar code to remove the next square from the tail of the queue:

```
xx = queue[gtail][0];
yy = queue[gtail][1];
qtail = (qtail + 1) % qsize;
```

Using the p a i n t ( ) 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 `u p d a t e ( )` for smooth screen updates.) You add the following `p a i n t (:` method and supporting fields to the `Maze` class:

```
protected Image      offscreenImage;
protected Graphics   offscr;

public synchronized void paint (Graphics g) {
  paintOffscreenImage();
  g.drawImage(offscreenImage. 0, 0, this);
  notifyAll();
```

`paint()` calls `pai ntOffscreenlmage( )` (another method in the `Maze` class) to do the work of creating the maze image in an offscreen image. You do the work in `p a i n t0 f f s c r ee n I ma g e ( )` rather than in `pai nt( ) so` that the subclasses can override `pa i nt ( )` without replacing the code in `pa i ntOffscreen Image( )`. Keeping the offscreen image code out of `pa i nt ( )` allows the subclasses to modify how `pa i nt ( )` works and still reuse the code in `p a i n t 0 f f s c r e e n I m a g e ( )` to generate the maze display- (Chapter 8 shows an example of overriding `p a i n t ( )` 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 `pa i nt ( )` rather than in the `Ma ze ( )` constructor because `C r e a t e Image ( )` 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 `pa i nt ( ).` The Abstract Window Toolkit (AWT) can't call `pa i nt ( )` until you add the component to the screen hierarchy, so `C r e a t e I m a g e ()` is guaranteed to work when `paint( )` calls it.

Next, `p a i n t O f f s c r e e n I m a g e ( )` loops through all the squares in the range of dirty squares and draws any squares that have the `D I RTY` bit set in the square. For each `DIRTY` square, `pa i ntOffscreenImage( )` calls `drawSquare( )` to draw the maze square and then calls `draw Path Square` to draw any path through the square. (The "Displaying a solution" section later in this chapter shows how to implement `d r a w P a t h S q u a r e ( ).`) If the square is the maze's start or finish square, `p a i n t O f f s c r e e n I m a g e ( )` calls `d r a w T a r g e t ( )` to draw colored circles to mark it.

Finally, you copy the offscreen image to the screen in paint( ) with this statement:

```
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 notifyAI 1 ( ) 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:

```
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 I wait(); I catch (InterruptedException e) {}
  long t = System.currentTimeMillis();
  if ((timer -= t - (1000L / maxFrameRate)) > 0)
    try ( Thread.sleep(timer); I
    catch (InterruptedException e) II
  timer = System.currentTimeMillist);
```

Calculating where the pikels 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 calculatin the pixel offsets and sizes in the `Maze` class. The `drawSquare( )` methods ¡mplemented by the classes that extend `Ma ze` use these pixel locations to draw the maze squares.

You calculate the pixel values in `resetMaze( ). re setMaze( )` adjusts the pixel width and height of each rectangle in the grid to fit the screen size aih the `Maze` canvas. `Maze` overrides the `reshape( )` method inherited from `C a n v a s` to resize the maze whenever the size of the canvas changes. If `r e s h a p e ( )` changes the size of the canvas, it calls `r e s e t M a z e ( )` to calcuilm the new size of the grid squares. Figure 7-7 and Table 7-2 show the fields thW `resetMaze( )` calculates.

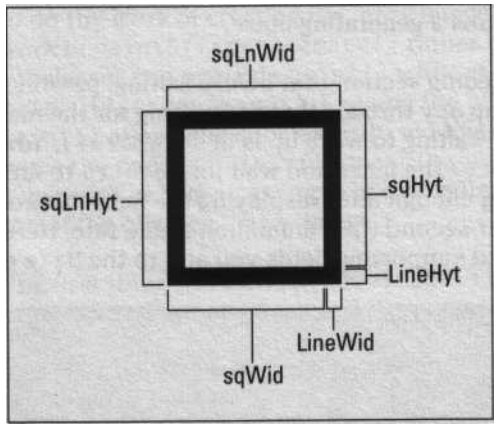


Figure 7-7: The pixel dimensions of a maze square.

Table 7-2 Pixels Values for Drawing the Maze

Field	What It Contains
`sgWid,sqHyt`	The width and height of a grid square in pixels. These ¡nclude the 1 `i neWi d` and `! i neHyt,` respectively, of the line on one side of the square.
`l i neWid,lineHyt`	The pixel width and height of the lines separating grid squares. You set these to zero for block mazes.
`sgLnWid,sqLnHyt`	The pixel width and height of a grid square plus the separating lines.
`l eftOffset, topOffset`	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 `ma ze [ 01 [ 01.`

Knowing that block mazes are simple is hat(the battle

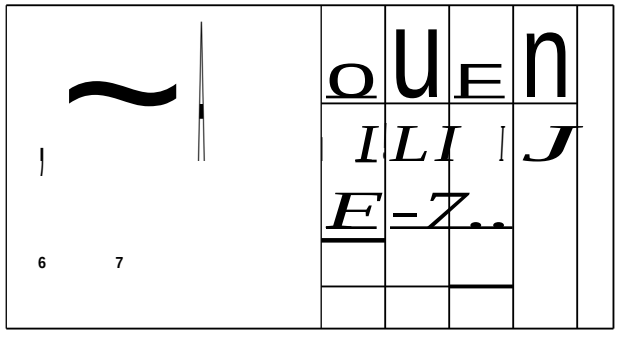
The squares in a block maze are simple colored rectangles. The type of maze square determines the color of the rectangle. Figure 7-1 earlier in this chapter shows an example of a block maze. You implement the `drawSquare( )` method for the B1 `ockMaze` class like this:

```
protected void drawSquare (int xx, int yy) I
  offscr.setColor(maze[xxl[yy1 == WALL ?
                  Color.gray : Color.white):
  offscr.fillRect(leftOffset + (xx * sgWid),
                  topOffset + (yy * sgHyt), sgWid, sgHyt);
```

The expression 1 `eft0ffset + (xx * sqWid)` calculates the pixel offset of the left edge of the maze square and `top Off set + (yy * sq Hy t)` calculates the top pixel offset of the square.

Displaying a watt *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.



iqm7-8: 7he 8 wall actions and ıssulting 16 wa l l squares

Notice that in Figure 7-8 all 16 squares draw all 4 corner sections of the wall, which happens because `Wa 1 1 Maze` only generates dense mazes with no open areas. If you allow wall mazes with open areas, you only draw a corner 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 `Wa11Maze` class, including the `drawSquare()` method, is included on the CD.

To reduce the number of `fi11Rect()` calls that `drawSquare()` has to use in order to draw the maze square, the `fi11Rect()` 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:

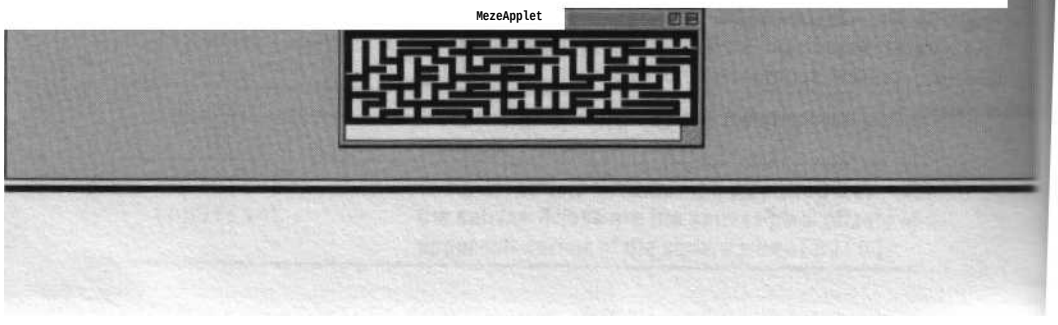
```
if (top - (sq & TOP) !- 0)
        offscr.fillRect(xoff, yoff, sgl.nuid-, LineHyt);
```

This code also sets the local variable `top` to `true` if it draws the top corners. 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 by 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 p a t h [ ] [ ] array. The bytes in p a t h [ ] [l have a bit set for each side of the corresponding square in the ma ze [ ] [ ] 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 p a t h [ ] [1. pa i ntOffscreenImage( ) calls the drawPathSquare( ) method in Maze to draw a single path square. Because some subclasses of Ma ze may want to calculate maze solutions but not display them, p a i n t O f f s c r e e n I ma g e ( ) only calls d r a w P a t h S q u a r e ( ) if the boolean field d i s p l a y P a t h is set.

d r a w P a t h S q u a r e ( ) uses pixel sizes and offsets that you initialize in r e s e t M a z e ( ). p W i d and p Hy t 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 M a z e like this:

```
protected int pWid, pHyt, pxoff, pyoff;
```

You initialize these fields by adding the following code to resetMaze ( ):

```
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)  >> l;
```

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:

```
int pw = sgWid - lineWid;
```

You use the pixel width pw to calculate the path width like this:

```
pWid = (pw & 1) == 0 ? Math.max(2, (pw >> 1) & -1)
                       Math.max(1, (pw >> 1) 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.

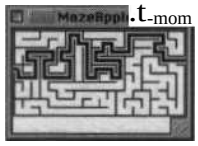


Figure 7-9: Displaying a solved wall maze.

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 sole, maze while it is still generating it. Figure 7-10 shows a block maze being solved while it is still being generated.

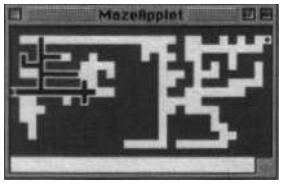


Figure 7-10: Solving a partially generated maze.

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 handle the different thread operations.

Because the only function of the class is to execute a thread,, you extend the `MazeThread` 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.MIR'PRIORITY + 1);
    if (solve)
      maze.traverse(show);
    else
      maze.generate(show);

1  // class MazeThread
```

Notice that if the operation is being animated, the r u n ( ) method sets the thread priority to M I N_P R I 0 R I TY + 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 M a z e A p p l e t class accepts certain HTML parameters. CD Chapter 1 discusses how to pass parameters to an applet from HTML. M a z e A p p 1 e t accepts the following parameters:

Parameter Name	What It Specifies
LI NEWIDTH	The width and height of the lines (1 i neWi d, 1 i neHyt). You only want to specify line width for wall mazes. The line width defaults to zero for block mazes and one for wall mazes.
MAZEWIDTH	The number of grid squares wide to make the maze. The maze width defaults to 30 squares.

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 "Wa IIMaze".

For example, you add the following applet tag to your HTML document block maze that is 20 squares wide and 15 squares tall:

```
<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 12 pixels tall with a 1-pixel border around the maze. MazeApp1et derives these pixel dimensions from the HTML tag like this:

```
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
```

MazeAppl et generates a new maze when the user clicks the maze with mouse button. If the user holds the Shift key down while clicking the on the maze and the maze has added the solution square to the maze. =me applet solves the maze. mouseDown( ) spawns a thread to generate or the maze. (CD Chapter 2 shows how to spawn threads from event handiew', such as mouseDown ( ).)

The Maze Appl et class is included on the CD.

Chapter 8

2-D Sprite Maze

-------oo 0x0*0*00.* * .a*0x0$.0*0...&000.0x0

~ This **Chapter**

Modeling game elements with sprites

Managing sprites with a sprite engine

Dusplaying and animating sprites

l ecting and handling sprite collisions

long game elements some intelligence

meting a 2-D maze game using sprites

▮ ao . Os . a . *0 #a*0 *0 0o * 000 0#0 +.

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 B 1 o c k M a z e class from Chapter 7.

Gentleman, Start Your Sprite Engines!

Sprite engine is just a fancy term for a data structure that keeps track ot sprites and tells them when to perform certain operations such as draw g moving, or animating. When applied to software, the term *engine* identid code that stands on it's own and is general enough to be used in a varieft ·O applications. A well-constructed sprite engine (which of course includes ther 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 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 PIR 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:

```
import java.awt.*;
public interface Sprite f
  void       setSpriteEngine (SpriteEngine se);
  boolean    updateSprite ();
  Rectangle drawSprite (Graphics g);
  Rectangle collisionBox ();
  Rectangle eollideWith (Object obj);
1  ll end interface Sprite
```

The sprite engine uses the S p r i t e interface methods to manage sprites. Table 8-1 shows the responsibilities for each of the five methods the `Spri to` interface defines.

Table 8-1 The Sprite Interface Methods

Method	***What the Sprite Engine Expects It To Do***
`setSpriteEngine()`	The sprite engine passes this method a reference to the engine when the sprite is added to the engine, and calls it with `nul l` when the sprite is removed from the engine. The sprite uses the reference to call methods in the sprite engine.
`updateSprite()`	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 `update Spri te( )`. `updateSpri te( )` returns `true` if you change the sprite's collision box, `f a l s e` if you don't.

(continued)

Table 8-1 *(continued)*

Method	*What the Sprite Engine Expects It To Do*
`drawSprite()`	The sprite engine passes this method a graphics context in which to draw the sprite. It returns a `Rectangle` representing the region drawn to the screen or `null` if it didn't draw anything.
`collisionBox()`	This method returns a `Rectangle` containing the sprite's collision box. The sprite engine uses the collision box to determine whether the sprite collides with other sprites. `collisionBox()` returns `null` to indicate that this sprite doesn't collide with other sprites.
`collideWith()`	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, `collideWith()` returns the possibly changed collision box for the sprite.

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 sprtse `collideWith()` returns the new collision box or `null` if the sprite engine manr not check for any more collisions with a given sprite during the current updat

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, th 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 collisim 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 ()        I return x; )
  public double centerY ()          return y; l
  public int    spriteWidth t)    I return width; )
  public int    spriteHeight ()   1 return height;

  public void setSpriteEngine (SpriteEngine se)
    spriteEngine = se;


  public boolean updateSprite () I return false; )
  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)
  I return collisionBox();

}  // end class SpriteObject
```

Spri t e 0 b j e c t 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

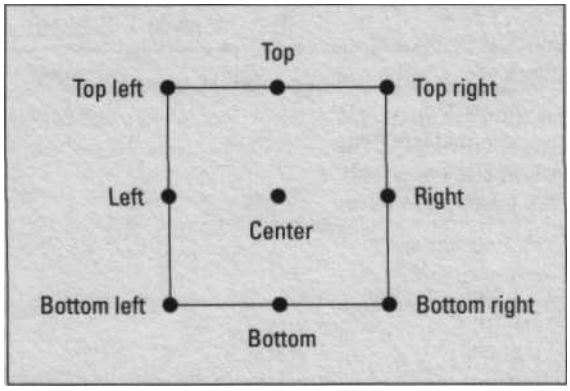


Figure 8-1: The nine ways to position a sprite's anchor point.

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 crawel across a ceiling.

Moving sprites around the play field

Notice that S p r i t e 0 b j e c t uses d o u b 1 e 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 floa point delta x and y values to smoothly move objects at any speed and in m direction.

You give your sprites motion by changing their position in the method u p d a t e S p r i t e () . Here is an example of how you add simple vector motion to your sprite:

```
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 I sprite engine handles all the collision detection. All updateSpri tell' 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 c o 1 1 i d e W i t h ( ) methods for each sprite involved in the collision. You implement col 1 i deWi th ( ) to resolve the collision. (The "Implementing a sprite" section earlier in this chapter discusses how the sprite engine calls c o 1 1 i d e W i t h ( ).)

The `Object Detector` class extends `Spri teObject`. 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 S p r i t e M a z e game presented in the "Building on the `B l o c k M a z e` 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 `Ob j ectDetect or` 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 `Ob j ectDetector` class:

```
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();
  }
   11 end class ObjectDetector
```

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 col 1 i s i onBox ( ) that defines the detection perimeter around the master sprite. The slave sprite only needs to implement the col 1 i s i o n Box ( ) and c o l l i d e W i t h ( ) methods; you leave the other methods empty like the corresponding methods in the Spri teOb j ect class.

Displaying sprites

You display a sprite by implementing the Sprite method d r a w S p r - : _ The sprite engine passes d r a w S p r i t e ( ) the graphics context in which draw the sprite. As an example, the following R o u n d S p r i t e class draws sprite as a colored oval:

```
import java.awt.*;
class RoundSprite extends SpriteObject I
  protected Color color;
  RoundSprite (double x, double y, int w, int h, Color c) i
    super(*, 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():

      end class RoundSprite
```

Notice in the call to f i 1 10 v a 1 ( ) that d r a wS p r i t e ( ) translates the spr s center anchor point to the upper-left corner of the sprite before drawing to match the upper-left coordinates that f i 1 10 v a 1 ( ) 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 zippin' around the screen. Here's an I ma g e S p r i t e class that you use to create sprites from loaded images:

```
import java.awt.*;
class ImageSprite extends SpriteObject I
  protected Image image:
  ImageSprite (Image image, double x, double y)
    super(x, y, 0, 0);
    setImage(image);


  public void setImage (Image img) I
    image = img;
    width = img.getWidth(null):
    height = img.getHeight(null);
```