

Learn Programming

Now!

Microsoft

XNA Game Studio 4.0

```

tell application "finder"
    set errorstring to "synchronization pro
    please try again with 2 folders."
    SearchOnlinePart
    If (count items in x) = 2 then
        display dialog errorstring buttons "ok" default button 1
    return item 1 of the variable with a new
    end if
    set y to item 1 of x
    if last character of (y as text) is "." then --it's a folder
        set y to item 2 of y
    SearchOnlinePart = New SearchOn
    if last character of (x as text) is "." then --it's a folder
        syncfolders(x, y) of me
    'site the control on the host user
    display dialog errorstring buttons "ok" default by
    Me.TargetPanel.Controls.Add(Me.m_
    Me.m_SearchOnlinePart.Dock = Docked
    display dialog ErrorString buttons "ok" default button 1
EndIf
end tell

Return Me.m_SearchOnlinePart
EndOf
on syncfolders(folder1, folder2)
    syncem(folder1, folder2)
    syncem(folder2, folder1)
end syncfolders

```

▶ *Design and build your own games for Xbox 360®, Windows® Phone 7, or your PC*

Rob Miles

**Microsoft® XNA®
Game Studio 4.0: Learn
Programming Now!**

PUBLISHED BY
Microsoft Press
A Division of Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399

Copyright © 2011 by Microsoft Corporation

All rights reserved. No part of the contents of this book may be reproduced or transmitted in any form or by any means without the written permission of the publisher.

Library of Congress Control Number: 2001012345
ISBN: 978-0-7356-5157-9

Printed and bound in the United States of America.

Microsoft Press books are available through booksellers and distributors worldwide. For further information about international editions, contact your local Microsoft Corporation office or contact Microsoft Press International directly at fax (425) 936-7329. Visit our Web site at www.microsoft.com/mspress. Send comments to mspinput@microsoft.com.

Microsoft and the trademarks listed at <http://www.microsoft.com/about/legal/en/us/IntellectualProperty/Trademarks/EN-US.aspx> are trademarks of the Microsoft group of companies. All other marks are property of their respective owners.

The example companies, organizations, products, domain names, e-mail addresses, logos, people, places, and events depicted herein are fictitious. No association with any real company, organization, product, domain name, e-mail address, logo, person, place, or event is intended or should be inferred.

This book expresses the author's views and opinions. The information contained in this book is provided without any express, statutory, or implied warranties. Neither the authors, Microsoft Corporation, nor its resellers, or distributors will be held liable for any damages caused or alleged to be caused either directly or indirectly by this book.

Acquisitions Editor: Devon Musgrave

Developmental Editor: Devon Musgrave

Project Editor: Valerie Woolley

Editorial and Production: Waypoint Press

Technical Reviewers: Nick Gravelyn, Kurt Meyer; Technical Review services provided by Content Master, a member of CM Group, Ltd.

Cover: Girvin

Body Part No. X17-37448

To Jake, a great dog who is much missed.

Table of Contents

Acknowledgments	xvii
Introduction	xix
Who This Book Is For	xix
System Requirements	xx
Code Samples	xx
Errata and Book Support	xx
We Want to Hear from You	xxi
Stay in Touch	xxi

Part I **Getting Started**

1 Computers, C#, XNA, and You	3
Introduction	3
Learning to Program	3
Becoming a Great Programmer	4
How the Book Works	4
C# and XNA	5
Getting Started	6
Installing the Development Environment and the XNA Framework	6
Setting Up a PC to Run XNA Games	7
Setting Up an Xbox 360 to Run XNA Games	7
Setting up a Windows Phone to run XNA games	10
Writing Your First Program	12
Creating Your First Project	12
Running Your First Program	14
Stopping a Program	16

What do you think of this book? We want to hear from you!

Microsoft is interested in hearing your feedback so we can continually improve our books and learning resources for you. To participate in a brief online survey, please visit:

www.microsoft.com/learning/booksurvey/

Storing Games on the Xbox 360 or Windows Phone	17
Running the Same XNA Game on Different Devices	17
Conclusion	19
Chapter Review Questions.	20
2 Programs, Data, and Pretty Colors.	21
Introduction.	21
Making a Game Program.	22
Statements in the <i>Draw</i> Method	23
Working with Colors.	24
Storing Color Values	24
Setting a Color Value	25
Controlling Color.	27
Games and Classes	27
Classes as Offices.	29
Game World Data	30
Storing Data in Computer Memory	31
Drawing by Using Our Color Intensity Variables.	32
Updating Our Colors.	33
Memory Overflow and Data Values.	35
Making a Proper Mood Light	36
Making Decisions in Your Program	37
The Completed Mood Light	41
Finding Program Bugs	42
Conclusion	44
Chapter Review Questions.	44
3 Getting Player Input.	47
Introduction.	47
Reading a Gamepad	48
Gamepads and Classes.	48
Finding a Gamepad.	50
Testing the Gamepad Status	51
Using the Keyboard	54
Stopping the Game with the Escape Key	54
Using a Gamepad and a Keyboard at the Same Time.	55
Adding Vibration.	56
Controlling the Vibration of a Gamepad	56
Testing Intensity Values.	57

Program Bugs	61
Conclusion	63
Chapter Review Questions.....	64

Part II **Images, Sound, and Text**

4	Displaying Images.....	67
	Introduction.....	67
	Resources and Content	68
	Getting Some Pictures	68
	Content Management Using XNA	69
	Working with Content Using XNA Game Studio	70
	XNA Game Studio Solutions and Projects	70
	Adding Content to a Project.....	72
	Using Resources in a Game	75
	Loading XNA Textures	75
	Positioning Your Game Sprite on the Screen.....	79
	Sprite Drawing with <i>SpriteBatch</i>	81
	Filling the Screen.....	83
	Conclusion	86
	Chapter Review Questions.....	86
5	Writing Text.....	87
	Introduction.....	87
	Text and Computers.....	87
	Text as a Resource.....	88
	Creating the XNA Clock Project.....	88
	Adding a Font Resource.....	88
	Loading a Font.....	91
	Drawing with a Font.....	92
	Changing the Font Properties.....	94
	Getting the Date and Time	95
	Making a Prettier Clock with 3-D Text	97
	Drawing Multiple Text Strings	97
	Repeating Statements with a <i>for</i> Loop	99
	Other Loop Constructions.....	101
	Fun with <i>for</i> Loops	101

Creating Fake 3-D	103
Creating Shadows Using Transparent Colors	104
Drawing Images with Transparency	105
Conclusion	106
Chapter Review Questions	106
6 Creating a Multi-Player Game	107
Introduction	107
Creating the Button-Bash Game	107
Level and Edge Detectors	111
Constructing the Complete Game	111
Adding Test Code	114
Conclusion	116
Chapter Review Questions	116
7 Playing Sounds	117
Adding Sound	117
Creating the Drum Pad Project	117
Capturing Sounds with Audacity	117
Storing Sounds in Your Project	119
Using Sounds in an XNA Program	121
Playing Background Music	123
Creating a RayGun	123
Conclusion	129
Chapter Review Questions	130
8 Creating a Timer	131
Making Another Game	131
Reaction Timer Bug	134
Finding Winners Using Arrays	136
Creating an Array	136
Using Data in an Array	137
Scanning an Array	138
Using an Array as a Lookup Table	140
Displaying the Winner	141
Conclusion	143
Chapter Review Questions	143

9	Reading Text Input	145
	Using the Keyboard in XNA	145
	Creating the Message Board Project	145
	Registering Key Presses	146
	The Keys Type	147
	Enumerated Types	148
	Working with Arrays, Objects, and References	148
	Values and References	149
	Arrays as Offices	149
	Say Hello to the Garbage Collector	151
	Using References and Values	151
	Why Do We Have References and Values?	153
	References and <i>GetPressedKeys</i>	153
	Displaying Keys	153
	Detecting Key Presses	155
	Decoding Key Characters	159
	Using the Shift Keys	160
	Editing the Text	161
	Conclusion	163
	Chapter Review Questions	163

Part III **Writing Proper Games**

10	Using C# Methods to Solve Problems	167
	Introduction	167
	Playing with Images	167
	Zooming In on an Image	167
	Creating a Zoom-Out	169
	Updating the Drawing Rectangle	170
	Creating a Method to Calculate Percentages	173
	Returning Nothing Using <i>void</i>	175
	Debugging C# Programs	179
	Hitting a Breakpoint	180
	Using Floating-Point Numbers in C#	183
	The Compiler and C# Types	184
	Compilers and Casting	185
	Expression Types	186

Stopping the Zoom.	188
Zooming from the Center	188
Conclusion	191
Chapter Review Questions.	192
11 A Game as a C# Program.	193
Introduction.	193
Creating Game Graphics	194
Projects, Resources, and Classes.	195
XNA Game Studio Solutions and Projects	195
The Program.cs File.	198
Renaming the <i>Game1</i> Class.	203
Creating Game Objects	205
Sprites in Games	205
Managing the Size of Game Sprites.	206
Moving Sprites.	209
Bouncing the Cheese	210
Dealing with Display Overscan.	211
Conclusion	213
Chapter Review Questions.	214
12 Games, Objects, and State.	215
Introduction.	215
Adding Bread to Your Game.	215
Using a Structure to Hold Sprite Information	216
Using the Gamepad Thumbsticks to Control Movement	218
Improving Programs Using Methods	219
Handling Collisions.	222
Making the Cheese Bounce off the Bat.	222
Strange Bounce Behavior.	223
Strange Edge Behavior.	224
Adding Tomato Targets	227
Tomato Collisions	229
Conclusion	232
Chapter Review Questions.	232

13	Making a Complete Game	233
	Introduction	233
	Making a Finished Game	233
	Adding Scores to a Game	233
	Adding Survival	235
	Adding Progression	236
	Improving Code Design	239
	Refactoring by Creating Methods from Code	240
	Refactoring by Changing Identifiers	241
	Creating Code Regions	244
	Creating Useful Comments	245
	Adding a Background	246
	Adding a Title Screen	247
	Games and State	247
	Using the State Values	248
	Building a State Machine	249
	Conclusion	252
	Chapter Review Questions	252
14	Classes, Objects, and Games	253
	Introduction	253
	Design with Objects	253
	An Object Refresher Course	254
	Cohesion and Objects	254
	Coupling Between Objects	257
	Designing Object Interactions	260
	Container Objects	261
	Background and Title Screen Objects	263
	Classes and Structures	264
	Creating and Using a Structure	264
	Creating and Using an Instance of a Class	265
	References	267
	Multiple References to an Instance	267
	No References to an Instance	268
	Why Bother with References?	268
	Value and Reference Types	269
	Should Our Game Objects Be Classes or Structures?	269

Creating a Sprite Class Hierarchy	271
The <i>BaseSprite</i> Class	271
Extending the <i>BaseSprite</i> to Produce a <i>TitleSprite</i>	272
Building a Class Hierarchy	273
Adding a Deadly Pepper	274
Creating a <i>DeadlySprite</i> Class	275
Conclusion	279
Chapter Review Questions.	280
15 Creating Game Components.	281
Introduction.	281
Objects and Abstraction	281
Creating an Abstract Class in C#.	282
Extending an Abstract Class	282
Designing with Abstract Classes.	284
References to Abstract Parent Classes.	284
Constructing Class Instances.	285
Constructors in Structures.	287
Constructors in Class Hierarchies	287
Adding 100 Killer Tangerines	289
Creating a <i>KillerSprite</i> Class.	290
Positioning the <i>KillerSprites</i> Using Random Numbers	290
Using Lists of References	293
Adding Artificial Intelligence	297
Chasing the Bread Bat	297
Adding Game Sounds.	302
From Objects to Components.	304
C# Interfaces	305
Creating an Interface	306
Implementing an Interface	307
References to Interfaces	307
Linking Bread, Cheese, and Tomatoes.	308
Designing with Interfaces	308
Conclusion	309
Chapter Review Questions.	309

16	Creating Multi-Player Networked Games	311
	Introduction	311
	Networks and Computers	311
	Starting with the Signal	311
	Building Up to Packets	312
	Addressing Messages	312
	Routing	313
	Calls and Datagrams	314
	Networks and Protocols	314
	Xbox Live	315
	Gamertags and Xbox Live	315
	System Link and XNA	316
	Bread and Cheese Pong	316
	Managing Gamer Profiles in XNA	317
	Ensuring a Gamer Is Signed In for Network Play	321
	Creating a Game Lobby	322
	Network Games and State	322
	Playing the Game	329
	The Completed Game	334
	Conclusion	334
	Chapter Review Questions	335

Part IV **Making Mobile Games for Windows Phone 7 with XNA**

17	Motion-Sensitive Games	339
	Introduction	339
	The Accelerometer	339
	What Does the Accelerometer Actually Do?	339
	Acceleration and Physics	340
	Making Sense of Accelerometer Readings	341
	Creating a “Cheese Lander” Tipping Game	343
	Game World Objects in “Cheese Lander”	343
	Getting Access to the Accelerometer Class from XNA	344
	Using the Accelerometer in an XNA Game	346
	Starting the Accelerometer	349

	Using Accelerometer Values in a Game.	349
	Using Vectors to Express Movement.	352
	Adding Friction	353
	Detecting Shaking.	354
	A Quick Digression About Threads and Synchronization	355
	Conclusion	357
	Chapter Review Questions.	357
18	Exploring Touch Input	359
	Introduction.	359
	The Windows Phone Touch Screen	359
	Getting Touch Input.	359
	Creating a Panic Button.	360
	Reading Touch Events	361
	Touch Location Types.	361
	Using the Location of a Touch	363
	Creating a Touch Drumpad	364
	Creating a <i>soundPad</i> Class for Each Drum Sound	364
	Storing <i>soundPad</i> Values in the Game.	365
	Drawing the Soundpads.	366
	Updating the Soundpads.	367
	Making the Soundpads Flash	368
	Creating a Shuffleboard Game	370
	The <i>PuckSprite</i> Class	370
	Conclusion	377
	Chapter Review Questions.	377
19	Mobile Game Development	379
	Introduction.	379
	The Windows Phone.	379
	The Windows Phone Marketplace	379
	Maximizing the Phone Battery Life in XNA Games.	380
	Setting the Update Rate of a Game	380
	Dealing with Changes in Phone Orientation	381

Selecting Orientations in an XNA Game	381
Getting Messages When the Orientation Changes.	382
Using a Specific Display Size for Windows Phone Games	383
Hiding the Windows Phone Status Bar	384
Stopping the Screen Timeout from Turning Off Your Game.	384
Creating a Phone State Machine	385
Games and States	385
Handing Incoming Phone Calls	390
Detecting Phone Calls	392
A Game as a Windows Phone Application	393
The Windows Phone Back and Start Buttons.	393
Starting New Programs with the Start Button.	396
Using Isolated Storage to Store Game State	397
Getting Your Games into the Marketplace.	403
The Windows Phone Marketplace	403
Registering for the App Hub.	404
Using a Windows Phone Device.	404
Creating Games for Sale.	405
Conclusion	405
Chapter Review Questions.	405
Answers to the Chapter Review Questions.	407
Index.	427



What do you think of this book? We want to hear from you!

Microsoft is interested in hearing your feedback so we can continually improve our books and learning resources for you. To participate in a brief online survey, please visit:

www.microsoft.com/learning/booksurvey/

Acknowledgments

I'm not sure if you are meant to have fun writing books, but I do. Thanks to Devon Musgrave, Ben Ryan, Valerie Woolley, and Steve Sagman for making everything fit so well together and to Kurt Meyer and Nick Gravely for making sure it all makes sense. I must also mention the XNA team who keep making a great thing better, year on year, and the Windows Phone team who have made something amazing.

Introduction

With Microsoft XNA, Microsoft is doing something really special. It is providing an accessible means for people to create programs for the Windows PC, Xbox 360, and Windows Phone. Now pretty much anyone can take a game idea, run it on a genuine console, and even send it to market in Xbox Live or the Windows Phone Marketplace.

This book shows you how to make game programs and run them on an Xbox 360, a Microsoft Windows PC, or a Windows Phone device. It also gives you an insight into how software is created and what being a programmer is really like.

Who This Book Is For

If you have always fancied writing software but have no idea how to start, then this book is for you. If you have ever played a computer game and thought, “I wonder how they do that?” or, better yet, “I want to make something like that,” then this book will get you started with some very silly games that you and all your friends can have a go at playing and modifying. Along the way, you’ll also get a decent understanding of C#, which is a massively popular programming language used by many thousands of software developers all over the world. The C# skills that you pick up in this book can also be used as the basis of a career in programming, should you find that you really enjoy writing programs. And because the design of the C# language is very similar to C, C++, and Java, you will find that your skills can be used with them too.

The book is structured into 19 chapters, starting with the simplest possible XNA program and moving on to show you how to use the Xbox gamepad, the keyboard, sounds, graphics, and network in your games. In the course of learning how to use C# and XNA, you create some very silly games, including Color Nerve, Mind Reader, Gamepad Racer, Bread and Cheese, and Button Bash. You can even download the full versions of these games from <http://www.versillygames.com> and use them at your next party. The final section shows you how to take your programming skills and use them to create games for the Windows Phone device.

With this book, I show you that programming is a fun, creative activity that lets you bring your ideas to life.

System Requirements

You need the following hardware and software to build and run the code samples for this book. Chapter 1, “Computers, C#, XNA, and You,” explains how to set up your environment.

- A Windows PC with 3-D graphics acceleration if you want to run your XNA games on your PC.
- Microsoft Windows Vista or Windows 7.
- Microsoft Visual Studio 2010 C# Express Edition for Windows Phone, Visual Studio 2010 Standard Edition, Visual Studio 2010 Professional Edition, or Visual Studio 2010 Team Suite.
- To test your games on a console, you need an Xbox 360 fitted with a hard disk. Your Xbox 360 must be connected to Xbox Live, and you need to join the App Hub. You will find out how to do this in Chapter 1.
- If you have a Windows Phone you can run XNA games on that as well. Any Windows Phone device can be connected to your PC so you can load your XNA games into it.

Code Samples

All the code samples discussed in this book can be downloaded from the book’s detail page, located at:

<http://oreilly.com/catalog/9780735651579>

Display the detail page in your Web browser, and follow the instructions for downloading the files.

There are also code samples and games at <http://www.veryillygames.com>.

Errata and Book Support

We’ve made every effort to ensure the accuracy of this book and its companion content. If you do find an error, please report it on our Microsoft Press site at Oreilly.com:

1. Go to <http://microsoftpress.oreilly.com>.
2. In the **Search** box, enter the book’s ISBN or title.
3. Select your book from the search results.
4. On your book’s catalog page, under the cover image, you’ll see a list of links.
5. Click **View/Submit Errata**.

You'll find additional information and services for your book on its catalog page. If you need additional support, please e-mail Microsoft Press Book Support at mspinput@microsoft.com.

Please note that product support for Microsoft software is not offered through the addresses above.

We Want to Hear from You

At Microsoft Press, your satisfaction is our top priority, and your feedback our most valuable asset. Please tell us what you think of this book at:

<http://www.microsoft.com/learning/booksurvey>.

The survey is short, and we read *every one* of your comments and ideas. Thanks in advance for your input!

Stay in Touch

Let's keep the conversation going! We're on Twitter: *<http://twitter.com/MicrosoftPress>*.

Chapter 3

Getting Player Input

In this chapter, you will

- Find out how Microsoft XNA represents the gamepads and keyboards.
- Discover the C# language structures that let us get player input.
- Write some really silly games and scare people with them.

Introduction

You now know the basics of computer game programming. You know that a program is actually a sequence of statements, each of which performs a single action. You have seen that statements are held inside methods, each of which performs a particular task, and that methods are held in classes along with data. The program itself works on data values, which are held in variables of a particular type, and the program can make decisions based on the values that the variables have. (If none of this makes much sense, reread Chapter 2, “Programs, Data, and Pretty Colors,” until it does.)

Now you are going to expand your understanding to include how to receive input from the outside world so that games can actually react to what the player does. You shall see that once we have done this, a number of possibilities open up, and you can create some truly silly games, including “Color Nerve,” “Mind Reader,” “The Thing That Goes Bump in the Night,” and “Gamepad Racer.”

Program Project: A Mood Light Controller

In Chapter 2, you created a light that changes color over time. I also mentioned that this is the kind of thing that will be used in the starships of the future. A color-changing light is not all that useful for reading books, but it’s great for setting moods; what our starship captain really needs is a light that she can set to any color. So now you are going to make a lamp that can be controlled by an Xbox gamepad. The user presses the red, blue, green, and yellow buttons on the gamepad to increase the amount of that color in the light. To make this work, you have to discover how to read the gamepad.

Before you start looking at gamepads, though, you need to decide how the program will actually work. Consider the following statement of C# from the previous mood-light program, which is part of the Update method:

```
if (redCountingUp) redIntensity++;
```

This is one of the tests that controls the intensity of the red part of the color. What it is saying is “If the Boolean value `redCountingUp` is `True`, increase the value of `redIntensity` by 1.” The statement is processed each time `Update` is called (at the moment that is 60 times a second), so this means that if `redCountingUp` is `True`, the red intensity of the screen gets progressively brighter over time.

You want to write some code that says, “If the red button on Gamepad 1 is being pressed, increase the value of `redIntensity` by 1.” Then, if the player holds down the button, the screen gets redder. So all you have to do is change this test to read the button on the gamepad, and you can create a user-controlled light easily.

Reading a Gamepad

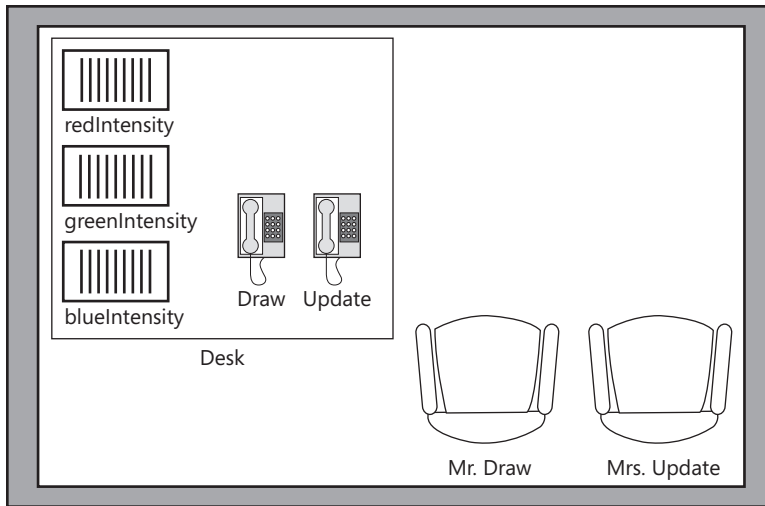
The gamepads are actually very complex devices. They are connected to the host device either by a universal serial bus (USB) cable or by a wireless connection. As far as you are concerned, the way that programs work with gamepads does not depend on how they are connected. The connection to a gamepad can be used to read the buttons and joysticks and can also be used to send commands to the gamepad—for example, to turn the vibration effect on and off. The Xbox and XNA provide support for up to four gamepads connected simultaneously.

Gamepads and Classes

The gamepad information is represented in XNA by means of a class called `GamePadState`. The job of this class is to provide the connection between the program and the physical gamepad that the player is holding. To understand how you are going to use this, you have to learn a bit more about how classes work.

You have already seen what a class is in the section “Games and Classes” in Chapter 2. A class contains data (variables that can hold stuff) and methods (code that can do stuff). You can think of a class as an office, with a desk holding the variables and people acting as the methods. Figure 3-1 shows the office plan for the class `Game1`, which you have seen is the basis of an XNA game.

This class contains some variables on the desk (in this case, the background color intensities) and two methods, which we have called Mr. Draw and Mrs. Update. Each method has a corresponding telephone. Programs can place calls to the telephones to request that the method perform the required task.



Game1 office

FIGURE 3-1 The Game1 class as an office plan.

The Great Programmer Speaks: Classes Are Not Really Offices Our Great Programmer has been reading these notes and finds them quite amusing. She says that classes are not exactly like offices, but she thinks that for the purpose of getting an understanding of how programs are constructed, it is okay to regard them as such.

When an XNA game starts, the XNA system makes an *instance* of the Game1 class that it then can ask to Draw and Update. When an instance of a class is created, the instructions for the methods that it contains are loaded into memory and space is set aside for the data variables that the instance holds.

The class files that you write give the plans for the class so when the program runs, instances of each class can be created. In real life, you would make a game office by building a room, putting a desk and some telephones in the room, and then hiring a Mr. Draw and a Mrs. Update. The process of making an instance of a class is similar. However, to save memory, the running program uses only one copy of the method code, which is shared among all the instances of a class.



Note It is important to remember that this happens when a program runs. The process of creating instances of classes is not performed by the compiler. The job of the compiler is to convert your C# source code into instructions that the target device runs. By the time that your program has control, the compiler has done its job, and the computer is just running the machine language output that the compiler produced.

Finding a Gamepad

XNA also looks after a lot of other things when a game is running, one of which is the `GamePad` class connected to all the gamepads. You don't have to know how the gamepad is actually connected; for all you know, it might use tiny pixies traveling up and down the wires carrying pixie notes written on pixie paper saying, "Master has pressed the Red Button," but then again it might not. Figure 3-2 shows how the `GamePad` class would look if it were an office.

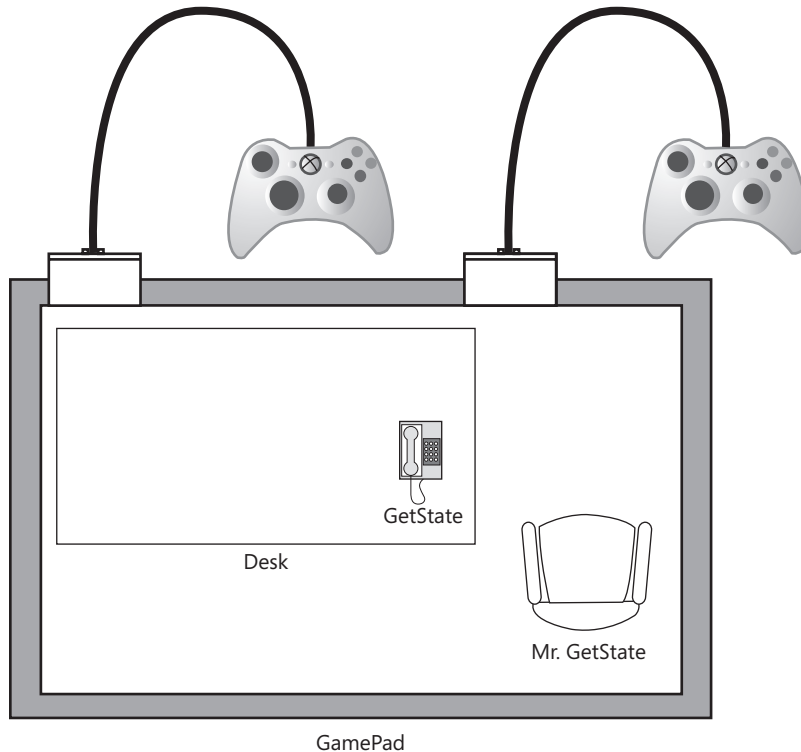


FIGURE 3-2 The `GamePad` class as an office.

The `GamePad` class contains a method called `GetState`, which gets the state of one of the gamepads. When `GetState` is called, it looks at one of the gamepads, reads its settings, and then sends information back for use in the statement it was called from.

The `GetState` method is supplied with a parameter that identifies the gamepad to be read. A *parameter* is a way that a call can give information to a method. You have seen these before; in your very first programs, you were passing `Color` parameters into the `Clear` method to select the color of the screen that you wanted.

In the case of the `GetState` method, the parameter identifies the gamepad that you want to read. If you are thinking in terms of offices, you can think of a parameter as part of the instructions that come down the telephone. When the phone rings and Mr. `GetState` answers it,

he is asked, "Get me the state of Gamepad 1." The information about the state of the gamepad is sent back in a `GamePadState` structure, which is shown in Figure 3-3.

GamePadState	
Buttons	
Green A	ButtonState.Pressed
Red B	ButtonState.Released
Blue X	ButtonState.Released
Yellow Y	ButtonState.Released
Start	ButtonState.Released
Back	ButtonState.Released

FIGURE 3-3 GamePadState structure with the green A button pressed.

You can think of this as a set of items filled in on a form if you wish, but actually it is a C# structure that contains the data members shown in Figure 3-3, as well as some other data.

So, if Mrs. Update wants to know the state of one of the gamepads on the Xbox, she calls the `GetState` method in the `GamePad` class and asks, "Can you give me the state of the gamepad for Player 1, please?" Mr. `GetState` jumps up, fills in a "GamePadState" form, and sends it back to her. Figure 3-4 gives the breakdown of the C# statement that gets the state of a gamepad into a variable of type `GamePadState`.

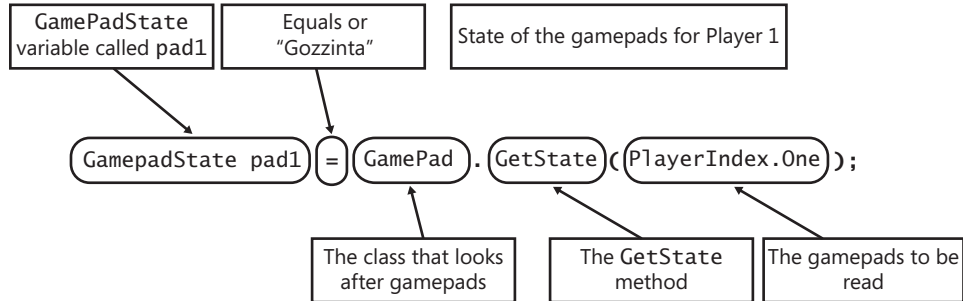


FIGURE 3-4 Getting the status of a gamepad.

Testing the Gamepad Status

Now that you have the status, you can use it in the program to see if a button has been pressed. Figure 3-5 shows the breakdown of the C# statement that will perform the test.

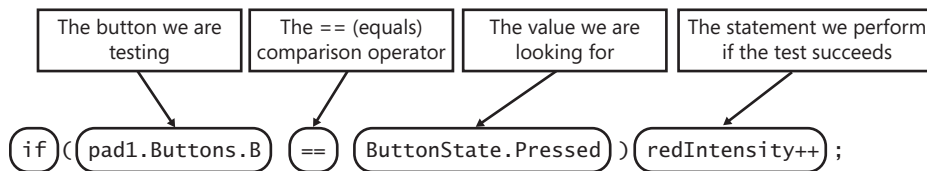


FIGURE 3-5 Testing a button on a gamepad.

This compares the state of the red button B with the value `ButtonState.Pressed`. If the two are equal, this means that the button is down, and the `Update` method must make the red intensity bigger. You can then use the same principle to manage the blue and green values, which means that you now have an `Update` method that looks like the following:

```
protected override void Update(GameTime gameTime)
{
    // Allows the game to exit
    if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
        this.Exit();

    GamePadState pad1 = GamePad.GetState(PlayerIndex.One);

    if (pad1.Buttons.B == ButtonState.Pressed) redIntensity++;
    if (pad1.Buttons.X == ButtonState.Pressed) blueIntensity++;
    if (pad1.Buttons.A == ButtonState.Pressed) greenIntensity++;

    base.Update(gameTime);
}
```

The only problem with the `Update` method described here is that the program doesn't handle the yellow button yet. When the yellow button is pressed, the program needs to increase the green and the red intensities; that is, it must perform two statements if the condition is true. It turns out that doing so is very easy; you can just put the two statements into a block that is controlled by the condition, as shown here:

```
if (pad1.Buttons.Y == ButtonState.Pressed)
{
    redIntensity++;
    greenIntensity++;
}
```

You have seen blocks before; the body of a method (the bit that does the work) is a block. In *C#* terms, a *block* is a number of statements that are enclosed in curly braces. The code shown here performs both statements if the condition is true because they are in a block controlled by the condition.

The Great Programmer Speaks: Blocks Rock Our Great Programmer tends to use blocks after `if` conditions even when she doesn't actually need to. She says that it makes the program text clearer, and that it is much easier to add extra statements later if you need to.

If you put the preceding statements into the `Update` method of one of your earlier `Mood Light` programs, you get compiler warning messages because the new version of `Update` doesn't use all the variables that were created for previous versions of the program. To get

rid of these warnings, you must delete the statements that create the unused variables. The Great Programmer doesn't like it when programs have variables in them that are not used. She says this looks unprofessional, and I agree with her.

Sample Code: Manual MoodLight All the sample projects can be obtained from the Web resources for this text, which can be found at <http://oreilly.com/catalog/9780735651579>. The sample project in the directory "01 Manual MoodLight" in the resources for this chapter implements the Update method, as shown in this section. You can increase the brightness of the colors on the screen by pressing the buttons on the gamepad.

Game Idea: Color Nerve

Every now and then, we are going to try out a game idea. These start out very simply and then build up to more complicated and interesting games. You can use the Manual MoodLight code to create your first game. The game uses something we saw in Chapter 2. You noticed that if you keep making a value bigger, there comes a point where it won't fit in the memory store allocated for it, and then it overflows. This is what caused the screen to go from bright white to black. However, you can use this to create our first "Very Silly Game."

Color Nerve is a game for two or more players. The players take turns pressing one or more buttons on the gamepad. (The other players must watch carefully to make sure that they actually do press at least one button.) Each player can press as many buttons as he wants for as long as he wants during his turn, but if the screen changes suddenly (because one of the color values has gone from 255 to 0), he is out, and the game continues. The last player left in the game is the winner.

This game can be very tactical. Players can press the buttons for very short times, or at the start of the game, they can show their nerve by holding the buttons down for longer periods, trying to cause problems for the next player. They can also try to work out which color has wrapped around so that they can press that button when it is their turn. The game works very well at parties, any number of people can take part, and the rules are very easy to understand. In Chapter 4, "Displaying Images," you will improve the game to add pictures as well as a plain screen.

Using the Keyboard

XNA works with keyboards as well as with gamepads. You might be surprised to learn that you can plug a USB keyboard into an Xbox 360 and use it just as you'd use the keyboard on the PC. If you want the program to work with the keyboard, you can add code that does this, as shown here:

```
KeyboardState keys = Keyboard.GetState();

if (keys.IsKeyDown(Keys.R)) redIntensity++;
if (keys.IsKeyDown(Keys.B)) blueIntensity++;
if (keys.IsKeyDown(Keys.G)) greenIntensity++;
if (keys.IsKeyDown(Keys.Y))
{
    redIntensity++;
    greenIntensity++;
}
```

Note that the process is very similar to how the gamepad works, but there are slight differences. You don't need to tell the `GetState` method on the `Keyboard` which keyboard to read because XNA supports only a single keyboard. The `KeyboardState` item that is returned from the call is not actually a piece of paper; instead, it is an object that provides methods that the program can use to discover whether a particular key is pressed. Rather than seeing if the state of a button is set to the value `ButtonState.Pressed`, the program can call the method `IsKeyDown`. You supply the `IsKeyDown` method with a parameter that identifies the key you are interested in, as follows:

```
if (keys.IsKeyDown(Keys.R)) redIntensity++;
```

This code is a conditional statement that increases the value of `redIntensity` if the R key is pressed. The method `IsKeyDown` returns `true` if the key is down and `false` if not. You can, therefore, use it to control the update of the `redIntensity` value.

Stopping the Game with the Escape Key

The `Update` method that is created when you make a new XNA game contains a test that checks for the Back button on gamepad 1 and calls the `Exit` method to stop the game when the Back button is pressed. If you are using a keyboard instead of a gamepad you will not be able to press this button to stop the game. You can add a test for the Escape key on the keyboard. This key is a "control" key, in that it does not actually relate to a printable character, but is designed to signal an action you want the program to take. Other control keys include the Enter key and the Backspace key. You can use the same `IsKeyDown` method to test for the Escape key.

```
if (keys.IsKeyDown(Keys.Escape)) Exit();
```

This code stops the game when the Escape key is pressed.

Using a Gamepad and a Keyboard at the Same Time

If you want to use a gamepad and a keyboard simultaneously, you have to test for both. This means that the Update method now looks like this:

```
protected override void Update(GameTime gameTime)
{
    GamePadState pad1 = GamePad.GetState(PlayerIndex.One);

    if (pad1.Buttons.Back == ButtonState.Pressed) Exit();
    if (pad1.Buttons.B == ButtonState.Pressed) redIntensity++;
    if (pad1.Buttons.X == ButtonState.Pressed) blueIntensity++;
    if (pad1.Buttons.A == ButtonState.Pressed) greenIntensity++;
    if (pad1.Buttons.Y == ButtonState.Pressed)
    if (pad1.Buttons.B == ButtonState.Pressed) redIntensity++;
    {
        redIntensity++;
        greenIntensity++;
    }

    KeyboardState keys = Keyboard.GetState();

    if (keys.IsKeyDown(Keys.Escape)) Exit();

    if (keys.IsKeyDown(Keys.R)) redIntensity++;
    if (keys.IsKeyDown(Keys.B)) blueIntensity++;
    if (keys.IsKeyDown(Keys.G)) greenIntensity++;
    if (keys.IsKeyDown(Keys.Y))
    {
        redIntensity++;
        greenIntensity++;
    }
    base.Update(gameTime);
}
```

This code is not good because you are doing the same thing twice, just triggered in a different way. The Great Programmer, if she ever saw this, would not be impressed. Fortunately C# provides a way that a program can combine two conditions and then perform some code if either condition is true. This way of combining conditions is called the *OR* logical operator because it is true if one thing or the other is true, and it is written in the program as two vertical bars (||):

```
GamePadState pad1 = GamePad.GetState(PlayerIndex.One);
KeyboardState keys = Keyboard.GetState();

if (pad1.Buttons.B == ButtonState.Pressed ||
    keys.IsKeyDown(Keys.R)) redIntensity++;
```

The *OR* logical operator is placed between two Boolean expressions that can be either true or false. If one or the other expression is true, the combined logical condition works out to be true.

In this code, if the red button is pressed on the gamepad *or* the R key is pressed on the keyboard (or both), the `redIntensity` value increases. This is exactly what you want, and it means that Color Nerve can now be played with the gamepad or the keyboard (or both at the same time). Logical operators are so called because they produce logical rather than numerical results. There are other logical operators that you will use as you create more complex programs.



Note If you find this logical operator stuff hard to understand, just go back to the problem that you are trying to solve. You want the program to perform a statement (`redIntensity++`) if the red key is pressed on the gamepad *or* if the R key is pressed on the keyboard. So you use the *OR* operator (`||`) to combine the two tests and make a condition that triggers if one or the other condition is true.

Sample Code: Color Nerve The sample project in the directory “02 Color Nerve” in the resources for this chapter implements the game. You can adjust the colors of the screen by pressing the gamepad buttons or a key on the keyboard.

Adding Vibration

The communication between the gamepad and the game works in both directions. Not only can you read buttons on the gamepad, but also you can send commands to the gamepad to turn on the vibration motors. Again, you don't have to know exactly how these messages are delivered; all you need to know is the features of XNA that are used to control this vibration effect.

This means you can make your Color Nerve game even more exciting by making the gamepad vibrate when the intensity values are getting close to their limits. It is interesting how features like this can enhance even a simple game. You will be using the vibration effect on the gamepads quite a lot in the next few games.

Controlling the Vibration of a Gamepad

The `GamePad` class provides a method called `SetVibration` that lets a program control the vibration motors:

```
GamePad.SetVibration(PlayerIndex.One, 0, 1);
```

The `SetVibration` method uses three parameters. The first one identifies which gamepad you want to vibrate. The second parameter is a value between 0.0 and 1 that controls the vibration of the left motor. The bigger the number, the more the gamepad vibrates. The third parameter controls the right motor in the same way as the left one. The statement shown

here would set the right motor of Gamepad 1 vibrating at full speed. The left motor is the low-frequency vibration, and the right motor is the high-frequency vibration.

If you think of the GamePad class/object having a man called Mr. SetVibration, this means that he would be told which gamepad to vibrate and the settings for the left and right motors. Once the method has been called, the gamepad starts to vibrate, and it keeps vibrating until you call the method again to change its setting. In other words, you can think of the SetVibration method as a switch that can be set to a number of different positions. Initially, both of the gamepad motors are set at 0, which means no vibration.

Testing Intensity Values

The game needs to decide when to turn on the vibration. To do this, it must test the intensity values and turn on the vibration motor if any of them is getting too large. The program can decide to turn on the motors if any of the red, green, or blue intensity values is greater than 220. To do this, the program must test the intensity values as follows:

```
if (redIntensity > 220)
{
    GamePad.SetVibration(PlayerIndex.One, 0, 1);
}
```

This code shows another form of condition. In the previous examples, the conditions have been checking to see if two values are equal. This code tests if one value is greater than another. The greater-than sign (>) is another logical operator. Placed between two values, it returns true if the value on the left is greater than the value on the right and false if not. That is exactly what you want.

Using the preceding code, the gamepad starts to vibrate using the right motor when the red intensity value goes above 220. If you add this code to the Update method in the Color Nerve game, you find that if you increase the red value, the gamepad starts to vibrate. Unfortunately, our program has a bug. When the red intensity value returns to 0, the vibration does not stop. You need to add some code that turns off the motor when the intensity value is less than 220. It turns out that this is very easy to do—you can add an else part to the condition:

```
if (redIntensity > 220)
{
    GamePad.SetVibration(PlayerIndex.One, 0, 1);
}
else
{
    GamePad.SetVibration(PlayerIndex.One, 0, 0);
}
```

The statement after the `else` is performed if the condition is found to be `false`. (You can add an `else` part to any `if` condition that you create.) This means that when the red intensity value returns to 0, the vibration stops. You can extend the tests using *OR* so that the program tests all the intensity values:

```
if ( redIntensity > 220 ||
    greenIntensity > 220 ||
    blueIntensity > 220 )
{
    GamePad.SetVibration(PlayerIndex.One, 0, 1);
}
else
{
    GamePad.SetVibration(PlayerIndex.One, 0, 0);
}
```

Now the vibration is controlled by all the intensity values. As an improvement to the game, you might want to experiment with different kinds of vibration for different colors, perhaps by using the low-frequency motor as well. This is controlled by the other value in the call of `SetVibration`:

```
GamePad.SetVibration(PlayerIndex.One, 1, 0);
```

The line of code shown here turns on the low-frequency vibration. You might also want to experiment with the thresholds at which the vibration starts.

The program still has one more problem. If you run it and make the gamepad vibrate, when the program finishes, the gamepad doesn't always stop vibrating. You need to add code that stops the vibration when the game ends. The game stops when the player presses the Back button on the gamepad. The test for this is in the `Update` method. If the Back button is pressed, the `Exit` method is called to stop the game:

```
if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
    this.Exit();
```

The `Exit` method removes the game display and shuts the game down in a tidy fashion. What the program must do is turn off the gamepad motors before `Exit` is called. To do this, the program needs to perform more than one statement if the Back button is pressed, so we need another block:

```
if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
{
    GamePad.SetVibration(PlayerIndex.One, 0, 0);
    this.Exit();
}
```

Now, when the player presses the Back button to end the program, the vibration motors are turned off.

The Great Programmer Speaks: When in Doubt, Make Sure Yourself The Great Programmer says that if you are in a situation where you are not sure whether something is always the case, you should add code to remove all possible doubt. Testing the vibration behavior described in this section, I discovered that the gamepad is left vibrating on earlier versions of XNA, but not on some newer ones. To make absolutely sure that the vibration stops regardless of the version of XNA under which your game runs, you should include the code to stop the vibration yourself.

Sample Code: Vibration Color Nerve Game The sample project in the “03 Color Nerve with Vibes” directory in the source code resources for this chapter holds a version of Color Nerve that has the vibration effect enabled.

Game Idea: Secret Vibration Messages

Once you see that it is easy to read gamepad buttons and drive the motors, you can start to have more fun with XNA, particularly with wireless gamepads. You can create mind-reading games where your assistant seems to know exactly what you are thinking. What the audience doesn't know is that both of you are holding Xbox gamepads in your jacket pockets and using them to send signals back and forth using the vibration feature. The code to do this is actually very simple, and you should be able to understand what it does:

```
protected override void Update(GameTime gameTime)
{
    // Allows the game to exit
    if(GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
    {
        GamePad.SetVibration(PlayerIndex.One, 0, 0);
        GamePad.SetVibration(PlayerIndex.Two, 0, 0);
        this.Exit();
    }

    GamePadState pad1 = GamePad.GetState(PlayerIndex.One);
    GamePadState pad2 = GamePad.GetState(PlayerIndex.Two);

    if (pad1.Buttons.A == ButtonState.Pressed)
    {
        GamePad.SetVibration(PlayerIndex.Two, 0, 1);
    }
    else
    {
        GamePad.SetVibration(PlayerIndex.Two, 0, 0);
    }

    if (pad2.Buttons.A == ButtonState.Pressed)
```

```

    {
        GamePad.SetVibration(PlayerIndex.One, 0, 1);
    }
    else
    {
        GamePad.SetVibration(PlayerIndex.One, 0, 0);
    }

    base.Update(gameTime);
}

```

The Update method reads the A button on the gamepad for Player 1. If this is pressed, it turns on the fast vibration motor in the gamepad for Player 2. It then repeats the process the other way, sending signals from Gamepad 2 to Gamepad 1. This gives you a way in which you can send wireless signals from one gamepad to another. Note that both conditions have else parts so that if the button is not pressed, the vibration is turned off.

You could also use this for practical jokes; for example, just leave a gamepad underneath your victim's bed and then wait until he turns the light off and settles down. Then give the vibration a quick blast for the maximum scare factor. Just don't blame me if you never get the gamepad back!

Sample Code: Vibration Messages The sample project in the "04 Mind Reader" directory in the source code resources for this chapter holds a version of the vibration message program. Just remember to use it wisely. The program also turns the display screen black so that it is not obvious that there is a program running.

Game Idea: Gamepad Racer

The final game idea in this chapter is really silly, but it can be great fun. The first thing you need to do is find a large, smooth table. Put a couple of books under the legs at one end so that the table is sloping, not horizontal. If you put a wireless Xbox gamepad at the top of the table and make the gamepad vibrate, it slides down the table toward the other end. You may need to experiment with the angle, but I've found that with care, you can arrange things so that a gamepad takes around 30 seconds to slide all the way down the table with vibration at full power. If you line up four gamepads on the top of the table, players can pick the one they think will win, and then you can race them down the slope.

The code for this game is very simple indeed; the Update method just turns on all the vibration motors in the gamepads:

```
protected override void Update(GameTime gameTime)
```

```

{
    // Allows the game to exit
    if(GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
    {
        GamePad.SetVibration(PlayerIndex.One, 0, 0);
        GamePad.SetVibration(PlayerIndex.Two, 0, 0);
        GamePad.SetVibration(PlayerIndex.Three, 0, 0);
        GamePad.SetVibration(PlayerIndex.Four, 0, 0);
        this.Exit();
    }

    GamePad.SetVibration(PlayerIndex.One, 1, 1);
    GamePad.SetVibration(PlayerIndex.Two, 1, 1);
    GamePad.SetVibration(PlayerIndex.Three, 1, 1);
    GamePad.SetVibration(PlayerIndex.Four, 1, 1);

    base.Update(gameTime);
}

```

The only complication is that when the game ends, you must turn off all the vibrations. Put all the gamepads at the top of the slope and then run the program. Press the Back button on Gamepad 1 to stop the game.

Sample Code: Gamepad Racer The sample project in the “05 GamepadRacer” directory in the source code resources for this chapter holds a version of the Gamepad Racer program.



Note By carefully tuning vibration values it is possible to “sabotage” gamepads so that the same one wins each time. Note that I do not condone such behavior.

Program Bugs

Your younger brother is still trying to learn to program, but he keeps having problems. He claims that this book is faulty because the programs don’t work properly when he types them in. He is trying to get the Color Nerve game to work, but every time he runs the program, the yellow intensity gets brighter whether he presses the button or not. You take a look at his program and find the following code in the Update method:

```

if (pad1.Buttons.Y == ButtonState.Pressed ||
    keys.IsKeyDown(Keys.Y)) ;
{
    redIntensity++;
    greenIntensity++;
}

```

This is the only part of the program where the yellow intensity is being increased, and it seems that the condition is being ignored.

This looks perfectly okay, and it seems to compile and run correctly, but it seems to be making the yellow intensity brighter every time. At this point, it is a good idea to look at Microsoft Visual Studio and see if the compiler is trying to tell you anything about the code. Figure 3-6 shows your brother's code after he has compiled it.

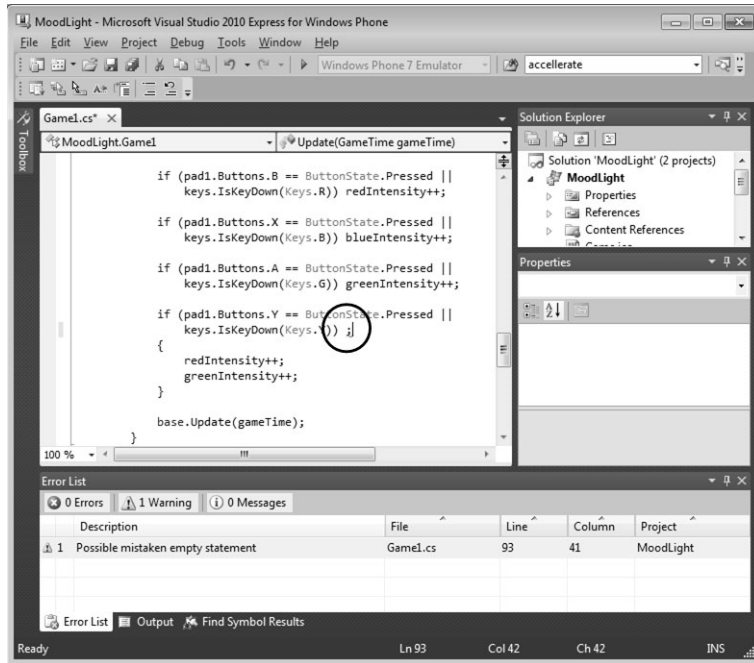


FIGURE 3-6 Visual Studio compiler warning display.

Your attention is drawn to the bottom left corner, where the message “Possible mistaken empty statement” appears. If you double-click this message, you find that the cursor moves to a point just after the `if` condition (I’ve drawn a circle around it in Figure 3-6).

The C# compiler is trying to tell us something about this statement. If we go back to the original listing, we find that your brother has added an extra semicolon at the end of the condition. The problem is that this ends the statement controlled by the condition. So if the R button or the R key is pressed or the Dpad is pressed down, the program decides to do nothing (an empty statement) and then goes on and performs the next statements no matter what, leading to the effect that we are seeing. Figure 3-7 shows how this happens.

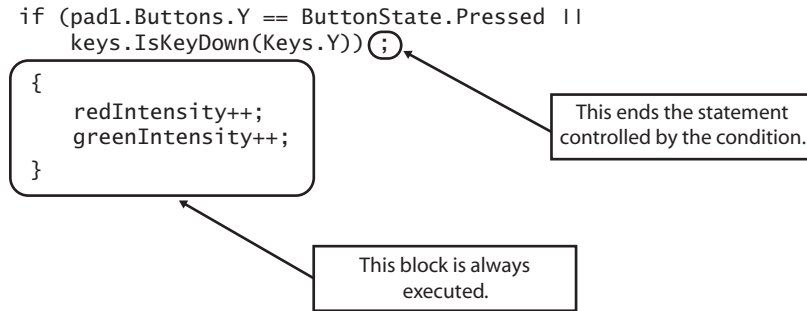


FIGURE 3-7 The effect of an extra semicolon.

You remove the semicolon, the warning goes away, and the program works fine. Your younger brother is now starting to revise his opinion of you and offers to take out the trash that night, even though it is your turn.

The Great Programmer Speaks: Helping Other People Is a Good Plan The Great Programmer has been watching all this with approval. She figures that it is always a good idea to try to help people who are stuck with a problem. Sometimes when a programmer working on uncovering a bug has the chance to explain what is going wrong with a piece of code to an innocent bystander, that can be enough to allow the programmer to work out what is broken. That means you can get a reputation as a fearsome bug fixer just by standing by. Furthermore, seeing what mistakes other people make can give you hints on things that you need to look out for when your programs go wrong. Oh, and sometimes you get your trash taken out for free.

Conclusion

You have learned a lot in this chapter, and you have finally managed to create some games that players can have fun with. You have seen how XNA allows programs to interact with physical devices by calling methods on classes, and we have seen how a program can make decisions on the information that it receives from the devices and use this to make simple (and silly) games.

Chapter Review Questions

No chapter would be complete without a review. So here it is. You should know the routine by now; just decide whether a statement is true or false and look the answers up in Appendix A at the back of the book to find out whether you are a winner or a loser.

1. If a class is an office, a method is a desk.
2. The compiler creates all the instances of classes in a program.
3. An `if` statement must have an `else` part.
4. A parameter is used to feed information into a class.
5. The `else` part of an `if` statement is always performed.
6. The state of a gamepad is represented in an XNA program by a byte value.
7. The `GamePad.GetState` method can be used to see if a button is pressed on a gamepad (this is a tough question; you are allowed to look at the chapter to work it out).
8. A block is a number of C# statements enclosed in curly brackets.
9. The C# condition `(true || false)` means “true or false” and would work out to `true`.
10. The C# condition `(redIntensity > 220)` evaluates to `true` if the value in `greenIntensity` is greater than 220.
11. The gamepad vibration always turns off automatically when an XNA game stops running.

Index

Numbers and Symbols

- 2-D vectors, 92
- 3-D effect for text, 99
 - fake, 103
- && (AND) operator, 178
- == comparison operator, 40
- .cs file extension, 22
- .NET classes
 - Random, 291
- .NET Framework, 95
- operator, 39
- ++ operator, 34
- || (OR) condition, 178
- #region compiler directives, 244

A

- Abs method, 226
- absolute value of a number, 226
- abstract classes, 282
 - designing with, 284
 - extending, 282
 - references to, 284
- acceleration, 336
- accelerometer, 335
 - acceleratorPower variable, 345
 - AccelerometerReadingEventArgs object, 344
 - adding reference to in XNA, 341
 - axes, 337
 - orientation, 338
 - ReadingChanged event, 343
 - reading X, Y, and X values, 342
 - sensitivity, 345
 - shaking, 350
 - starting, 345
 - using values, 345
 - velocity, 346
- Accelerometer class, 340
- AccelerometerReadingEventArgs object, 344
- access modifiers, 217
- Add Existing Item – Content dialog box, 73
- adding content, 72
- adding items to lists, 294
- Add method, 294
- algorithms, 37
- alpha channel, 104

- AND (&&) logical operator, 111
- App Hub community, 6-7, 399
 - membership levels, 7
 - registering, 400
- arg parameter, 202
- arithmetic OR operator, 378
- arrays
 - as lookup tables, 140
 - bounds, 138
 - creating, 136
 - elements, 136
 - index of elements, 137
 - one-dimensional, 136–143
 - reference variable, 149
 - subscript of elements, 137
- artificial intelligence (AI), 297
- aspect ratio, 207
- aspectRatio variable, 208
- assets, 69
- asynchronous processes, 351
- attract mode, 251
- Audacity, 117, 118
- audio. *See* sounds
- automatic sign-in, 316
- avatars, 311

B

- Back button in Windows Phone, 390
 - detecting, 390
- background music, 123, 128
- backgrounds, 272
 - adding, 246
- base class. *See* parent class
- base key word, 276
- BaseSprite class, 272
- batching up sprite-drawing instructions, 81
- battery life in Windows Phone, 376
- behaviors, 27
 - Draw, 27
 - Update, 27
- binding methods, 321
- bitmaps, 68
- bits, 35, 307
- blocks, 23, 52
- bodies of methods, 174
- Boolean algebra, 37

bounds of an array

- bounds of an array, 138
- braces, 23
- Bread and Cheese game, 193
- break key word, 158
- breakpoints, 179, 224
 - hitting, 180
- broadcast addresses, 308
- broadcasting video, 310
- Button-Bashing Mob game, 107
- button positions on gamepads, 110
- bytes, 35

C

- calling methods, 175
- calls, 310
- Cannot implicitly convert type error message, 185
- casts, 185
- CheckCollision method, 258
- Cheese Lander game, 339
- child classes, 270
- classes, 27
 - abstract, 282
 - Accelerometer, 340
 - and references, 267
 - child, 270
 - collection, 293
 - constructors, 286
 - creating, 265
 - extending, 273
 - GamePad, 50
 - GamePadState, 48
 - Gamer, 317
 - Guide, 317
 - hierarchy, 270
 - inheritance, 270
 - instances, 49
 - MediaPlayer, 128
 - members, 29
 - namespaces, 200
 - parent, 270
 - protected members, 274
 - Random, 291
 - SoundEffectInstance
 - SpriteBatch, 81
 - static, 201
 - TouchPanel, 357
 - vector, 348
 - vs. structures, 264
- class instances, 285
- class members
 - static, 292
- clear method, 24
- client-server games
 - PlayingAsHost state, 326
 - PlayingAsPlayer state, 326
 - server behavior, 326
- closing curly bracket (}), 23
- code design, 239
- code regions, 244
- code review, 286
- cohesion, 254
- collection classes, 293
- collisions, handling, 222
- Color Nerve game, 53
- Color Nerve with a Picture game, 85
- colors
 - controlling, 27
 - setting values, 25
 - storing values, 24
 - structures, 24
 - transparency, 104
- comments, 245
- compilation errors, 15
- compiler directives
 - using, 199
- compiler
 - casting, 186
 - conditional compilation, 115
 - directives, 199
 - errors, 15
 - errors vs. warnings, 38
 - integer division vs. floating-point division, 186
 - preprocessor, 115
 - warnings, 38
- components, 281–309
- computer memory, 31
 - memory overflow, 35
- conditional compilation, 115, 199
- conditional statements, 37. *See also* conditions
 - else parts, 39
 - if, 38
- connectivity. *See* networks
- constructors, 286
 - child classes, 288
 - class hierarchies, 287
 - structures, 287
- container objects, 261
- Contains method, 359
- content
 - adding, 72
 - adding to projects, 72
 - sounds, 119
- content folders, 196

content management, 69
 pipeline, 75
 content management solution, 12
 Content Manager, 12, 197
 adding content, 72
 sounds, 119
 XML file format, 91
 control expression, 160
 control keys, 54
 coordinates, 79
 copyright, 118
 coupling objects, 257
 C# programs, 193–214
 Create Directory For Solution check box, 14
 Create Directory For Solution option, 70
 CreateInstance method
 Create method, 321
 cropping pictures, 69

D

datagrams, 310
 data loss, 185
 data types
 double, 191
 float, 191
 date
 getting, 95
 value, 95
 DateTime.Now, 95
 DateTime type, 95
 Now property, 96
 deadly pepper sprite, 275
 debugging, 179, 224
 breakpoints, 179
 declaring a delegate, 322
 declaring a variable, 25
 delegates, 322
 parameters, 322
 type-safe, 322
 development environment. *See* Microsoft
 Visual Studio 2010 Express Edition for
 Windows Phone
 directives, 115
 DirectX version, 7
 displayHeight variable, 206
 displaying keys, 153
 display overscan, televisions, 211
 displayWidth variable, 206
 Dispose method, 323
 DoAdd method, 322
 DoSimpleSum delegate type, 322
 double precision floating-point values, 207

do - while loop construction, 101
 Draw method, 23, 27, 76
 DrawText method, 93, 234
 Dream-Build-Play contest, 8
 Drumpad game, 360
 Drum Pad project, 117

E

edge detectors, 111
 else parts, 39
 emulator, 7
 encapsulation, 257
 enumerated types, 148
 erasing saved game files, 398
 error list, 38
 error messages
 Cannot implicitly convert type, 185
 type or namespace name could not be found,
 200
 EventHandler method, 323
 events, 321
 event generator, 323
 exception handlers, 284
 exceptions, 78, 398
 Existing Item - Content dialog box, 120
 Exit method, 54
 Express edition
 downloading, 6
 Windows Phone, 6
 expressions, 98
 extending classes, 273
 Extensible Markup Language (XML), 90
 Extract Method dialog box, 240

F

fake 3-D text, 103
 fields, 80
 files
 folders, 195
 organizing, 195
 filling the screen with a picture, 83
 finger control. *See* touch input
 floating-point numbers, 183
 floating-point values
 double, 207
 float, 207
 floating-point variables, 191
 float type, 183
 font files, 88
 fonts
 adding as resources, 88

fonts (*continued*)
 drawing with, 92
 files, 88
 Kootenay, 89
 loading, 91
 properties, 94
 scaling, 94
 SprintFont type, 91
 sprite font references, 89
 vectors, 92

forces, 336

foreach loops, 295

for loops, 99, 156

frameworks, 6

friction, 349

fully qualified names, 200

G

Game1 class, renaming, 203

Game1.cs file, 198

Game_Activated method, 388

game consoles, 81

Game_Deactivated method, 388

game lobby, 318

Game Over displays, 384

gameOver method, 251

GamePad class, 50

Gamepad Racer game, 60

gamepads

and keyboards, 55

button positions, 110

ButtonState, 52

getting state, 50

reading, 48–53

SetVibration method, 56

thumbsticks, 218

vibration, 56

vibration frequency, 57

vibration motors, 57

GamePadState, 113

GamePadState class, 48

GamePadState structure, 51

Gamer class, 317

gamer profiles, 314

automatic sign-in, 316

GamerServicesComponent, 313

gamertags, 311

games

adding scores, 233

artificial intelligence (AI), 297

attract mode, 251

backgrounds, 246

Escape key, 54

levels, 236

multi-player, 107–116, 307

networked, 307

progression in, 236

selling, 375

sound, 117

state, 247

survival, 235

title screens, 247

GameSpriteStruct variable, 217

GameState variable, 383

game topology, 325

peer to peer, 325

server-client, 326

game world data, 28

garbage collector, 151, 268

generic methods, 77

generics, 293

GetPressedKeys method, 147

GetState method, 50

Giant Clock project, 87

gozzinta operators, 26

graphical objects. *See* sprites

graphics processor unit (GPU), 81

gravity, 336

greater-than sign (>) operator, 57

Guide class, 317

H

handleIncomingCall method, 387

hardware and driver requirements for XNA, 7

headers for methods, 173

hostSession_GamerJoined method, 323

I

IDE. *See* Integrated Development Environment
 Microsoft Visual Studio 2010 Express Edition
 for Windows Phone, 6

identifiers, 25

changing, 241

if condition, 38

images. *See* pictures

Implement Abstract Class option, 282

indexes, 137

Indie games, 399

inheritance

child, 270

overriding methods from parent, 273

parent, 270

inheritance of behaviors, 270

- Initialize method, 80
- inputs
 - edge-triggered, 146
 - keyboard, 146
 - level-sensitive mode, 146
- instances, 49
 - references to, 267
- integer variables, 108
- Integrated Development Environment, 6
- IntelliSense, 84, 245, 344
 - overriding a method, 276
 - public override, 276
- Intellisense comments, 245
- interfaces, 305
 - C#, 306
 - creating, 306
 - designing with, 308
 - implementing, 307
 - references to, 307
- Internet, 309
 - calls, 310
 - datagrams, 310
 - connections, 310
 - Internet Protocol (IP), 311
 - TCP/IP, 311
 - Transport Control Protocol (TCP), 311
- Internet Protocol (IP), 311
- Internet service provider (ISP), 309
- Intersects method, 222
- int variable, 108
- IsConnected property, 113
- IsDataAvailable property, 328
- IsKeyDown method, 54
- IsLooped property
- isolated storage in Windows Phone, 393

J

- JakeDisplay project, 70
- Joint Photographic Experts Group, 68

K

- keyboards
 - and gamepads, 55
 - control keys, 54
 - displaying keys, 153
 - IsKeyDown method, 54
 - KeyboardState, 54
 - key presses, 146
 - keyState, 147
 - Keys type, 147

- oldKeyState, 147
- rollover, 146
- uppercase keys, 148
- KeyboardState, 54
- keys. *See also* keyboards
 - arrays, 150
 - decoding, 160
 - detecting presses, 155
 - displaying, 153
 - GetPressedKeys method, 148
 - Keys type, 147
 - lower-case, 161
 - registering presses, 146
 - ToLower method, 161
- Keys element in an array, 147
- Keys type, 147
- KeyViewer project, 154
- KillerSprite sprite, 290
- Kootenay font, 89
 - sizing, 95

L

- learning programming, 3
- Length property, 150
- level detectors, 111
- levels, 236
- library projects, 72
- life counters, 235
- light-emitting diode (LED), 307
- linking objects, 257
- linking to resources, 74
- links, 74
- List class, 294
- List collections
 - creating, 293
 - Remove methods, 296
- lists
 - accessing elements, 294
 - Add method, 294
 - foreach loops, 295
 - references, 293
 - Remove methods, 296
 - type, 294
- LoadContent method, 76
- loadGame method, 392
- loading saved games on Windows Phone, 397
- Load method, 77, 91
- lobby display, 324
- local Gamer Profiles, 312
- localization, 96
- local variables, 28
- lock objects, 351

- locks, 351
- logical operators, 56
 - AND (&&), 111
 - greater-than sign (>), 57
- lookup tables using arrays, 140
- loop constructions, 99
 - do - while, 101
 - for, 101
 - while, 101
- loops
 - abandoning, 159
 - break, 158
 - for, 159
 - foreach, 295
- low-level data types, 269

M

- machine instructions, 15
- Main method, 199, 202-203
 - arg parameter, 202
 - parameters, 202
- managing resources, 12
- markup languages, 90
- Math.Abs method, 348
- Math class, 226
- Math.Min method, 349
- measuring forces, 336
- MediaPlayer class, 128
- members of classes, 29
- memory, garbage collector process, 151
- memory overflow, 35
- Message Board project, 145, 158
- method body, 174
- method header, 174
- methods, 23
 - Abs, 226
 - Add, 294
 - base key word, 276
 - body, 174
 - calling, 175
 - CheckCollision, 258
 - Clear, 24
 - collapsing, 244
 - constructors, 286
 - Contains, 359
 - CreateInstance
 - Dispose, 323
 - DoAdd, 322
 - Draw, 27
 - drawText, 234
 - DrawText, 93
 - EventHandler, 323
 - Exit, 54
 - Game_Activated, 388
 - Game_Deactivated, 388
 - gameOver, 251
 - generic, 77
 - GetPressedKeys, 147
 - GetState, 50
 - handleIncomingCall, 387
 - header, 173
 - identifiers, 173
 - inheritance, 273
 - Initialize, 80
 - Intersects, 222
 - IsKeyDown, 54
 - Load, 77
 - LoadContent, 76
 - Math.Abs, 348
 - Math.Min, 349
 - parameters, 50
 - pauseGame, 387
 - placeholders, 282
 - Play, 122, 128
 - refactoring using, 240
 - reference parameters, 221
 - Remove, 161
 - resumeGame, 388
 - return type void, 175
 - Run, 203
 - ScaleSprites, 218
 - SendData, 327
 - SetVibration, 56
 - sharing values between, 28
 - signature, 174
 - startGame, 251
 - static, 202
 - testing, 176
 - ToLongDateString(), 96
 - ToLongTimeString(), 96
 - ToShortDateString(), 96
 - ToShortTimeString(), 96
 - ToString(), 96
 - UnloadContent, 81
 - Update, 27
 - updatePlayingGame, 385
 - value parameters, 221
 - virtual, 273
 - WriteLine, 396
- method signature, 174
- Microsoft Cross-Platform Audio Creation Tool (XACT), 117
- Microsoft.Devices.Sensors library, 341
- Microsoft DreamSpark initiative, 8, 400
- Microsoft Faculty Connection, 8

- Microsoft .NET Framework, 95
- Microsoft Paint, 69
- Microsoft Visual Studio resource library files, 340
- Microsoft Visual Studio 2010 Express Edition for Windows Phone, 6
 - starting, 6
- Mind Reader game, 59
- Mob Reaction Timer game, 131
- modifiers, access, 217
- Mood Light Controller project, 47
- movement. *See* vectors
- moving sprites, 209
- mp3 files, 119
- MSDN Academic Alliance, 8
- multi-player games, 107–116, 307. *See also* networked gaming
- multiple touches, 363
- multitouch, 355
- music. *See* sounds

N

- namespaces, 199
 - fully qualified, 200
- narrowing, 185
- nesting for loops, 156
- nesting loops, 156
- networked game state, 318
- networked gaming, 307
 - client behavior, 329
 - finding games, 325
 - game lobby, 318
 - host roles, 320
 - player roles, 320
 - server behavior, 326
 - signing in players, 320
 - split-screen multi-player mode, 317
 - System Link, 312
 - titleScreen state, 320
 - topology, 325
 - Xbox Live, 311
- networks, 307
 - addresses, 308
 - broadcast addresses, 308
 - destination addresses, 309
 - encoding messages, 309
 - Internet Protocol (IP), 311
 - messages, 309
 - protocols, 308, 310
 - routing, 309
 - streaming media, 310
 - System Link, 311

- TCP/IP, 311
- Transport Control Protocol (TCP), 311
- Xbox Live, 311
- New Project dialog box, 70
- Newton, Isaac, 336
- Newton's Second Law of Motion, 336
- NotSignedIn state, 320
- null references, 127

O

- object hierarchies, 270
- objects, 253–280
 - advantages of classes, 270
 - advantages of structures, 270
 - cohesion, 254
 - container, 261
 - coupling, 257
 - defined, 254
 - encapsulation, 257
 - interaction tables, 260
 - linkage, 257
 - managed by reference, 269
 - managed by value, 269
 - member privacy, 274
 - need, 253
 - object-based design, 254
 - private data, 255
 - protecting data in, 255
 - together, 254
- oldKeyState variable, 147
- open curly bracket character ({), 23
- operands, 34
- operators
 - , 39
 - ++, 34
 - == comparison, 40
 - logical, 56
 - operands, 34
 - OR, 55
- OrientationChanged events, 378
- OR logical operator (||), 55
- overflow, 35

P

- PacketReader class, 326
- PacketWriter class, 326
- packet writers, Write method, 327
- Paint.NET graphics program, 69, 194, 275
- panic button, 356
- Pan property, 125
- parameters, 50

parent classes

- parent classes
 - abstract references to, 285
 - base class, 289
- passing value parameters into method calls, 221
- pauseGame method, 387
- PCs, running XNA games, 7
- peer reviewed games, 399
- peer to peer game topology, 325
- period character (.), 168
- Picture Display game, 67
- picture mood lights, 86
- picture recognition game, 86
- pictures. *See also* sprites
 - adding to projects, 73
 - compression, 69
 - cropping, 69
 - drawing with transparency, 105
 - fade-in effect, 105
 - file types, 68
 - filling the screen, 83
 - formats, 68
 - for Windows Phone display, 38
 - lossless, 68
 - Paint.NET program, 69
 - scaling, 69
 - textures, 75
 - transparency, 69
 - zooming, 167, 188
- pipeline, 75
- Pitch property
- pixels, 79
- PlayingAsHost state, 323, 326
- PlayingAsPlayer state, 326
- Play method, 122, 128
- Portable Network Graphics (PNG), 68
- position on the screen, 79
- preprocessor commands, 115
- private fields in structures, 217
- profiles, 314
- Program.cs source file, 198, 202
- programming languages, 5
- programs
 - blocks, 23, 52
 - comments, 23, 245
 - debugging, 224
 - events, 321
 - methods, 23
 - sensible layout, 33
 - starting, 14
 - statements, 23
 - stopping, 16
 - structures, 24
- projects, 12
 - adding content, 72
 - copying for another device, 17
 - creating, 12, 70
 - empty, 15
 - library, 72
 - multiple, 17
 - organizing files, 195
 - Starter Kits, 12
 - StartUp, 18
 - templates, 12
 - types, 12
- project templates, 12
- properties, 95, 298
 - IsConnected, 113
 - Length, array, 150
 - Now, 96
 - ThumbSticks, 218
- protected class members, 274
- protecting data inside objects, 255
- protocols, 308, 310
 - Internet Protocol (IP), 311
 - internetwork, 311
 - local-level, 311
 - TCP/IP, 311
 - Transport Control Protocol (TCP), 311
- pseudorandom numbers, 291
- public fields in structures, 217
- public keyword, 217
- public override, 276
- publishing games in Windows Phone Marketplace, 8

R

- Random class, 291
- random sequence of numbers, 293
- RayGun effect, 123
- ReadingChanged event, 343
- Rectangle structure, 80
- Rectangle type, 79
- refactoring, 240
 - changing identifiers, 240
 - creating methods from code, 240
- refactoring code, 205
- references, 221, 267–269
 - advantages, 268
 - disadvantage, 269
 - lists, 293
 - multiple to a single class, 267
 - vs. value types, 269
- references to project items, 70
- reference types, 149

- reference variables, 151, 266
- ref modifiers, 221
- regions, 244
- registering a Windows Phone, 11
- Remove method, 161
- renaming items, 205
 - globally, 205
 - refactoring, 205
- renaming the Game1 class, 203
- resistive touch screens, 355
- resources
 - adding to projects, 73
 - fonts, 88
 - linking, 74
 - sprites, 79
- resumeGame method, 388
- return statement, 174
- reusing code with projects, 72
- roles, 324
 - host, 324
 - player, 324
 - SelectingRole state, 324
- routing, 309
- Run method, 203
- running your first program, 14

S

- sample rate of sounds, 118
- ScaleSprites method, 218
- scaling pictures, 69
- scaling text sprites, 94
- scores
 - adding, 233
 - displaying, 234
- screens, getting width, 84
- screen timeout on Windows Phone, 380
- SDK. *See* Software Development Kit; *See also* XNA SDK
 - architecture, 6
 - frameworks, 6
 - XNA, 5
- secure Web sites, 309
- selecting a player role, 324
- SelectingRole state, 320
- selling games, 375
- SendData method, 327
 - Chat parameter, 327
 - InOrder parameter, 327
 - None parameter, 327
 - ReliableInOrder parameter, 328
 - Reliable parameter, 328
- SetVibration method, 56
- shadows, 104
- Shaker game, 335
- ShowSignIn page, 320
- Shuffleboard game, 366
- SignedInGamers property, 317
- SignIn method, 317
- sizing sprites, 208
- Software Development Kits, 5
- Solution Explorer, 71, 341
 - font items in, 89
 - New Folder option, 196
- solutions
 - Solution Explorer, 71
 - vs. projects, 70, 195
- SoundEffectInstance class
- SoundEffect variable, 121
- sounds, 117–130, 302
 - audio channels, 123
 - background music, 123, 128
 - capturing, 118
 - disk space, 118
 - format, 118
 - IsLooped property
 - longer samples, 128
 - MediaPlayer class, 129
 - mp3 files, 119
 - Pan property, 125
 - Pitch property, 125
 - playing, 122
 - quality, 118
 - recommended sample rate and resolution, 118
 - resolution, 118
 - sample rate, 118
 - simultaneous, 123
 - State property, 129
 - storing, 120
 - wav files, 118
 - Windows Media Player, 120
 - Windows Phone channel limit, 304
 - wma files, 119
- source code, 15
- split-screen multi-player mode, 317
- SprinkleFont, 91
- SpriteBatch, 77
- SpriteBatch class, 81
- spriteBatch variable, 81
- sprite class hierarchies, 271
- sprite font references, 89
- sprites, 205
 - aspect ratios, 207
 - deadly pepper, 275
 - display size, 206

- sprites (*continued*)
 - managing size, 206
 - moving, 209
 - overscan, 211
 - positioning, 79
 - random positioning, 290
 - removing, 231
 - sizing, 80, 208
 - structures for information, 216
 - tinting, 85
 - visibility. *See also* visibility
 - Start button, 108
 - Start button in Windows Phone, 390
 - starting new programs, 392
 - Start Debugging button, 14
 - Starter Kits, 12
 - downloading additional, 12
 - startGame method, 251
 - starting projects, 12
 - state, 247
 - diagram, 249
 - known good, 398
 - machines, 249
 - saving, 394
 - variable, 248
 - state diagrams, 382
 - state machines, 381
 - creating, 383
 - Windows Phone, 383
 - statements, 23
 - conditional, 37
 - switch, 160
 - using, 202
 - State property, 129
 - static classes, 201
 - static class members, 292
 - static methods, 202, 226
 - status bar, hiding on Windows Phone, 380
 - streaming media, 310
 - Stream objects, 393
 - streams
 - Close method, 396
 - connecting, 394
 - input/output classes, 395
 - isolated storage, 394
 - rawStream, 396
 - StreamReader, 397
 - StreamWriter, 396
 - text, 394
 - StreamWriter, 396
 - strings, 96
 - struct. *See* structures
 - structures, 24
 - constructors, 287
 - creating, 264
 - fields, 217
 - private fields, 217
 - public fields, 217
 - sprite information, 216
 - vs. classes, 264
 - student developers, 400
 - subscripts, 137
 - Super Zoom Out game, 169
 - SupportedOrientations property, 377
 - survival, 235
 - switch statements, 160
 - System Link, 311, 312
 - local Gamer Profiles, 312
 - System namespace, 199, 226
- ## T
- TargetElapsedTime property, 376
 - test-driven development, 176
 - testing methods, 176
 - text, 87-106
 - 3-D effect, 99
 - case, 161
 - drawing, 92
 - drawing multiple strings, 97
 - editing, 161
 - entry, 145
 - font properties, 94
 - newline character, 161
 - reading input, 145-163
 - scaling, 94
 - shadows, 104
 - strings, 96
 - vectors, 92
 - text streams, 394
 - Texture2D type, 76
 - texture file not found exception, 78
 - textures, 75
 - background, 246
 - reference, 227
 - reusing for multiple objects, 262
 - sizing for Windows Phone, 229
 - this key word, 259
 - throw keyword, 398
 - thumbsticks
 - ThumbSticks property, 218
 - values, 218
 - ThumbSticks property, 218
 - ticks, 210, 376
 - time
 - getting, 95

- time (*continued*)
 - localization, 96
 - value, 95
- timers, 131–143
- timer variable, 131–143
- TimeSpan variables, 376
 - ticks, 376
- title screens, 247
- titleScreen state, 320
- TitleSprite, 272
- together objects, 254
- ToLongDateString() method, 96
- ToLongTimeString() method, 96
- ToLower method, 161
- ToShortDateString() method, 96
- ToShortTimeString() method, 96
- ToString method, 154
- ToString() method, 96
- touch input, 355–373
 - capacitive touch screens, 355
 - Contains method, 359
 - dragging sprites, 367
 - finger-controlled game objects, 366
 - multiple touches, 363
 - multitouch, 355
 - pinching, 355
 - TouchLocation items, 356
 - touch location life cycle, 357
 - TouchLocation values, 359
 - TouchPanel objects, 355
- TouchLocation items, 356
- touch location life cycle, 357
- TouchLocation values, 359
- TouchPanel class, 357
- TouchPanel objects, 355
- ToUpper method, 161
- transparency, 104
 - alpha channel, 104
 - pictures, 105
- Transport Control Protocol (TCP), 311
- type of a variable, 25
- types
 - DateTime, 95
 - enumerated, 148
 - float, 183
 - managed by reference
 - value vs. reference, 149
- type-safe delegates, 322

U

- Undo command, 243
- UnloadContent method, 81
- updateBackground method, 246
- Update method, 27, 76
- updatePlayingGame method, 385
- update rate
 - effect on battery life, 377
 - PC and Xbox vs. Windows Phone, 376
 - TargetElapsedTime property, 376
- using directives, 200
- using statement, 202

V

- value types, 149
- value variables, 151
- variables
 - array, 136
 - aspectRatio, 208
 - bool type, 37
 - GameSpriteStruct, 217
 - identifiers, 25
 - int, 108
 - keyState, 147
 - local, 28
 - oldKeyState, 147
 - reference, 151, 266
 - spriteBatch, 81
 - timer, 131–143
 - type, 25
 - value, 151
- vector classes, 348
 - Length method, 348
 - Vector2 type, 348
- vectors
 - 2-D, 92
 - controlling sounds, 349
 - friction, 349
 - Vector2 type, 348
 - Vector3 data type, 344
- velocity, 346
 - reducing with friction, 349
- vibration, 56
 - low-frequency and high-frequency, 57
- virtual methods, 273
- visibility
 - initial state, 230
 - Visible field, 230
- Visual Studio, 6
 - service packs, 6
 - versions, 6
- Visual Studio 2010 Windows Phone emulator, 10
- Visual Studio solution (.sln) file, 18
- void, 175

W

- WaitingAsPlayer state, 324
- warnings, 39
- wav files, 118
- wavy blue lines in the compiler, 184
- while loop construction, 101
- Windows Bitmap (BMP), 68
- Windows Live ID, 8
- Windows Media Player, 120
- Windows Phone
 - accelerometer, 335
 - accelerometer axes, 338
 - Back and Start buttons, 389
 - battery life, 376
 - capacitive touch screens, 355
 - connecting to Visual Studio 2010, 11
 - detecting phone calls, 388
 - display resolution, 68, 379
 - hiding status bar, 380
 - incoming calls, handling, 386
 - isolated storage, 393
 - Landscape mode, 377
 - loading saved games, 397
 - memory restrictions, 74
 - orientation, 338
 - OrientationChanged event, 378
 - orientations, 377
 - registering, 11
 - registering devices, 400
 - saving game state, 392, 394
 - screen timeout, 380
 - selecting orientations, 377
 - setting up to run XNA games, 10
 - sizing textures for, 229
 - state machine, 381
 - storing games, 17
 - streams, 394
 - supported orientations, 377
 - touch input, 355–373
 - update rate, 376
 - Xbox Live Games menu, 17
- Windows Phone emulator in Visual Studio 2010, 10, 372
- Windows Phone Marketplace, 8, 10, 375, 399
- workspaces, 195
- WriteLine method, 396

X

- XACT audio tool. *See also* Microsoft Cross-Platform Audio Creation Tool

Xbox 360

- connected to TV, 85
- Game Library, 17
- gamer profiles, 313
- gamepad, 7
- Guide, 313
- linking to XNA Game Studio, 8
- networking, 313
- running XNA games, 7
- storing games, 17
- Xbox gamepads
 - wired, 7
 - wireless, 7
- Xbox Live, 311
 - avatars, 311
 - Community Games, 8
 - gamertags, 311
- Xbox Live Gamer Tag, 8
- Xbox Live Indie Games, 8
- Xbox Live subscriptions, 7
- x coordinate values, 79
- XML, 90
- XNA
 - development environment, 6
 - framework, 6
 - hardware and driver requirements, 7
 - IDE, 6
 - installing on PC, 7
- XNA Game Studio
 - connecting to Xbox devices, 8
 - content folders, 196
 - program files, 198
 - Solution Explorer, 22
 - solutions vs. projects, 195
- XNA Game Studio Connect application, 7-8
- XNA Game Studio Device Center, 8
 - adding an Xbox, 9
- XNA Game Studio projects. *See* projects
 - creating, 12
- XNA Indie Games, 399
- XNA workspaces, 17

Y

- y coordinate values, 79

Z

- zooming in on pictures, 167
- zooming pictures, 188
- Zune, 10