

O'REILLY®

Head First

C#

A Learner's Guide to
Real-World Programming
with C# and .NET

Andrew Stellman
& Jennifer Greene

Unity Lab #2

Unity is a great way to practice your C# skills! These Unity Labs are specially designed to help you ramp up on Unity and start writing 3D games in C#.

Check out our GitHub page for videos, downloads, and more!

Fifth Edition
Also covers .NET MAUI & Unity

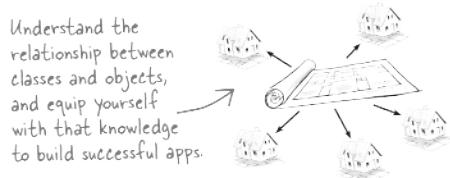


A Brain-Friendly Guide

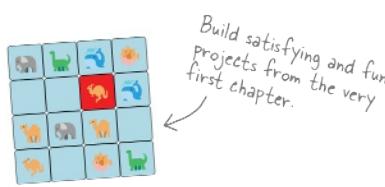
Head First C#

What will you learn from this book?

Create apps, games, and more using this engaging, highly visual introduction to C#, .NET, and software development. You'll learn how to use classes and object-oriented programming, create 3D games in Unity, and query data with LINQ. And you'll do it all by solving puzzles, completing hands-on exercises, and building real-world applications. Interested in a development career? You'll learn important development techniques and ideas—just like many others who've learned to code with this book and are now professional developers, team leads, coding streamers, and more. There's no experience required except the desire to learn. And this is the best place to start.



Understand the relationship between classes and objects, and equip yourself with that knowledge to build successful apps.



Build satisfying projects from the very first chapter.

What's so special about this book?

If you've read a Head First book, you know what to expect: a visually rich format designed for the way your brain works. If you haven't, you're in for a treat. With this book, you'll learn C# through a multisensory experience that engages your mind—rather than a text-heavy approach that puts you to sleep.

C#/ .NET

US \$79.99

CAN \$99.99

ISBN: 978-1-098-14178-3



5 7 9 9 9

"Thank you so much! Your books have helped me to launch my career."

—**Ryan White**
Game Developer

"In a sea of dry technical manuals, *Head First C#* stands out as a beacon of brilliance. Its unique teaching style not only imparts essential knowledge but also sparks curiosity and fuels passion for coding. An indispensable resource for beginners!"

—**Gerald Versluis**
Senior Software Engineer
at Microsoft

"Andrew and Jennifer have written a concise, authoritative, and, most of all, fun introduction to C# development."

—**Jon Galloway**
Senior Program Manager on
the .NET Community Team
at Microsoft

O'REILLY®

Praise for Head First C#

“In a sea of dry technical manuals, *Head First C#* stands out as a beacon of brilliance. Its unique teaching style not only imparts essential knowledge but also sparks curiosity and fuels passion for coding. An indispensable resource for beginners!”

—**Gerald Versluis, Senior Software Engineer at Microsoft**

“*Head First C#* started my career as a software engineer and backend developer. I am now leading a team in a tech company and an open source contributor.”

—**Zakaria Soleymani, Development Team Lead**

“Thank you so much! Your books have helped me to launch my career.”

—**Ryan White, Game Developer**

“If you’re a new C# developer (welcome to the party!), I highly recommend *Head First C#*. Andrew and Jennifer have written a concise, authoritative, and most of all, fun introduction to C# development. I wish I’d had this book when I was first learning C#!”

—**Jon Galloway, Senior Program Manager on the .NET Community Team, Microsoft**

“Not only does *Head First C#* cover all the nuances it took me a long time to understand, it has that Head First magic going on where it is just a super fun read.”

—**Jeff Counts, Senior C# Developer**

“*Head First C#* is a great book with fun examples that keep learning interesting.”

—**Lindsey Bieda, Lead Software Engineer**

“*Head First C#* is a great book, both for brand-new developers and developers like myself coming from a Java background. No assumptions are made as to the reader’s proficiency, yet the material builds up quickly enough for those who are not complete newbies—a hard balance to strike. This book got me up to speed in no time for my first large-scale C# development project at work—I highly recommend it.”

—**Shalewa Odusanya, Principal**

“*Head First C#* is an excellent, simple, and fun way of learning C#. It’s the best piece for C# beginners I’ve ever seen—the samples are clear, the topics are concise and well written. The mini-games that guide you through the different programming challenges will definitely stick the knowledge to your brain. A great learn-by-doing book!”

—**Johnny Halife, Partner**

“*Head First C#* is a comprehensive guide to learning C# that reads like a conversation with a friend. The many coding challenges keep it fun, even when the concepts are tough.”

—**Rebeca Dunn-Krahn, Founding Partner, Sempahore Solutions**

Praise for Head First C#

“I’ve never read a computer book cover to cover, but this one held my interest from the first page to the last. If you want to learn C# in depth and have fun doing it, this is THE book for you.”

—**Andy Parker, fledgling C# Programmer**

“It’s hard to really learn a programming language without good, engaging examples, and this book is full of them! *Head First C#* will guide beginners of all sorts to a long and productive relationship with C# and the .NET Framework.”

—**Chris Burrows, Software Engineer**

“With *Head First C#*, Andrew and Jenny have presented an excellent tutorial on learning C#. It is very approachable while covering a great amount of detail in a unique style. If you’ve been turned off by more conventional books on C#, you’ll love this one.”

—**Jay Hilyard, Director and Software Security Architect, and author of
C# 6.0 Cookbook**

“I’d recommend this book to anyone looking for a great introduction into the world of programming and C#. From the first page onward, the authors walk the reader through some of the more challenging concepts of C# in a simple, easy-to-follow way. At the end of some of the larger projects/labs, the reader can look back at their programs and stand in awe of what they’ve accomplished.”

—**David Sterling, Principal Software Developer**

“*Head First C#* is a highly enjoyable tutorial, full of memorable examples and entertaining exercises. Its lively style is sure to captivate readers—from the humorously annotated examples to the Fireside Chats, where the abstract class and interface butt heads in a heated argument! For anyone new to programming, there’s no better way to dive in.”

—**Joseph Albahari, inventor of LINQPad, and coauthor of C# 12 in a Nutshell and
C# 12 Pocket Reference**

“[*Head First C#*] was an easy book to read and understand. I will recommend this book to any developer wanting to jump into the C# waters. I will recommend it to the advanced developer that wants to understand better what is happening with their code. [I will recommend it to developers who] want to find a better way to explain how C# works to their less-seasoned developer friends.”

—**Giuseppe Turitto, Director of Engineering**

“Andrew and Jenny have crafted another stimulating Head First learning experience. Grab a pencil, a computer, and enjoy the ride as you engage your left brain, right brain, and funny bone.”

—**Bill Mietelski, Advanced Systems Analyst**

“Going through this *Head First C#* book was a great experience. I have not come across a book series which actually teaches you so well....This is a book I would definitely recommend to people wanting to learn C#.”

—**Krishna Pala, MCP**

Praise for the Head First Approach

“I received the book yesterday and started to read it...and I couldn’t stop. This is definitely très ‘cool.’ It is fun, but they cover a lot of ground and they are right to the point. I’m really impressed.”

—**Erich Gamma, IBM Distinguished Engineer, and coauthor of *Design Patterns***

“One of the funniest and smartest books on software design I’ve ever read.”

—**Aaron LaBerge, SVP Technology & Product Development, ESPN**

“What used to be a long trial and error learning process has now been reduced neatly into an engaging paperback.”

—**Mike Davidson, former VP of Design, Twitter, and founder of Newsvine**

“Elegant design is at the core of every chapter here, each concept conveyed with equal doses of pragmatism and wit.”

—**Ken Goldstein, Executive VP & Managing Director, Disney Online**

“Usually when reading through a book or article on design patterns, I’d have to occasionally stick myself in the eye with something just to make sure I was paying attention. Not with this book. Odd as it may sound, this book makes learning about design patterns fun.

“While other books on design patterns are saying ‘Bueller...Bueller...Bueller...’ this book is on the float belting out ‘Shake it up, baby!’”

—**Eric Wuehler**

“I literally love this book. In fact, I kissed this book in front of my wife.”

—**Satish Kumar**

Related books from O'Reilly

We love this book so much!



C# 12 in a Nutshell by Joseph Albahari

Unity Game Development Cookbook, 2nd Edition

by Paris Buttfield-Addison, Jon Manning, and Tim Nugent

Programming C# 12 by Ian Griffiths

Functional Programming with C# by Simon J. Painter

C# 12 Pocket Reference by Joseph Albahari and Ben Albahari

Learning Blazor by David Pine

Other books in O'Reilly's *Head First* series

Head First Android Development

Head First Design Patterns

Head First Git

Head First Python

Head First Go

Head First HTML and CSS

Head First iPhone and iPad Development

Head First Java

Head First JavaScript Programming

Head First Learn to Code

Head First Object-Oriented Analysis and Design

Head First Programming

Head First Python

Head First Software Architecture

Head First SQL

Head First Swift

Head First Web Design

Head First C#



Andrew Stellman
Jennifer Greene

O'REILLY®

Beijing • Boston • Farnham • Sebastopol • Tokyo

Head First C#

by Andrew Stellman and Jennifer Greene

Copyright © 2024 Andrew Stellman and Jennifer Greene. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly Media books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (oreilly.com). For more information, contact our corporate/institutional sales department: (800) 998-9938 or corporate@oreilly.com.

Series Creators:	Kathy Sierra and Bert Bates
Series Advisors:	Eric Freeman and Elisabeth Robson
Acquisitions Editor:	Brian Guerin
Development Editor:	Michele Cronin
Production Editor:	Katherine Tozer
Proofreader:	Piper Editorial Consulting, LLC
Indexer:	Potomac Indexing, LLC
Cover Design:	Susan Thompson, based on a series design by Ellie Volckhausen
Cover and Interior Illustrations:	José Marzan Jr.
Page Viewers:	Samosa the Pomeranian and Rosalie the Australian Labradoodle

Printing History:

November 2007: First Edition

May 2010: Second Edition

August 2013: Third Edition

December 2020: Fourth Edition

July 2024: Fifth Edition



The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. The *Head First* series designations, *Head First C#*, and related trade dress are trademarks of O'Reilly Media, Inc.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and O'Reilly Media, Inc., was aware of a trademark claim, the designations have been printed in caps or initial caps.

While every precaution has been taken in the preparation of this book, the publisher and the authors assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

No pigeons, ducks, ostriches, elephants, dogs, hippos, chameleons, bees, clowns, aliens, game masters, jewel thieves, or superheroes were harmed in the making of this book.

ISBN: 978-1-098-14178-3

[LSI]

[2024-12-19]

Unity Lab #2

Write C# Code for Unity

Unity isn't *just* a powerful, cross-platform engine and editor for building 2D and 3D games and simulations. It's also a **great way to get practice writing C# code.**

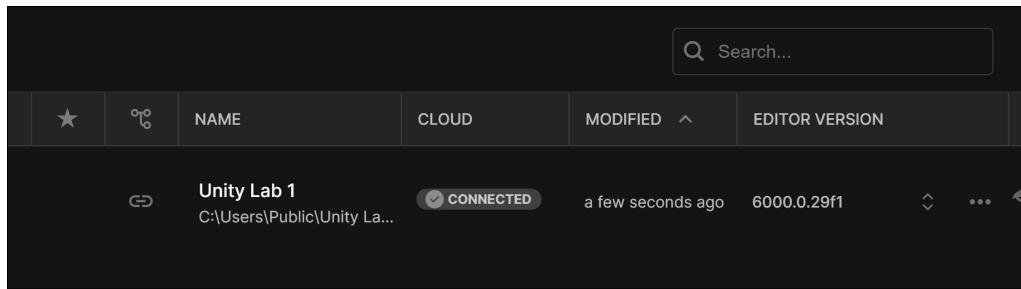
In the last Unity Lab, you learned how to navigate around Unity and your 3D space, and started to create and explore GameObjects. Now it's time to write some code to take control of your GameObjects. The whole goal of that lab was to get you oriented in the Unity editor (and give you an easy way to remind yourself of how to navigate around it if you need it).

In this Unity Lab, you'll start writing code to control your GameObjects. You'll write C# code to explore concepts you'll use in the rest of the Unity Labs, starting with adding a method that rotates the 8 Ball GameObject that you created in the last Unity Lab. You'll also start using the Visual Studio debugger with Unity to sleuth out problems in your games.

C# scripts add behavior to your GameObjects

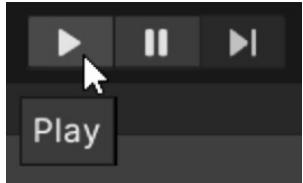
Now that you can add a GameObject to your scene, you need a way to make it, well, do stuff. That's where your C# skills come in. Unity uses **C# scripts** to define the behavior of everything in the game.

This Unity Lab will introduce tools that you'll use to work with C# and Unity. You're going to build a simple "game" that's really just a little bit of visual eye candy: you'll make your 8 ball fly around the scene. Start by going to Unity Hub and **opening the same project** that you created in the first Unity Lab.



Here's what you'll do in this Unity Lab:

- ➊ **Attach a C# script to your GameObject.** You'll add a Script component to your Sphere GameObject. When you add it, Unity will create a class for you. You'll modify that class so that it drives the 8 ball sphere's behavior.
- ➋ **Use Visual Studio to edit the script.** Remember how you set the Unity editor's preferences to make Visual Studio the script editor? That means you can just double-click on the script in the Unity editor and it will open up in Visual Studio.
- ➌ **Play your game in Unity.** There's a Play button at the top of the screen. When you press it, it starts executing all of the scripts attached to the GameObjects in your scene. You'll use that button to run the script that you added to the sphere.



The Play button does not save your game!
So make sure you save early and save often.
A lot of people get in the habit of saving
the scene every time they run the game.

- ➍ **Use Unity and Visual Studio together to debug your script.** You've already seen how valuable the Visual Studio debugger is when you're trying to track down problems in your C# code. Unity and Visual Studio work together seamlessly so you can add breakpoints, use the Locals window, and work with the other familiar tools in the Visual Studio debugger while your game is running.

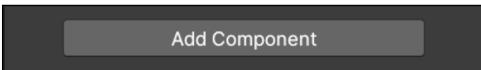
Unity Lab #2

Write C# Code for Unity

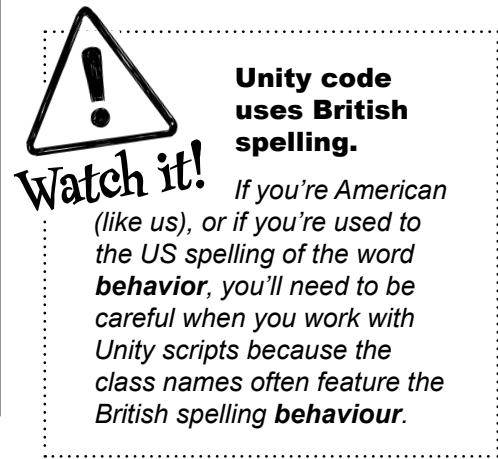
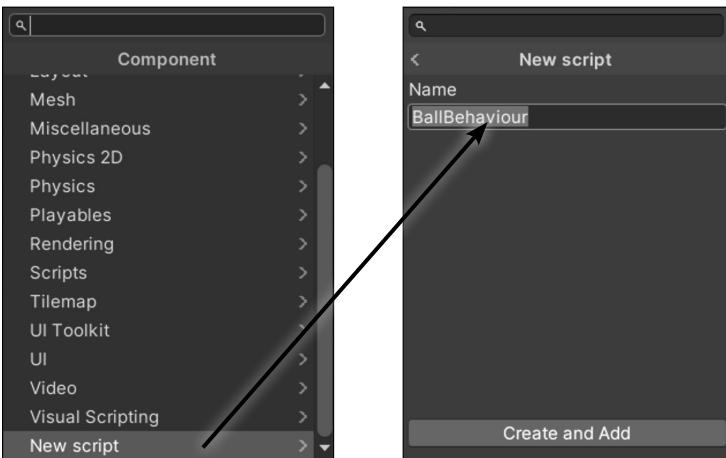
Add a C# script to your GameObject

Unity is more than an amazing platform for building 2D and 3D games. Many people use it for artistic work, data visualization, augmented reality, and more. It's especially valuable to you, as a C# learner, because you can write code to control everything that you see in a Unity game. That makes Unity **a great tool for learning and exploring C#**.

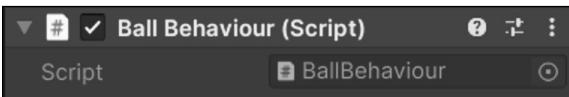
Let's start using C# and Unity right now. Make sure the Sphere GameObject is selected, then **click the Add Component button** at the bottom of the Inspector window.



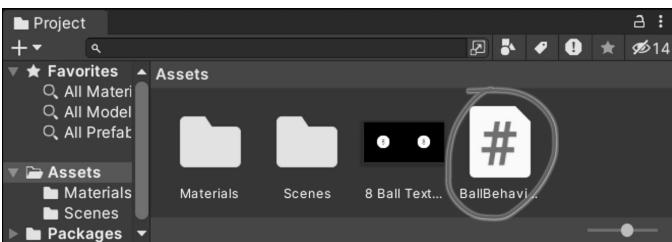
When you click it, Unity pops up a window with all of the different kinds of components that you can add—and there are **a lot** of them. **Choose “New script”** to add a new C# script to your Sphere GameObject. You'll be prompted for a name. **Name your script BallBehaviour.**



Click the “Create and Add” button to add the script. You'll see a component called *Ball Behaviour (Script)* appear in the Inspector window.



You'll also see the C# script in the Project window.



The *Project* window gives you a folder-based view of your project. Your Unity project is made up of files: media files, data files, C# scripts, textures, and more. Unity calls these files **assets**. The *Project* window was displaying a folder called **Assets** when you right-clicked inside it to import your texture, so Unity added it to that folder.

Did you notice a folder called **Materials** appeared in the *Project* window as soon as you dragged the 8 ball texture onto your sphere?

Write C# code to rotate your sphere

In the first lab, you told Unity to use Visual Studio as its external script editor. So go ahead and **double-click your new C# script in the Assets window**. When you do, **Unity will open your script in Visual Studio**. Your C# script contains a class called BallBehaviour with two empty methods called Start and Update:

```
using UnityEngine;

public class BallBehaviour : MonoBehaviour
{
    // Start is called before the first frame update
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {
    }
}
```

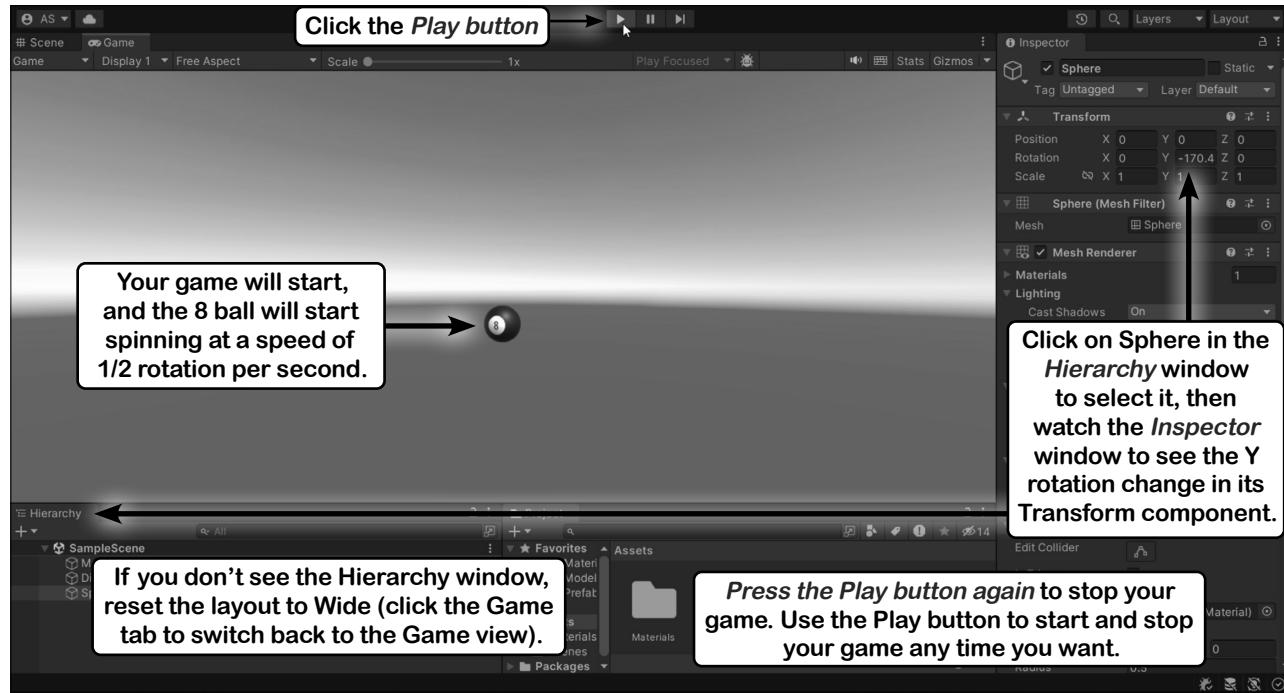
You opened your C# script in Visual Studio by clicking on it in the Hierarchy window, which shows you a list of every GameObject in the current scene. When Unity created your project, it added a scene called SampleScene with a camera and a light. You added a sphere to it, so your Hierarchy window will show all of those things.

If Unity didn't launch Visual Studio and open your C# script in it, go back to the beginning of Unity Lab 1 and make sure you followed the steps to set the External Tools preferences.

Here's a line of code that will rotate your sphere. **Add it to your Update method:**

```
transform.Rotate(Vector3.up, 180 * Time.deltaTime);
```

Now **go back to the Unity editor** and click the Play button in the toolbar to start your game:



Your Unity Code Up Close



```
using UnityEngine; ←  
  
public class BallBehaviour : MonoBehaviour  
{  
    // Start is called before the first frame update  
    void Start()  
    {  
    }  
  
    // Update is called once per frame  
    void Update()  
    {  
        transform.Rotate(Vector3.up, 180 * Time.deltaTime);  
    }  
}
```

The `transform.Rotate` method causes a GameObject to rotate. The first parameter is the axis to rotate around. In this case, your code used `Vector3.up`, which tells it to rotate around the Y axis. The second parameter is the number of degrees to rotate.

A *frame* is a fundamental concept of animation. Unity draws one still frame, then draws the next one very quickly, and your eye interprets changes in these frames as movement. Unity calls the `Update` method for every `GameObject` before each frame so it can move, rotate, or make any other changes that it needs to make. A faster computer will run at a higher *frame rate*—or number of frames per second (FPS)—than a slower one.

Inside your `Update` method, multiplying any value by `Time.deltaTime` turns it into that value per second.

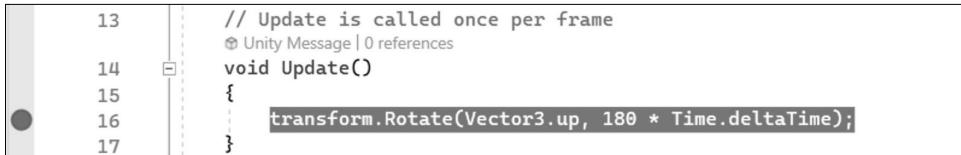
Different computers will run your game at different frame rates. If it's running at 30 FPS, we want one rotation every 60 frames. If it's running at 120 FPS, it should rotate once every 240 frames. Your game's frame rate may even change if it needs to run more or less complex code.

That's where the `Time.deltaTime` value comes in handy. Every time the Unity engine calls a `GameObject`'s `Update` method—once per frame—it sets `Time.deltaTime` to the fraction of a second since the last frame. Since we want our ball to do a full rotation every two seconds, or 180 degrees per second, all we need to do is multiply it by `Time.deltaTime` to make sure that it rotates exactly as much as it needs to for that frame.

`Time.deltaTime` is static—and like we saw in Chapter 3, you don't need an instance of the `Time` class to use it.

Add a breakpoint and debug your game

Let's debug your Unity game. First **stop your game** if it's still running (by pressing the Play button again). Then switch over to Visual Studio, and **add a breakpoint** on the line that you added to the Update method:



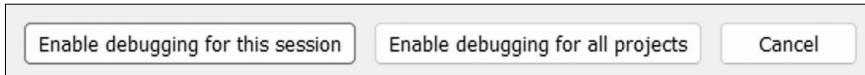
```

13     // Update is called once per frame
14     void Update()
15     {
16         transform.Rotate(Vector3.up, 180 * Time.deltaTime);
17     }

```

- ★ In Visual Studio, click the  Attach to Unity button or choose Debug >> Start Debugging (F5) from the menu.
- ★ In VSCode, choose  Run and Debug (Ctrl+Shift+D) on the left, then click  Attach to Unity

Attaching to Unity **starts the debugger**. Now switch back to the Unity editor. If this is the first time you're debugging this project, the Unity editor will pop up a dialog window with these buttons:



Press the “Enable debugging for this session” button (or if you want to keep that pop-up from appearing again, press “Enable debugging for all projects”). Visual Studio is now **attached** to Unity, which means it can debug your game.

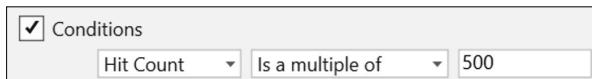
Now **press the Play button in Unity** to start your game. Since Visual Studio is attached to Unity, it **breaks immediately** on the breakpoint that you added, just like with any other breakpoint you've set.

Use a hit count to skip frames

 Congratulations, you're now debugging a game!

Sometimes it's useful to let your game run for a while before your breakpoint stops it. For example, you might want your game to spawn and move its enemies before your breakpoint hits. Let's tell your breakpoint to break every 500 frames. You can do that by adding a **Hit Count condition** to your breakpoint:

- ★ In Visual Studio, right-click on the breakpoint dot (●) at the left side of the line, choose **Conditions** from the pop-up menu, select **Hit Count** and **Is a multiple of** from the dropdowns, and enter **500** in the box:



- ★ In VSCode, right-click the breakpoint dot, choose **Edit breakpoint** from the dropdown, and enter **500**:



Now the breakpoint will only pause the game every 500 times the Update method is run—or every 500 frames. If your game is running at 60 FPS, then when you press Continue the game will run for a little over 8 seconds before it breaks again. **Press Continue, then switch back to Unity** and watch the ball spin until the breakpoint breaks.



VSCode lets you set a hit count for breakpoints just like Visual Studio does, but at the time we're writing this that feature does not work when debugging Unity projects.

Unity Lab #2

Write C# Code for Unity

Use the debugger to understand Time.deltaTime

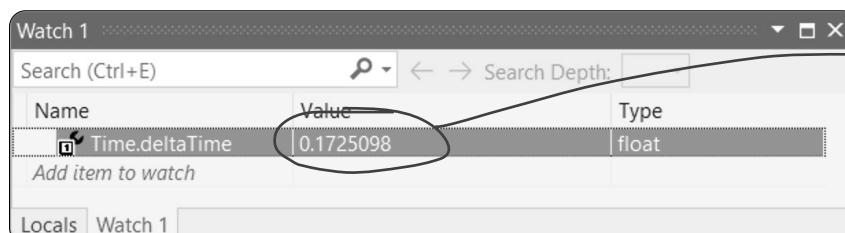
You're going to be using `Time.deltaTime` in many of the Unity Labs projects. Let's take advantage of your breakpoint and use the debugger to really understand what's going on with this value.

While your game is paused on the breakpoint in Visual Studio, **hover over `Time.deltaTime`** to see the fraction of a second that elapsed since the previous frame (you'll need to put your mouse cursor over `deltaTime`). Then **add a watch for `Time.deltaTime`** by selecting `Time.deltaTime` and choosing Add Watch from the right-mouse menu.

```
12
13     // Update is called once per frame
14     void Update()
15     {
16         transform.Rotate(Vector3.up, 180 * Time.deltaTime);
17     }
18 }
```

Every time the breakpoint pauses the game, your `Time.deltaTime` watch will show you the fraction of a second since the previous frame. Can you use this number to figure out the FPS we were getting when we took this screenshot?

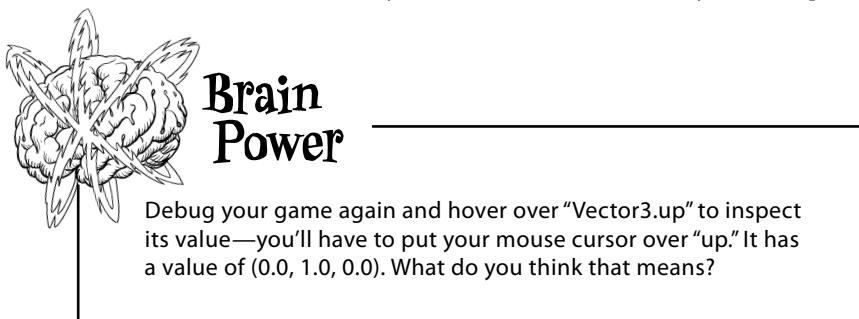
Continue debugging (F5), just like with the other apps you've debugged, to resume your game. The ball will start rotating again, and after another 500 frames the breakpoint will trigger again. You can keep running the game for 500 frames at a time. Keep your eye on the `Time.deltaTime` value each time it breaks, either in the pinned value or in the watch window.



Stop debugging (Shift + F5) to stop your program. Then **start debugging again**. Since your game is still running, the breakpoint will continue to work when you reattach Visual Studio to Unity. Once you're done debugging, **toggle your breakpoint again** so the IDE will still keep track of it but not break when it's hit. **Stop debugging** one more time to detach from Unity.

Go back to Unity and **stop your game**—and **save it**, because the Play button doesn't automatically save the game.

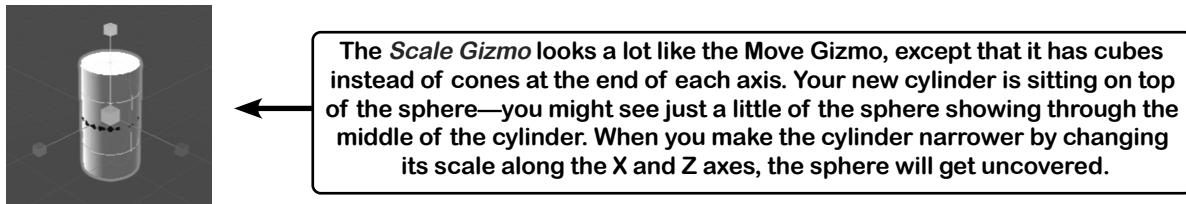
The Play button in Unity starts and stops your game. Visual Studio will stay attached to Unity even when the game is stopped.



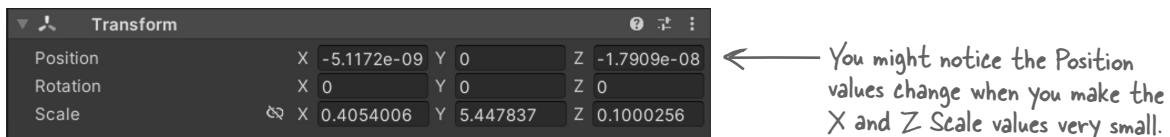
Add a cylinder to show where the Y axis is

Your sphere is rotating around the Y axis at the very center of the scene. Let's add a very tall and very skinny cylinder to make it visible. **Create a new cylinder** by choosing 3D Object >> Cylinder from the GameObject menu. Make sure it's selected in the Hierarchy window, then look at the Inspector window and check that Unity created it at position (0, 0, 0)—if not, use the context menu (⋮) to reset it.

Let's make the cylinder tall and skinny. Choose the Scale tool from the toolbar: either click on it (⧉) or press the R key. You should see the Scale Gizmo appear on your cylinder:

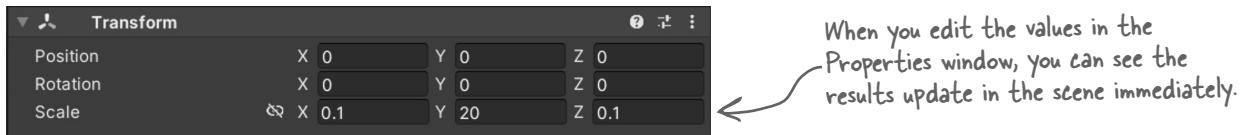


Click and drag the green cube up to elongate your cylinder along the Y axis. Then click on the red cube and drag it toward the cylinder to make it very narrow along the X axis, and do the same with the blue cube to make it very narrow along the Z axis. Watch the Transform panel in the Inspector as you change the cylinder's scale—the Y scale will get larger, and the X and Z values will get much smaller.



Click on the X label in the Scale row in the Transform panel and drag up and down. Make sure you click the actual X label to the left of the input box with the number. When you click the label it turns blue, and a blue box appears around the X value. As you drag your mouse up and down, the number in the box goes up and down, and the Scene view updates the scale as you change it. Look closely as you drag—the scale can be positive and negative. Now **reset the Transform window**.

Now **select the number inside the X box and type 0.1**—the cylinder gets very skinny. Press Tab and type **20**, then press Tab again and type **0.1**, and press Enter.



Now your sphere has a very long cylinder going through it that shows the Y axis where Y = 0.



Unity Lab #2

Write C# Code for Unity

Add fields to your class for the rotation angle and speed

In Chapter 3, you learned how C# classes can have **fields** that store values methods can use. Let's modify your code to use fields. Add these four lines just under the class declaration, **immediately after the first curly brace { :**

```
public class BallBehaviour : MonoBehaviour
{
    public float XRotation = 0;
    public float YRotation = 1;
    public float ZRotation = 0;
    public float DegreesPerSecond = 180;
```

These are just like the fields that you added to the projects in Chapters 3 and 4. They're variables that keep track of their values—each time Update is called, it reuses the same field over and over again.

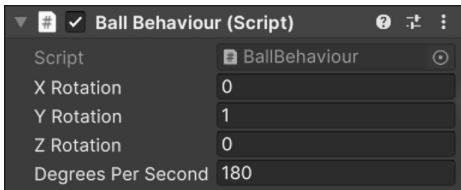
The XRotation, YRotation, and ZRotation fields each contain a value between 0 and 1, which you'll combine to create a **vector** that determines the direction that the ball will rotate:

```
new Vector3(XRotation, YRotation, ZRotation)
```

The DegreesPerSecond field contains the number of degrees to rotate per second, which you'll multiply by Time.deltaTime just like before. **Modify your Update method to use the fields.** This new code creates a Vector3 variable called **axis** and passes it to the transform.Rotate method:

```
void Update()
{
    Vector3 axis = new Vector3(XRotation, YRotation, ZRotation);
    transform.Rotate(axis, DegreesPerSecond * Time.deltaTime);
}
```

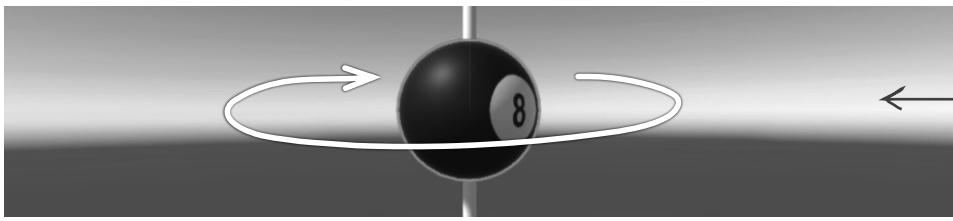
Select the Sphere in the Hierarchy window. Your fields now show up in the Script component. When the Script component renders fields, it adds spaces between the capital letters to make them easier to read.



When you add public fields to a class in your Unity script, the Script component displays input boxes that let you modify those fields. If you modify them while the game is *not* running, the updated values will get saved with your scene. You can also modify them while the game is running, but they'll revert when you stop the game.

Run your game again. **While it's running**, select the Sphere in the Hierarchy window and change the degrees per second to 360 or 90—the ball starts to spin at twice or half the speed. Stop your game—and the field will reset to 180.

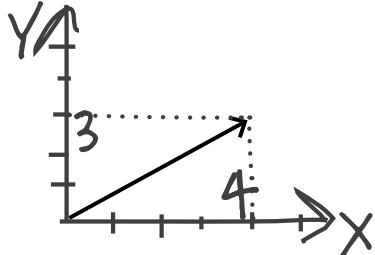
While the game is stopped, use the Unity editor to change the X Rotation field to 1 and the Y Rotation field to 0. Start your game—the ball will rotate away from you. Click the X Rotation label and drag it up and down to change the value while the game is running. As soon as the number turns negative, the ball starts rotating toward you. Make it positive again, and it starts rotating away from you.



When you use the Unity editor to set the Y Rotation field to 1 and then start your game, the ball rotates clockwise around the Y axis.

Use Debug.DrawRay to explore how 3D vectors work

A **vector** is a value with a **length** (or magnitude) and a **direction**. If you ever learned about vectors in a math class, you probably saw lots of diagrams like this one of a 2D vector:



Here's a diagram of a two-dimensional vector. You can represent it with two numbers: its value on the X axis (4) and its value on the Y axis (3), which you'd typically write as (4, 3).

That's not hard to understand...on an intellectual level. But even those of us who took a math class that covered vectors don't always have an **intuitive** grasp of how vectors work, especially in 3D. Here's another area where we can use C# and Unity as a tool for learning and exploration.

Use Unity to visualize vectors in 3D

You're going to add code to your game to help you really “get” how 3D vectors work. Start by having a closer look at the first line of your Update method:

```
Vector3 axis = new Vector3(XRotation, YRotation, ZRotation);
```

What does this line tell us about the vector?

- ★ **It has a type: Vector3.** Every variable declaration starts with a type. Instead of using string, int, or bool, you're declaring it with the type Vector3. This is a type that Unity uses for 3D vectors.
- ★ **It has a variable name: axis.**
- ★ **It uses the new keyword to create a Vector3.** It uses the XRotation, YRotation, and ZRotation fields to create a vector with those values.

So what does that 3D vector look like? There's no need to guess—we can use one of Unity's useful debugging tools to draw the vector for us. **Add this line of code to the end of your Update method:**

```
void Update()
{
    Vector3 axis = new Vector3(XRotation, YRotation, ZRotation);
    transform.Rotate(axis, DegreesPerSecond * Time.deltaTime);
    Debug.DrawRay(Vector3.zero, axis, Color.yellow);
}
```

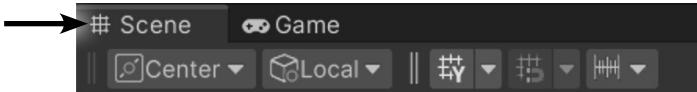
The Debug.DrawRay method is a special method that Unity provides to help you debug your games. It draws a **ray**—which is a vector that goes from one point to another—and takes parameters for its start point, end point, and color. There's one catch: **the ray only appears in the Scene view**. The methods in Unity's Debug class are designed so that they don't interfere with your game. They typically only affect how your game interacts with the Unity editor.

Unity Lab #2

Write C# Code for Unity

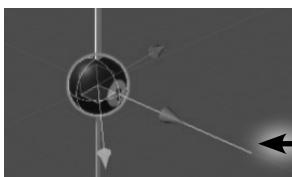
Run the game to see the ray in the Scene view

Now run your game again. You won't see anything different in the Game view because `Debug.DrawRay` is a tool for debugging that doesn't affect gameplay at all. Use the Scene tab to **switch to the Scene view**. You may also need to **reset the Wide layout** by choosing Wide from the Layout dropdown.



Now you're back in the familiar Scene view. Do these things to get a real sense of how 3D vectors work:

- ★ Use the Inspector to **modify the BallBehaviour script's fields**. Set the X Rotation to 0, Y Rotation to 0, and **Z Rotation to 3**. You should now see a yellow ray coming directly out of the Z axis and the ball rotating around it (remember, the ray only shows up in the Scene view).



The vector (0, 0, 3) extends 3 units along the Z axis. Look closely at the grid in the Unity editor—the vector is exactly 3 units long. Try clicking and dragging the Z Rotation label in the Script component in the Inspector. The ray will get larger or smaller as you drag. When the Z value in the vector is negative, the ball rotates in the other direction.

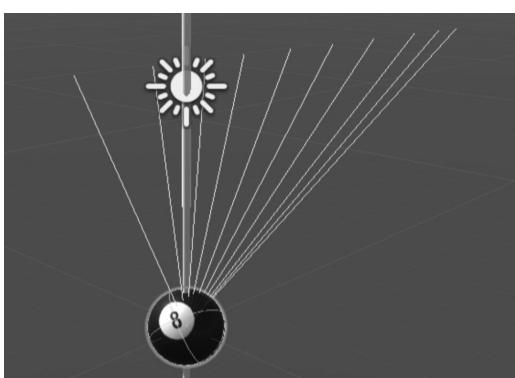
- ★ Set the Z Rotation back to 3. Experiment with dragging the X Rotation and Y Rotation values to see what they do to the ray. Make sure to reset the Transform component each time you change them.
- ★ Use the Hand tool and the Scene Gizmo to get a better view. Click the X cone on the Scene Gizmo to set it to the view from the right. Keep clicking the cones on the Scene Gizmo until you see the view from the front. It's easy to get lost—you can **reset the Wide layout to get back to a familiar view**.

Add a duration to the ray so it leaves a trail

You can add a fourth argument to your `Debug.DrawRay` method call that specifies the number of seconds the ray should stay on the screen. Add `.5f` to make each ray stay on screen for half a second:

```
Debug.DrawRay(Vector3.zero, axis, Color.yellow, .5f);
```

Now run the game again and switch to the Scene view. Now when you drag the numbers up and down, you'll see a trail of rays left behind. This looks really interesting, but more importantly, it's a great tool to visualize 3D vectors.

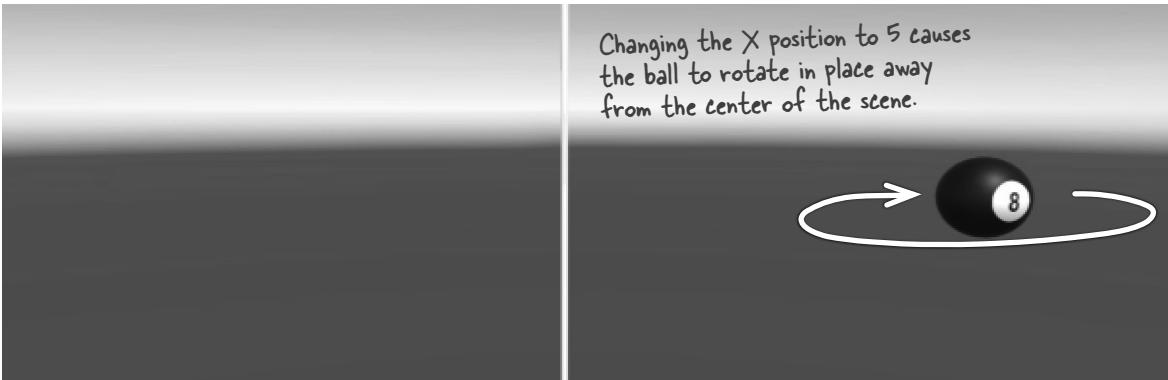


Making your ray leave a trail is a good way to help you develop an intuitive sense of how 3D vectors work.

You can use the Inspector window to modify the fields in a Script component while the game is running. The field values will reset when you stop the game. It will remember the values if you set them while the game is stopped.

Rotate your ball around a point in the scene

Your code calls the `transform.Rotate` method to rotate your ball around its center, which changes its X, Y, and Z rotation values. **Select Sphere in the Hierarchy window and change its X position to 5** in the Transform component. Then **use the context menu (⋮) in the BallBehaviour Script component** to reset its fields. Run the game again—now the ball will be at position (5, 0, 0) and rotating around its own Y axis.



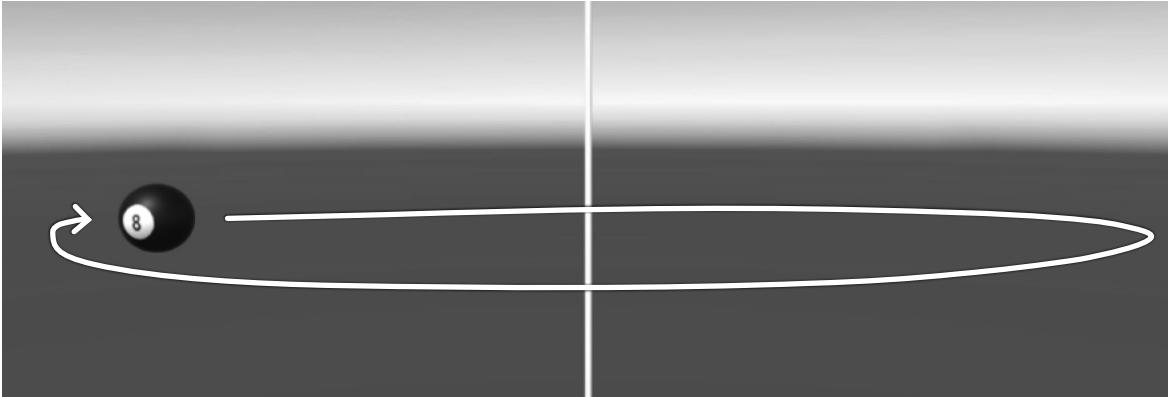
Let's modify the `Update` method to use a different kind of rotation. Now we'll make the ball rotate around the center point of the scene, coordinate (0, 0, 0), using the **`transform.RotateAround` method**, which rotates a `GameObject` around a point in the scene. (This is *different* from the `transform.Rotate` method you used earlier, which rotates a `GameObject` around its center.) Its first parameter is the point to rotate around. We'll use **`Vector3.zero`** for that parameter, which is a shortcut for writing `new Vector3(0, 0, 0)`.

Here's the new `Update` method:

```
void Update()
{
    Vector3 axis = new Vector3(XRotation, YRotation, ZRotation);
    transform.RotateAround(Vector3.zero, axis, DegreesPerSecond * Time.deltaTime);
    Debug.DrawRay(Vector3.zero, axis, Color.yellow, .5f);
}
```

This new `Update` method rotates the ball around the point (0, 0, 0) in the scene.

Now run your code. This time it rotates the ball in a big circle around the center point:

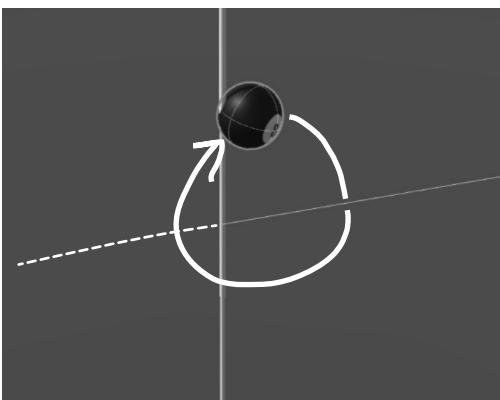


Use Unity to take a closer look at rotation and vectors

You're going to be working with 3D objects and scenes in the rest of the Unity Labs throughout the book. Even those of us who spend a lot of time playing 3D video games don't have a perfect feel for how vectors and 3D objects work, and how to move and rotate in a 3D space. Luckily, Unity is a great tool to **explore how 3D objects work**. Let's start experimenting right now.

While your code is running, try changing parameters to experiment with the rotation:

- ★ **Switch back to the Scene view** so you can see the yellow ray that Debug.DrawRay renders in your BallBehaviour.Update method.
- ★ Use the Hierarchy window to **select the Sphere**. You should see its components in the Inspector window.
- ★ Change the **X Rotation, Y Rotation, and Z Rotation values** in the Script component to **10** so you see the vector rendered as a long ray. Use the Hand tool (Q) to rotate the Scene view until you can clearly see the ray.
- ★ Use the Transform component's context menu (⋮) to **reset the Transform component**. Since the center of the sphere is now at the zero point in the scene, (0, 0, 0), it will rotate around its own center.
- ★ Then **change the X position** in the Transform component to **2**. The ball should now be rotating around the vector. You'll see the ball cast a shadow on the Y axis cylinder as it flies by.



While the game is running, set the X, Y, and Z Rotation fields in the BallBehaviour Script component to 10, reset the sphere's Transform component, and change its X position to 2—as soon as you do, it starts rotating around the ray.

Try **repeating the last three steps** for different values of X, Y, and Z rotation, resetting the Transform component each time so you start from a fixed point. Then try clicking the rotation field labels and dragging them up and down—see if you can get a feel for how the rotation works.

Unity is a
great tool to
explore how 3D
objects work
by modifying
properties on
your GameObjects
in real time.

Get creative!

This is your chance to **experiment on your own with C# and Unity**. You've seen the basics of how you combine C# and Unity GameObjects. Take some time and play around with the different Unity tools and methods that you've learned about in the first two Unity Labs. Here are some ideas:

- ★ Add cubes, cylinders, or capsules to your scene. Attach new scripts to them—make sure you give each script a unique name!—and make them rotate in different ways.
- ★ Try putting your rotating GameObjects in different positions around the scene. See if you can make interesting visual patterns out of multiple rotating GameObjects.
- ★ Try adding a light to the scene. What happens when you use `transform.rotateAround` to rotate the new light around various axes?
- ★ Here's a quick coding challenge: try using `+=` to add a value to one of the fields in your BallBehaviour script. Make sure you multiply that value by `Time.deltaTime`. Try adding an `if` statement that resets the field to 0 if it gets too large.

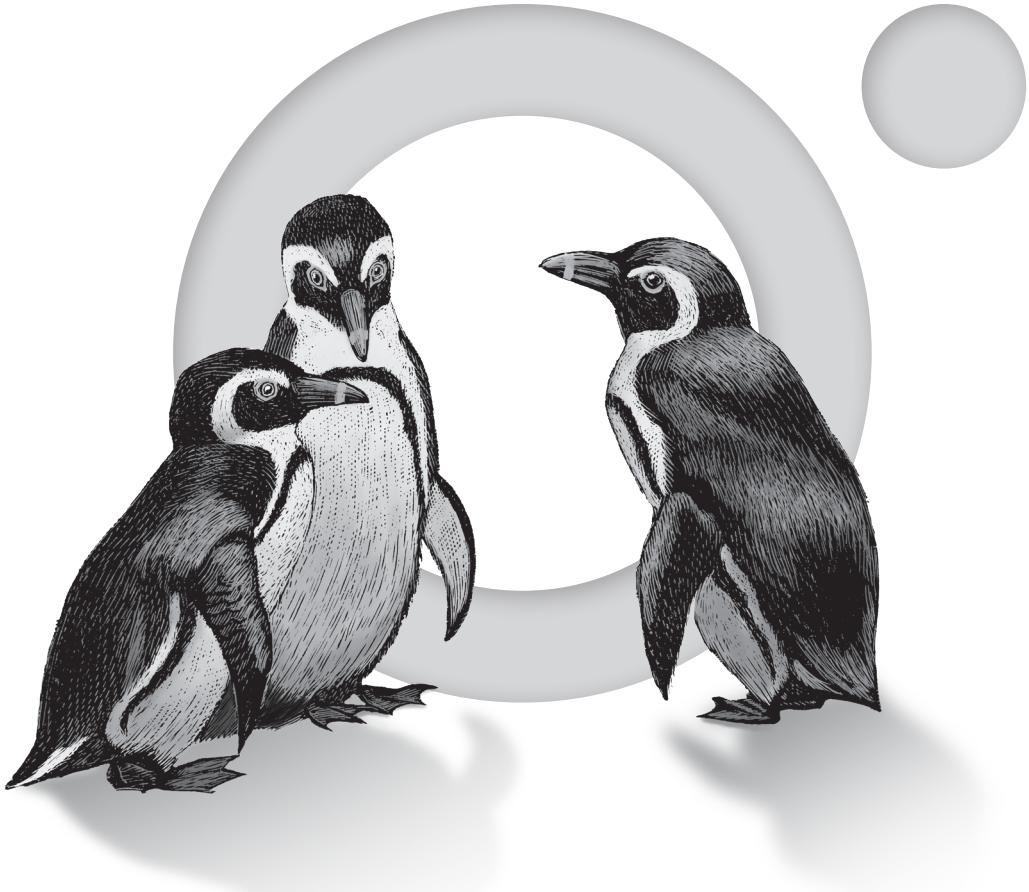


Before you run the code, try to figure out what it will do. Does it act the way you expected it to act? Trying to predict how the code you added will act is a great technique for getting better at C#.

Take the time to experiment with the tools and techniques you just learned. This is a great way to take advantage of Unity and Visual Studio as tools for exploration and learning.

Bullet Points

- The **Scene Gizmo** always displays the camera's orientation.
- You can **attach a C# script** to any GameObject. The script's `Update` method will be called once per frame.
- The **transform.Rotate** method causes a GameObject to rotate a number of degrees around an axis.
- Inside your `Update` method, multiplying any value by `Time.deltaTime` turns it into that value per second.
- You can **attach** the Visual Studio debugger to Unity to debug your game while it's running. It will stay attached to Unity even when your game is not running.
- Adding a **Hit Count condition** to a breakpoint makes it break after the statement has executed a certain number of times.
- A **field** is a variable that lives inside of a class outside of its methods, and it retains its value between method calls.
- Adding public fields to the class in your Unity script makes the Script component show **input boxes that let you modify those fields**. It adds spaces between capital letters in the field names to make them easier to read.
- You can create 3D vectors using **new Vector3**. (You learned about the `new` keyword in Chapter 3.)
- The **Debug.DrawRay** method draws a vector in the Scene view (but not the Game view). You can use vectors as a debugging tool, but also as a learning tool.
- The **transform.RotateAround** method rotates a GameObject around a point in the scene.



O'REILLY®

Learn from experts.
Become one yourself.

60,000+ titles | Live events with experts | Role-based courses
Interactive learning | Certification preparation



Try the O'Reilly learning platform
free for 10 days.