O'REILLY®

Fourth Edition

Head First C#

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

Andrew Stellman
& Jennifer Greene

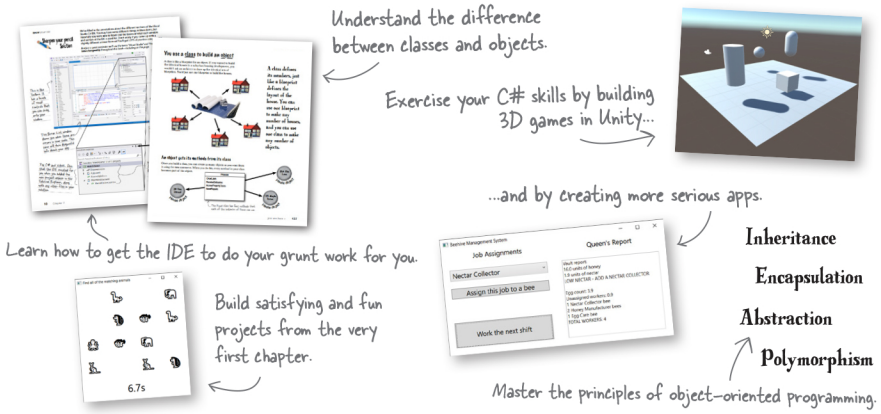
A Brain-Friendly Guide

Head First

C#

What will you learn from this book?

Dive into C# and create apps, user interfaces, games, and more using this fun and highly visual introduction to C#, .NET Core, and Visual Studio. With this completely updated guide, which covers C# 8.0 and Visual Studio 2019, beginning programmers like you will build a fully functional game in the opening chapter. Then 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, doing hands-on exercises, and building real-world applications. By the time you're done, you'll be a solid C# programmer—and you'll have a great time along the way!



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

—Ryan White
Game Developer

"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

"If you want to learn C# in depth and have fun doing it, this is THE book for you."

—Andy Parker
Fledgling C# programmer

What's so special about this book?

Based on the latest research in cognitive science and learning theory, *Head First C#* uses a visually rich format to engage your mind rather than a text-heavy approach that puts you to sleep. Why waste your time struggling with new concepts? This multisensory learning experience is designed for the way your brain really works.

.NET

US $64.99 **CAN $85.99**

ISBN: 978-1-491-97670-8

9 781491 976708 56499

O'REILLY®

Head First C#

Fourth Edition

Andrew Stellman
Jennifer Greene

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Head First C#

Fourth Edition

by Andrew Stellman and Jennifer Greene

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 (*http://oreilly.com*). For more information, contact our corporate/institutional sales department: (800) 998-9938 or *corporate@oreilly.com*.

Series Creators:	Kathy Sierra, Bert Bates
Cover Designer:	Ellie Volckhausen
Brain Image on Spine:	Eric Freeman
Editors:	Nicole Taché, Amanda Quinn
Proofreader:	Rachel Head
Indexer:	Potomac Indexing, LLC
Illustrator:	Jose Marzan
Page Viewers:	Greta the miniature bull terrier and Samosa the Pomeranian

Printing History:

November 2007: First Edition.
May 2010: Second Edition.
August 2013: Third Edition.
December 2020: Fourth Edition

No bees, space aliens, or comic book heroes were harmed in the making of this book.

ISBN: 978-1-491-97670-8

[LSI] [2020-11-13]

Unity Lab

Collision Detection

In the last Unity Lab, you started building a game where the player scores points by moving around the floor and running into targets. You used billiard balls as targets, and wrote code that used Unity's physics engine to move them around the play area. Now it's time to finish the game. And to do that, you'll use **collision detection**.

Collision detection means detecting when two or more objects have hit each other—it's one of the basic problems in video game programming. There's sometimes a lot of math involved. Luckily, it's also something that **Unity's physics engine** does for us.

In this Unity Lab, you'll take advantage of the features in Unity's physics engine that let your GameObjects react when they bump into each other, modifying your game to **score points** when the player collides with a ball. And you'll **add sounds** that play any time two balls collide with each other.

Let's start by experimenting with collisions

A **collision** happens any time Unity's physics engine causes to GameObjects to bump into each other. When you run your game, Unity is constantly doing **collision detection**, firing off collision events every time two GameObjects touch. And you may not realize it, but you've been taking advantage of Unity's collision detection since the third Unity Lab. Remember when you added a RigidBody to your Prefab to keep balls from overlapping? The collision detection system in Unity's physics engine detected the overlaps and automatically moved the balls to keep them from overlapping.

Create a new project and add two GameObjects

Let's set up a new Unity project (we called ours *Collision_Experiment*) to experiment with how Unity's collision detection system works. Move the Main Camera, then add a Cube named Floor and a Sphere named Ball, both with Rigidbody components. Floor has "Is Kinematic checked" and Ball has "Use Gravity" checked.

Name	Type	Position	Rotation	Scale
Main Camera	Camera	(0, 4, -15)	(0, 0, 0)	(1, 1, 1)
Floor – with a Rigidbody component that has "Use Gravity" unchecked and "Is Kinematic" checked	Cube	(-5, 0, 0)	(0, 0, 10)	(30, 0.1, 10)
Ball – with a Rigidbody component that has "Use Gravity" checked	Sphere	(0, 10, 0)	(0, 0, 0)	(1, 1, 1)

Run your game. The ball will drop to the floor, then roll down. Just like you'd expect it to.

Add a script to the ball that responds to collisions

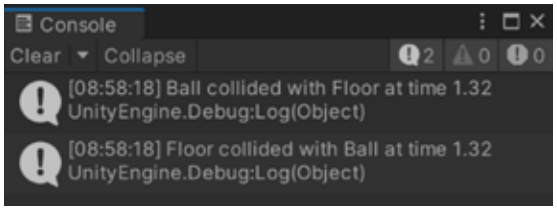
When your GameObjects needed to respond to the mouse, you added a script with OnMouseDown or OnMouseDrag methods. Now we want a GameObject to respond to collisions, and all we need to do to use Unity's collision detection is **add an OnCollisionEnter method**.

Create a **new script called CollisionBehaviour** and add this OnCollisionEnter method:

Time.time returns the time in seconds since the game started.

```
void OnCollisionEnter(Collision collision)
{
    Debug.Log($"{this.name} collided with {collision.gameObject.name} at time {Time.time}");
}
```

That method is called every time the GameObject it's attached to collides with another GameObject. Drag the new script onto **both the Ball and Floor GameObjects**, then start the game. As soon as the ball hits the floor, you'll see two lines added to the console: one line written when Ball detects a collision with Floor, and the other written when Floor detects a collision with Ball.



The Unity editor also shows the most recent console message in the status bar at the bottom of the window.

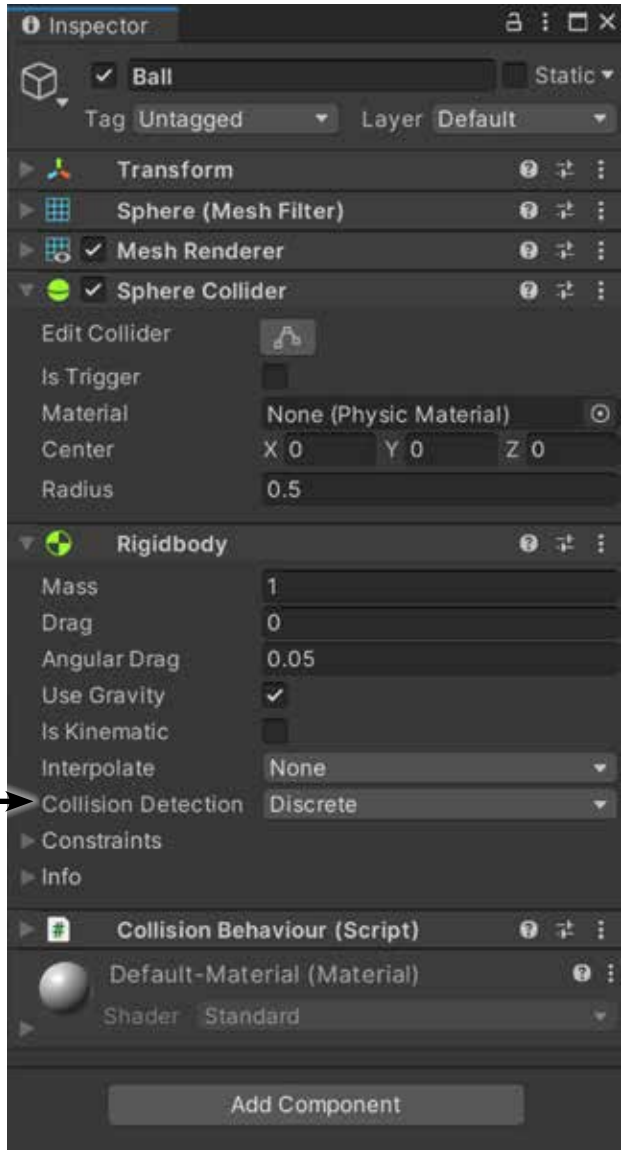
Colliders Up Close

In the previous Unity Labs, we learned that components are the fundamental building blocks of a GameObject's behavior—a Light is just a GameObject that has a Transform component that gives it its position and rotation, and a Light component that makes it emit light. So let's take a closer look at the components that make collisions work.

You already know what the Transform component does: it provides position, rotation, and scale. The next two components give the GameObject its visual shape. A mesh is a 3D shape made up of polygons. A mesh filter takes a mesh from your game's assets and passes it to a mesh renderer, which is what displays it on the screen. In this case, it uses a sphere mesh, one of the primitive meshes that's part of every Unity project.

Unity's physics engine detects a collision any time two collider components make contact with each other. While the mesh renderer defines the GameObject's visual shape, the collider defines its shape for the purpose of physical collisions. A Sphere Collider is a collider that's shaped like a sphere, which works really well for balls, and also things that need to tumble and roll (like rocks falling down a hill). You can change a Sphere Collider's radius and move its center around, but it stays a sphere.

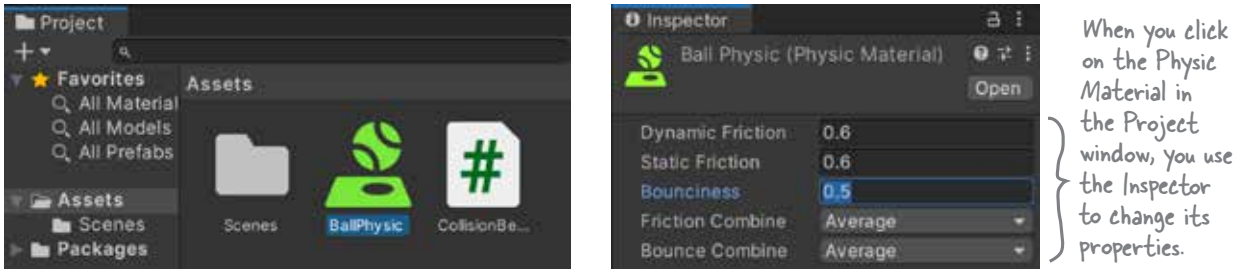
Unity only detects collisions between two GameObjects if at least one of them has a Rigidbody component and is currently moving. Most of the time we'll leave the Rigidbody's Collision Detection property set to Discrete, which is normal collision detection that checks each frame for collisions. If you have a game with extremely fast-moving GameObjects, you can change this to Continuous, which does the math to compute whether a collision happened between frames. This can slow down your game's performance, so you should leave it set to Discrete unless you really have a need for continuous collision detection.



Make the ball bounce

When you run your game, the ball just falls down and... nothing. It just starts rolling. Doesn't that look a little weird? Real objects don't quite act that way. They bounce, don't they? So we'd expect Unity's physics engine to simulate bouncing. And, in fact, it does! You can make your ball bouncy by adding a **physic material**, which lets you give your GameObjects bounciness and friction (the force that prevents things from sliding off of each other).

Click on the Assets folder in the Project window, then choose **Create >> Physic Material** from the Assets menu to add a new physic material to your Assets folder. Right-click on it and **rename it BallPhysic**. When you click on BallPhysic in the Project window, the Inspector shows you options. Use the Inspector to **set the Bounciness to 0.5**:



Now go back to the Hierarchy window and select the Ball GameObject, then look at its Sphere Collider component in the Inspector window. Its Material is currently set to None (Physic Material). Use the object selector ( ) to choose your new Physic Material, BallPhysic.

Run your game. The ball bounces! Now let's **run some physics experiments** to learn more about how bounciness works. Make sure you open the Console window to see the times that get written to the log.

- Run your game and write down the times of the first, second, and third bounces:

  First bounce: ..... Second bounce: ..... Third bounce: .....

- Select BallPhysic in the Project window. Its Bounciness value ranges from 0 to 1. Set its Bounciness to .8 and run the game again. Write down the times for each of the first three bounces:

  First bounce: ..... Second bounce: ..... Third bounce: .....

- Select the Floor GameObject and check the Inspector. It has a Box Collider component—it's a collider that works just like the sphere collider, except that it's shaped like a box. Set its Material to BallPhysic, then run the game again and write down the new times for each of the first three bounces:

  First bounce: ..... Second bounce: ..... Third bounce: .....

Run each experiment a few times. Do the times always stay consistent from run to run?

Use a mesh to change the shape of your GameObject

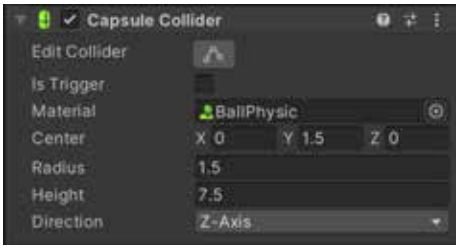
In an earlier Unity Lab, you looked at the Unity Manual page for primitive GameObjects—it was called *Primitive and placeholder objects*, and the very first thing it told you was that Unity can work with 3D models of any shape that can be created with modeling software. And now that you've started to experiment with colliders, you've got all of the basic tools you need to start doing that. So let's **modify your GameObject to use a mesh**. We'll use the 3D modeling equivalent of a "Hello, World!" program, the Utah Teapot (sometimes called the Newell Teapot, named after its creator, Martin Newell, who created it in 1975 at the University of Utah).

The Utah Teapot is a standard 3D test model.

Download the Utah Teapot mesh from GitHub (you may need to right-click on the Download button and choose "Save as…") and drag the file **into your Assets folder**: *https://github.com/head-first-csharp/fourth-edition/tree/master/Unity_Labs/Meshes/Utah_teapot.obj*

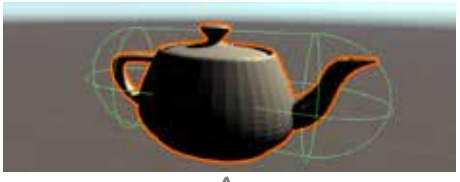
Click on Ball in the Hierarchy, then find the Mesh Filter component in the Inspector. Use the object selector (⊙) to **choose Utah_teapot**. Now your ball looks like a teapot. So go ahead and run your game.

Hmm. That ***definitely does not look right***. The teapot starts falling, hits the floor, and bounces. But then it rotates too much while hovering in the air, and after its second bounce it rotates and hovers again, and when it makes contact with the floor it sinks into it. What's going on?



Switch to a capsule collider for more accuracy

The reason the teapot looks weird is because its mesh filter makes it ***look*** like a teapot, but its sphere collider makes it ***bounce*** like a ball. So let's make it bounce a little more realistically. Use the checkbox next to the Sphere Collider component in the Inspector to disable it. Then use the Add Component button to **add a Capsule Collider**. Set its Material to BallPhysic, Center to (0, 1.5, 0), Radius to 1.5, Height to 7.5, and Direction to Z-Axis. Notice that the Scene View now shows a capsule outline around the teapot? That's the collider. As far as the physics simulation is concerned, that's the shape of the GameObject. Now **run your game again**—since the capsule is much closer to the shape of the teapot, it behaves a lot more like you'd expect it to.



The capsule collider is closer to the shape of the teapot, so it behaves more naturally.

Use a mesh collider for a realistic simulation

The teapot is better, but not perfect: you can see the edges of the teapot sink into the floor as it rolls. Let's make the simulation more accurate with a **mesh collider**. Mesh colliders make the physics simulation use a 3D mesh to treat a GameObject as if it has a specific shape. The downside is that it takes more processing power and can lower your game's framerate.

- **Disable the Sphere Collider** using its checkbox
- Use the Add Component button to **add a Mesh Collider**.
- Expand the new Mesh Collider and **check its Convex box** to tell Unity it can collide with other GameObjects. Run your game again—the teapot rolls down the floor perfectly.

Modify your game to detect collisions with the targets

We finished Unity Lab 7 by adding a OneBallBehavior script to the OneBall prefab that uses physics forces to move balls randomly around the play area. Let's pick up where that Lab left off. We'll use what we've learned about collision detection to **finish the game you started in the last Unity Lab**.

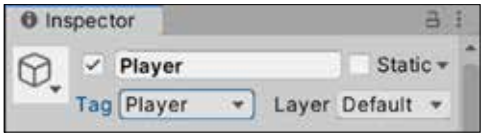
Open up the project from that lab (you named it *Unity Labs 7 and 8*). You'll add the main gameplay element: when the player collides with a billiard ball, the ball disappears and the score goes up. First, we'll modify the GameController script to keep track of the score. Then we'll add an OnCollisionEnter method to the OneBallBehavior script that updates the score—but only if the ball collides with the Player object.

Our GitHub repo has separate project folders for labs 7 and 8 to make it easier to follow along with the PDFs.

1 Update the GameObject to keep track of the score.

First, update GameController to keep track of the score, just like you did in the second Unity Lab.

```
public int Score = 0;

public void CollidedWithBall()
{
    Score++;
}
```



2 Update the GameObject to keep track of the score.

You've used tags in the last two Unity projects. You'll do the same here—**set the Player GameObject's tag to `Player`** by clicking on the Player cylinder and setting its tag to Player.

3 Modify OneBallBehavior to get a reference to the Player.

Next, **add a private GameController field to OneBallBehaviour** and set it in the Awake method.

```
private GameController gameController;

void Awake()
{
    rigidBody = GetComponent<Rigidbody>();
    gameController = GameObject.Find("Main Camera").GetComponent<GameController>();
}
```

4 Add collision detection to the OneBallBehaviour script.

Now **add the OnCollisionEnter method** to the OneBallBehaviour class that calls CollidedWithBall:

```
void OnCollisionEnter(Collision collision)
{
    if (collision.gameObject.CompareTag("Player"))
    {
        gameController.CollidedWithBall();
        Destroy(gameObject);
    }
}
```

When the ball collides with another GameObject, Unity calls its OnCollisionEnter method and passes it a Collision object that has a reference to the other GameObject. We'll use the GameObject.CompareTag method to check if the collided object has the Player tag.

Run your game. When the player collides with a ball, the ball disappears and the GameController.Score field goes up.

There are definitely some challenges in this exercise. Take it one part at a time. You can do this!

Give your game **two modes**—playing and game over—and **add a UI to your game** with a "Play Again" button and text with the score. The code will be *really similar* to code that you wrote in Unity Lab #4 ("User Interfaces"), so go back and have another look at it. Then see how much of this exercise you do using the code you wrote for Lab #4 as a template before you peek at the solution.

Take another look at Unity Lab #4 ("User Interfaces") if you need a refresher on how to create a UI in Unity.

Create a UI and add a Text GameObject. Add a UI with a Text GameObject named Score in the upper right corner of the Canvas. This will exactly like how you added the Text and Button GameObjects in the second Unity Lab. Put the score on the right side of the screen—try setting Pos X to -150 and Pos Y to -50 and adjust them from there. Use the color selector to choose a color that you can see against the play area—we used color `98F3F3`—and set its font size to 18. And since the Text is on the right side of the screen, make its paragraph alignment right-justified.

You can even change the font if you'd like—just drag any TrueType font (*.ttf) into your project (try creating a Fonts folder for it)—we used `Consolas`—then use the Select Font window by clicking the object selector ( ) next to Font.

Add a Button to the UI. Add a Button GameObject in the center of the Canvas with the text **`Play Again!`** – this will work exactly Button GameObjects in the second Unity Lab.

Add `using UnityEngine.UI;` to the top of your GameController.cs file. Then **add four fields** to your GameController class:

```
public bool GameOver = true;
public int MaxScore = 10;
public Text ScoreText;
public Button PlayAgainButton;
```

Here's a chance to use the Quick Fix menu. Make sure to add `using UnityEngine.UI;` to your GameController.cs file, or your code won't compile.

Set the GameController fields so they have references to the UI GameObjects you added. Select Main Camera in the Hierarchy window and **use the Inspector to set the fields** to reference the UI GameObjects that you just added, either by dragging the Score and Button GameObjects onto the fields, or by using the object selector ( ).

Add methods to update the score and start the game. Add an **Update method** to your GameController class that sets the Text score in the upper right corner to **`"Score: "`** followed by the score, the text **`" of "`**, and the maximum score. Then **add a method called StartGame** to your GameController class hides the "Play Again!" button, resets the score to 0, and sets the GameOver field to false.

Make the Button start the game. Click on the Button in the Hierarchy window and use the On Click () box to make the button call the GameController.StartGame method – this is identical to what you did in the last Unity project.

Clean up the play area when the game is over. Modify the **OneBallBehaviour.Update method** so it destroys the ball if the game is over, to clean up the play area when the game ends. You can keep the call to Debug.DrawRay.

Stop the game when it's over. Modify the **GameController.DropABall method** so that it only adds a ball if the game is not over. Then modify GameContorller.CollidedWithBall to end the game if the player reaches the maximum score.

Once you've done all of that, run your game. You might want to speed up your player by adjusting the values in the Player GameObject's NavMesh Agent component. Wait a minute—***the player moves even when the game is over***.

Here's one more challenge: Can you figure out how to keep the player from moving if the game is over?

The first thing we asked you to do is add the UI GameObjects, just like you did in Unity Lab #4.

Here's a quick review of how to do that:

1. Choose GameObject >> Text from the menu. That adds a Canvas to the Hierarchy window, with a Text underneath it.
2. Use the Inspector to set the name of the Text to `Score`. Use the Rect Transform component to set PosX to -150 and PosY to 150. Click the Anchor box and hold down alt-shift (or alt-option-shift on a Mac) to set both the pivot and position to top right. Use the Text component to set the font, font size, and color.
3. Choose GameObject >> Button from the menu to add a Button under Canvas. Expand the Button in the Hierarchy window, select the Text GameObject underneath it, and use its Text component to set the text to `Play Again!`.

And here's all of the code that you needed to add **to the OneBallBehaviour class**:

```
    void Update()
    {
        Debug.DrawRay(transform.position, forceAdded, Color.white);
        if (gameController.GameOver)
        {
            Destroy(gameObject);
        }
    }
```

To keep the player from moving when the game is over, **add a GameController field to the MoveToClick class** and set it in an Awake method, exactly like the OneBallBehaviour code above.

```
private NavMeshAgent agent;
private GameController gameController;

void Awake()
{
    agent = GetComponent<NavMeshAgent>();
    gameController = GameObject.Find("Main Camera").GetComponent<GameController>();
}
```

Then **modify the MoveToClick.Update method** by taking the existing contents of the method and wrapping them in an if statement that checks if the game is over:

```
    void Update()
    {
        if (!gameController.GameOver)
        {
            // The previous contents of the Update method go here
        }
    }
```

You can also combine the two if statements at the top of the Update method instead of using nested if statements:

```
        if (!gameController.GameOver && Input.GetMouseButtonDown(0))
```

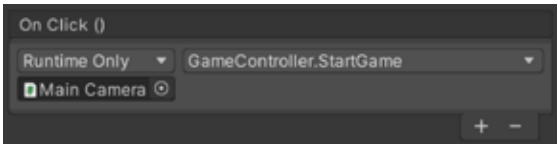
They both do the same thing! It's up to you to decide which one is easier to understand. This is an example of how you write code in two different ways that have the same behavior.

Here's the updated GameController class. You modified it to **add fields** to store the UI objects and score and manage the game mode, add the **StartGame and Update methods**, and only drop balls if the game is not over.

```
using UnityEngine.UI;

public class GameController : MonoBehaviour
{
    public float Height = 10F;
    public float RepeatRate = 1F;
    public GameObject Prefab;
    public int Score = 0;

    public bool GameOver = true;
    public int MaxScore = 10;
    public Text ScoreText;
    public Button PlayAgainButton;

    void Start()
    {
        InvokeRepeating("DropABall", 0F, RepeatRate);
    }

    private void DropABall()
    {
        if (!GameOver)
        {
            GameObject ball = Instantiate(Prefab);
            ball.transform.position =
                 new Vector3(10f - Random.value * 20f, 5f, 5f - Random.value * 10f);
        }
    }

    public void CollidedWithBall()
    {
        Score++;
        if (Score >= MaxScore)
        {
            GameOver = true;
            PlayAgainButton.gameObject.SetActive(true);
        }
    }

    void Update()
    {
        ScoreText.text = $"Score: {Score} of {MaxScore}";
    }

    public void StartGame()
    {
        PlayAgainButton.gameObject.SetActive(false);
        Score = 0;
        GameOver = false;
    }
}
```

While you're playing the game, click on GameController in the Hierarchy window and check the GameOver box to set the field to true. The game stops playing, but the button doesn't show up. Does it make sense to create a separate script for the button that checks the GameController object's GameOver field and either shows or hides the button? This is another one of those "no right or wrong answer" questions. Try modifying your game to do that instead—see which way you like better.

Don't forget to use the inspector to make the button's OnClick event call the GameController.StartGame method.



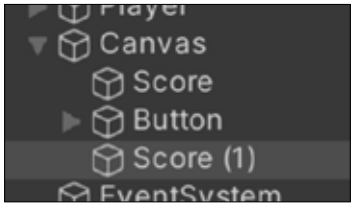
Unity calls a GameObject's OnCollisionEnter method any time it collides with another GameObject. Your game uses that to detect when the player and ball collide.

Add a timer

Let's make the game feel more competitive by **giving the player a timer** so the goal of the game is to beat your lowest tie. But instead of adding a new Text GameObject to your Canvas, let's **duplicate the Score Text**.

1. **Duplicate the Score text.** Expand Canvas in the Hierarchy window, right-click on Score, and choose Duplicate. The editor will create a new GameObject called Score (1) that's an exact copy of Score.

   - Rename it *Time*.
   - Use the Rect Transform to set its width to 250.
   - Anchor it to the top left and set its alignment so it's left-aligned.
   - Set its Pos X to the negative of the Pos X of the Score Text GameObject (so if Score.PosX is -150, set Timer.PosX to 150), and set its Pos Y to be identical to Score.
   - Use the Text component to set its text to: `Time elapsed: 0.0`



When you duplicate the Score Text, a copy of it appears in the Hierarchy.

2. **Add fields to your GameController class.** You'll need one field to keep track of the elapsed time, and another to store a reference to the Text GameObject that you just added.

```
private float GameTimer = 0f;
public Text TimerText;
```

   Then use the object selector ( ) to bring up the Set GameObject window and set the TimerText field to the Text GameObject.

3. **Modify the Update method to display the time and update it if the game is running.** Here's the code to add:

```
void Update()
{
    ScoreText.text = $"Score: {Score} of {MaxScore}";
    if (!GameOver)
    {
        GameTimer += Time.deltaTime;
    }
    TimerText.text = $"Time elapsed: {GameTimer:0.00}";
}
```

4. **Reset the time when the game starts.** Add this line of code to the StartGame method:

```
GameTimer = 0;
```

Now run your game again. You'll see a timer that starts running when your game starts, and stops when you get the last ball and the game ends.

Add sounds to your game

Sounds and audio can be an important part of the aesthetics of any video game, and Unity has a very effective system that lets you manage sounds. Click on your Main Camera and **look for the Audio Listener component**:

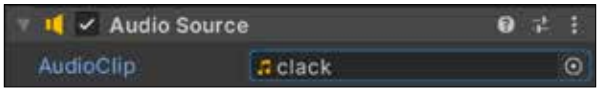
Click the manual button and read the Unity reference for AudioListener. It's not long, and really informative!

An **Audio Listener** acts as a virtual microphone in your scene. Your GameObjects can make sounds during your game, and the Audio Listener picks up those sounds and plays them back through the computer's speakers.

Go to *https://github.com/head-first-csharp/fourth-edition/tree/master/Unity_Labs/Sounds* and **download two audio files**, `clack.mp3` and `chirp.mp3`. Create a **new folder called Sounds** and drag them into it.

We want our ball to make a clacking sound every time it hits another ball, so the first thing we'll need to do is add an audio source to our prefab. This is just like how you added a component to a prefab in the second Unity Lab: go to the Project window and **click on the OneBall Prefab** in the Prefabs folder, then click Add Component in the Inspector and **choose Audio >> Audio Source**.

Use the object selector ( ) next to AudioClip to bring up the Select AudioClip window. Choose Clack:



The object selector button brings up the Select AudioClip window, just like it brings up other Select windows in other components.

Now run your game. Wait a minute—the **sound happens every time a ball is instantiated**. What's happening?

Take a closer look at the Audio Source component. Notice how the Play On Awake box is checked? That causes the audio source to play whenever the GameObject's Awake method is called. So uncheck it.

Run your game again. Now the balls don't make any sound.

You've got everything you need to add audio to your game... but there are a bunch of problems to solve to get there! The solution to this exercise walks you through all of the steps to do it. See how far you can get before you peek at it.

Can you use what you already know to figure out how to make the balls play `clack.mp3` when they collide with each other, and make the game play `chirp.mp3` when a ball is destroyed?

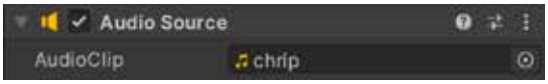
1. You'll need to find a way to detect collisions between two balls. *(Hint: create a tag called `Ball`. Use the GameObject's CompareTag method to check if the collided object has the tag `Ball`.)*

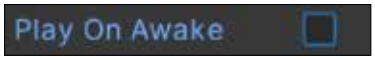
2. You'll also need to add an AudioSource component to Main Camera and set its AudioClip to `chirp.mp3`.

3. If you have a reference to an AudioSource component, the **AudioSource.Play method** will play the sound. You can set an AudioSource field in the Awake method, just like you set the Rigidbody field in OneBallBehaviour.Awake.

We asked you to modify your game to play the "clack" sound any time two balls hit each other, and the "chirp" sound whenever a ball was destroyed. And we gave you a starting point: you added an AudioSource component to the OneBall prefab, and we told you that you could play its sound by getting a reference to the AudioSource component and calling its Play method. But that's all we gave you—there were a lot of other problems to solve! How much were you able to figure out?

Step 1. Add an Audio Source component to the Main Camera. Set its AudioClip to chirp, and uncheck Play On Awake.





Step 2. Add an AudioSource field to GameController, and add an Awake method to set it:

```
private AudioSource audioSource;

void Awake()
{
    audioSource = GetComponent<AudioSource>();
}
```

Step 3. Modify the GameController.CollidedWithBall method to play the audio clip:

```
public void CollidedWithBall()
{
    audioSource.Play();
    // The rest of the method is the same
```

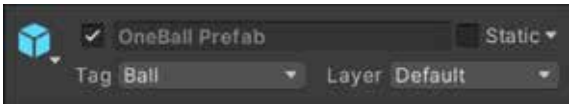
You got a reference to the AudioSource component in the Awake method, so now you can use it to play its sound.

Step 4. Add an AudioSource field to OneBallBehaviour, and an identical line in OneBallBehaviour.Awake to set it.

```
private AudioSource audioSource;

void Awake()
{
    audioSource = GetComponent<AudioSource>();
    // The rest of the method is the same
```

We've said it a bunch of times already, but it's worth repeating: it is not cheating to peek at the exercise solution while you're trying to solve it. Trying to solve a problem and then looking at the solution is a great way to get it into your brain.

Step 5. Create a new Ball tag, then set the OneBall Prefab tag to Ball.



Step 6. Call the AudioSource.Play method to play the sound any time the ball collides with another ball:

```
void OnCollisionEnter(Collision collision)
{
    if (collision.gameObject.CompareTag("Ball"))
    {
        audioSource.Play();
    }
    // The rest of the OnCollisionEnter method stays the same
```

Use the label to play the "clack" sound only if a ball collided with another ball.

Now your game plays the chirp sound when the player collects a ball, and the clack sound when to balls collied.

Physics experiment, part 2: The ball that stopped falling

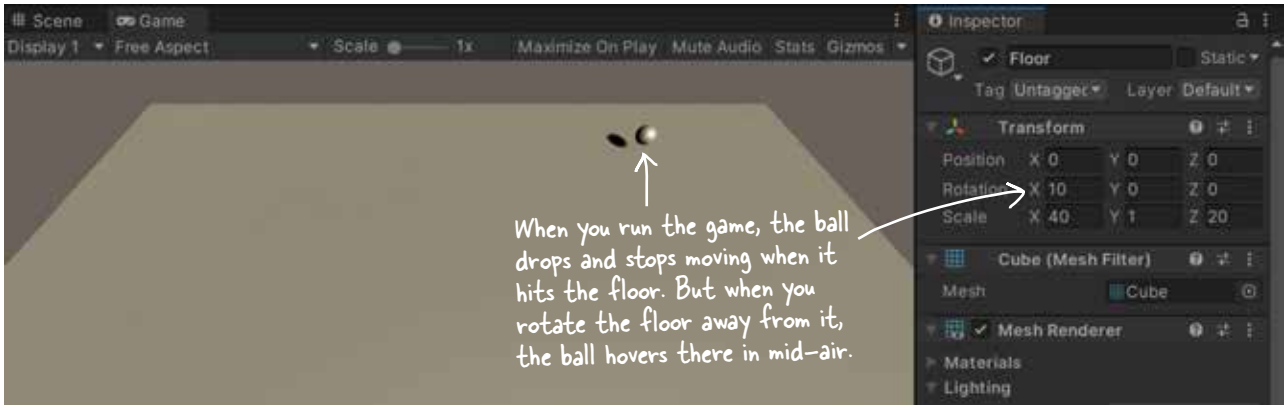
When you ran your physics experiments in the last Unity Lab, there was one weird behavior that never got a good explanation: **the mystery of the ball that stopped falling**. Let's reproduce that part of the experiment.

Create a **new Unity project** (we called ours *Sleep_Experiment*). Move the Main Camera to a new position, and add two GameObjects, a floor and a ball (like in the first experiment) to the scene:

Name	Type	Position	Rotation	Scale
Main Camera	Camera	(0, 15, -9)	(65, 0, 0)	(1, 1, 1)
Floor - don't add a Rigidbody component	Cube	(0, 0, 0)	(0, 0, 0)	(40, 1, 20)
Ball - with Rigidbody, "Use gravity" checked	Sphere	(10, 5, 5)	(0, 0, 0)	(1, 1, 1)

Now you're all set to re-run your physics experiment. Press play to start your game—the sphere will start falling, and stop when it hits the floor.

While the game is running, **select the cube in the Hierarchy window and set its X rotation to 10**. The sphere should now be hovering over the floor—which is weird! What's going on? Why is the ball just hovering there?



When you start the game, the sphere starts falling. Rotate the floor away from the sphere, and it just hovers in mid-air, ***even though "Use gravity" is checked***. If you click on the X label in the Rotation row of the cube's Transform component and drag up and down over and over again, eventually the sphere "remembers" that it has to obey gravity and starts falling again. Why is it just hovering there and not falling?

Even more confusingly, if you click and drag up and down on the cube's X rotation to tilt it backward and forward or adjust its Y position up and down so it hits the ball, eventually the sphere will start falling again. What's going on?

Can you guess why the ball stopped responding to gravity when you rotated the floor away from it? What do you think?

Unity puts Rigidbodies to sleep when they stop moving

The reason that the ball is just hovering there is that it's **sleeping**. When a Rigidbody stops moving (or is moving very, very slowly), the physics engine stops keeping track of it. Instead, it puts the Rigidbody to sleep, temporarily removing it from the physics simulation so it doesn't need to waste CPU cycles on it. If another GameObject collides with it or a force is applied, the physics engine will wake it up again.

Most of the time you don't notice any of this because it happens automatically. But sometimes Unity doesn't wake up a sleeping Rigidbody—for example, if a **static collider** (or a collider without a Rigidbody) is moved into it or away from it because its position was changed by modifying its Transform component.

Luckily, we can tell if an object is asleep by calling its Rigidbody.IsSleeping method, and wake it up by calling its Rigidbody.WakeUp method. Go ahead and try that right now. **Add a C# script to the sphere**:

```
private Rigidbody rigidBody;

void Awake()
{
    rigidBody = GetComponent<Rigidbody>();
}

void Update()
{
    Debug.Log($"Sleeping: {rigidBody.IsSleeping()}  Time: {Time.time:0.00}");
}

void OnMouseDown()
{
    rigidBody.WakeUp();
}
```

Take a minute and read the Rigidbody overview page in the Unity manual. It tells you a little more about rigidbody sleeping.

When you click the sphere, its OnMouseDown method wakes up its Rigidbody.

The Update method logs a message with the time since the game started and whether or not the sphere's Rigidbody is sleeping.

Start the game again and wait for the ball to drop. The Update method will log False while the GameObject is awake. When it hits the floor, the physics engine will put it to sleep and its Update method will log True:

```
Sleeping: False  Time: 22.63
```

Rotate the floor away again—the ball will stay asleep. Then click on the sphere that's hovering over the floor to wake it up. As soon as it's awake, gravity kicks in and it starts falling again until it hits the floor and goes back to sleep.

Specifying formats with string interpolation

In Chapter 4 you learned about passing a format argument to ToString, and used it to convert a decimal to a price: price.ToString("c") – the "c" argument tells it to format the string like a currency. You can specify formats with string interpolation, too. The expression{GameTimer:0.00} makes string interpolation to do the same thing as GameTimer.ToString("0.00") – the 0.00 format rounds the number to two decimal places.

Take a few minutes and read the C# documentation about string interpolation, because it's a really powerful tool. Not only can you include formats, but you can also include alignments. The documentation includes links to some great tutorials: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/tokens/interpolated

You should be saving early and saving often. Let's learn that lesson the easy way, so you don't have to learn it the hard way.

But let's do one more little experiment to help make this sink in.

How to crash Unity in three easy steps

Step 1: *Add a new GameObject to your scene—it doesn't matter what kind of object you add.*

Step 2: *Add C# script to that GameObject called CrashUnity. Do not save your scene. Edit the script and add an infinite loop to the CrashUnity.Update method:*

```
void Update()
{
    while (true)
    {
        transform.position = Vector3.zero;
    }
}
```

Step 3: *Save the script, but do not save your scene. Run your game. Nothing happens. Wait. Wait longer. Keep waiting. Bored yet? No? Well, keep waiting. And waiting.*

Hmm… it looks like Unity crashed! Open the Windows Task Manager, click on Unity, and then click End Task to force Unity to quit. (On a Mac, right-click on it in the dock and choose Force Quit. On Unix, kill the Unity process.) Go back to Unity Hub and re-open your project.

Uh-oh—looks like ***the GameObject that you added disappeared****! That's why you need to save your scene often. If one of your scripts has an infinite loop, Unity will freeze, and you won't get a chance to save the scene.*

But wait a minute—the CrashUnity script is still there! That's because when you created the script, Unity actually created the .cs file in the Scripts folder. And that makes sense, because Visual Studio had to be able to edit and save the script outside of Unity. That's why it still shows up in the Unity editor.

Get creative!

Can you find ways to improve your game and get practice writing code? Here are some ideas:

- Can you figure out how to score more points if the ball is moving faster? You can use `Rigidbody.velocity.magnitude` to get the speed.
- Here's a tougher one: can you figure out how to make the game score extra points if the player hits the ball before it lands? You'll need to watch for a collision with the floor.
- Can you find other ways to make the game more interesting? How about making the balls launch into the air if the player clicks on them, and the player scores score more for higher targets (with a higher `transform.position.y` value)?
- Try making each target keep track of how many other targets it collided with. If a ball hits too many other balls, it disappears and the player loses a point.

You can use Unity's physics engine to move your GameObjects automatically, instead of moving them manually.

BULLET POINTS

- **Collision detection** means detecting when two or more objects have hit each other. When the physics engine detects a collision between two GameObjects, it calls their **OnCollisionEnter methods**.
- Unity **detects collisions** between two GameObjects if at least one of them has a Rigidbody component and is currently moving.
- A GameObject's **mesh filter** component takes a mesh—a 3D shape made up of polygons—from your game's assets and passes it to a **mesh renderer**, which displays it on the screen.
- The mesh renderer defines the GameObject's visual shape, while the collider defines its shape **for the purpose of physical collisions**.
- You can specify physic material in your collider's properties to give GameObjects bounciness and friction.
- Use **GameObject.CompareTag** to check an object's tag.
- You can get more accurate physics with a collider that matches the shape of the mesh used with the renderer. Primitive colliders like **sphere and capsule colliders** have the same shapes as primitive GameObjects.
- A **mesh collider** lets the physics engine accurately simulate collisions by creating a collider that's the shape of a mesh, at a cost of increased processing power and potentially lower frame rates.
- The **AudioSource component** lets GameObjects play sounds that are picked up by the camera's AudioListener. Uncheck **"Play On Awake"** to keep the AudioSource from playing the sound as soon as the GameObject is instantiated.
- When the physics engine detects a Rigidbody that's stopped moving, it tells it to **sleep**. This is normally undetectable, but if a floor falls away from a sleeping GameObject it won't respond to gravity until it wakes up.
- A sleeping Rigidbody will **wake up** when another GameObject collides with it. You can also call its Rigidbody component's **WakeUp method**.
- When using string interpolation, specify **string formats** by adding a colon followed by the format after the variable name. The format "0.00" rounds a number to two decimal places.
- The Unity editor will freeze if a GameObject's Update method is stuck in an infinite loop.