Head First

C#

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

Andrew Stellman
& Jennifer Greene

A Brain-Friendly Guide

Unity Lab #3
GameObject Instances

C# is an object-oriented language, and since these Head First C# Unity Labs are all **about getting practice writing C# code**, it makes sense that these labs will focus on creating objects.

You've been creating objects in C# since you learned about the **new** keyword in Chapter 3. In this Unity Lab you'll **create instances of a Unity GameObject** and use them in a complete, working game. This is a great jumping-off point for writing Unity games in C#.

The goal of the next two Unity Labs is to **create a simple game** using the familiar billiard ball from the last lab. In this lab, you'll build on what you learned about C# objects and instances to start building the game. You'll use a **prefab**—Unity's tool for creating instances of GameObjects—to create lots of instances of a GameObject, and you'll use scripts to make your GameObjects fly around your game's 3D space.

Let's build a game in Unity!

Unity is all about building games. So in the next two Unity Labs, you'll use what you've learned about C# to build a simple game. Here's the game that you're going to create:

When you start the game, the scene slowly fills up with billiard balls. The player needs to keep clicking them to make them disappear. Once there are 15 balls in the scene, the game is over.
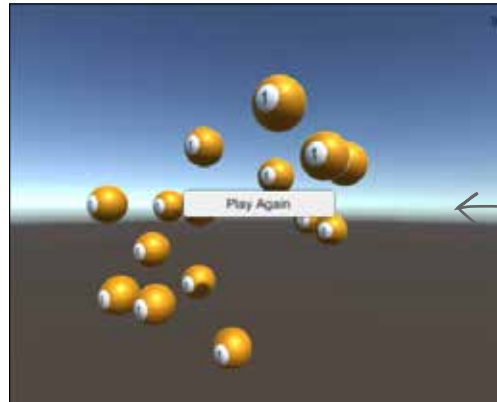
The game displays a score in the upper-right corner. Players score a point for each ball they click on.

When the game is over, a Play Again button lets the player start a new game.

So let's get started. The first thing you'll do is get your Unity project set up. This time we'll keep the files a little more organized, so you'll create separate folders for your materials and scripts—and one more folder for prefabs (which you'll learn about later in the lab):

1. Before you begin, close any Unity project that you have open. Also close Visual Studio—you'll let Unity open it for you.

2. **Create a new Unity project** using the 3D template, just like you did for the previous Unity Labs. Give it a name to help you remember which labs it goes with—name it *Unity Labs 3 and 4*.

3. Choose the Wide layout so your screen matches the screenshots.

4. Create a folder for your materials underneath the Assets folder. **Right-click on the Assets folder** in the Project window and choose Create >> Folder. Name it *Materials*.

5. Create another folder under Assets named *Scripts*.

6. Create one more folder under Assets named *Prefabs*.
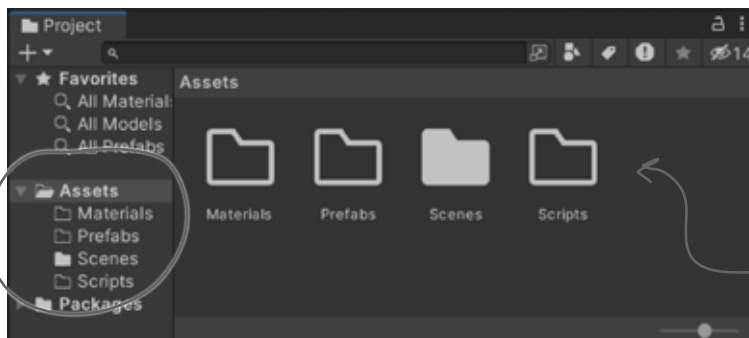
Make sure you create the Materials, Scripts, and Prefabs folders underneath Assets.

The Project window shows folders as outlines when they're emtpty.

Create a new material inside the Materials folder

Double-click on your new Materials folder to open it. You'll create a new material here.

Go to https://github.com/head-first-csharp/fifth-edition and click on the Billiard Ball Textures link (just like you did in the first Unity lab) and download the texture file *1 Ball Texture.png* into a folder on your computer, then add it into your Materials folder—just like you did with the downloaded file in the first Unity Lab: right-click on the Materials folder you just created, then choose *Import New Asset…* and add the file.

Now you can create the new material. Right-click on the Materials folder in the Project window and **choose Create >> Material**. Name your new material **1 Ball**. You should see it appear in the Materials folder in the Project window.
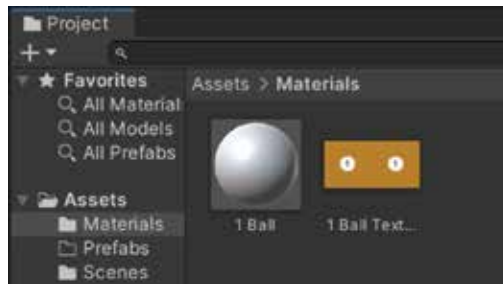
> In the previous Unity Labs we used a <u>texture</u>, or a bitmap image file that Unity can wrap around GameObjects. When you dragged the texture onto a sphere, Unity automatically created a <u>material</u>, which is what Unity uses to keep track of information about how a GameObject should be rendered that can have a reference to a texture. This time you're creating the material manually. Just like last time, you may need to click the Download button on the GitHub page to download the texture PNG file.

Make sure the 1 Ball material is selected in the Materials window, so it shows up in the Inspector. Click on the *1 Ball Texture* file and **drag it into the box to the left of the Albedo label**.

Select the 1 Ball material in the Project window so you can see its properties, then drag the texture map onto the box to the left of the Albedo label.

You should now see a tiny little picture of the 1 Ball texture in the box to the left of Albedo in the Inspector.

Now your material looks like a billiard ball when wrapped around a sphere.

Behind the Scenes

GameObjects reflect light from their surfaces.

When you see an object in a Unity game with a color or texture map, you're seeing the surface of a GameObject reflecting light from the scene, and the **albedo** controls the color of that surface. Albedo is a term from physics (specifically astronomy) that means the color that's reflected by an object. You can learn more about albedo from the Unity Manual. Choose "Unity Manual" from the Help menu to open the manual in a browser and search for "albedo"—there's a manual page that explains albedo color and transparency.

Spawn a billiard ball at a random point in the scene

Create a new Sphere GameObject with a script called OneBallBehaviour:

★ Choose 3D Object >> Sphere from the GameObject menu to **create a sphere**.

★ Drag your new **1 Ball material** onto it to make it look like a billiard ball.

★ Next, **right-click on the Scripts folder** that you created in the Project window, click Create in the menu, and choose C# Script. This will **create a new C# script** named OneBallBehaviour.

★ **Drag the script onto the Sphere** in the Hierarchy window. Select the sphere and make sure a Script component called "One Ball Behaviour" shows up in the Inspector window.

Double-click on your new script to edit it in Visual Studio. ***Add exactly the same code*** that you used in BallBehaviour in the first Unity Lab, then **comment out the Debug.DrawRay line** in the Update method.

Your OneBallBehaviour script should now look like this:

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

    // Start is called before the first frame update
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {
        Vector3 axis = new Vector3(XRotation, YRotation, ZRotation);
        transform.RotateAround(Vector3.zero, axis, DegreesPerSecond * Time.deltaTime);
        // Debug.DrawRay(Vector3.zero, axis, Color.yellow);
    }
}
```

We won't include the using lines in the script code, but assume they're there.

When you add a Start method to a GameObject, Unity calls that method every time a new instance of that object is added to the scene. If the Start method is in a script attached to a GameObject that appears in the Hierarchy window, that method will get called as soon as the game starts.

Unity often instantiates a GameObject some time before it's added to the scene. It only calls the Start method when the GameObject is actually added to the scene.

You won't need this line, so comment it out.

Now modify the Start method to move the sphere to a random position when it's created. You'll do this by setting **transform.position**, which changes the position of the GameObject in the scene. Here's the code to position your ball at a random point—**add it to the Start** method of your OneBallBehaviour script:

```
    // Start is called before the first frame update
    void Start()
    {
        transform.position = new Vector3(3 - Random.value * 6,
            3 - Random.value * 6, 3 - Random.value * 6);
    }
```

Remember, the Play button does not save your game! Make sure you save early and save often.

Use the Play button in Unity to run your game. A ball should now be circling the Y axis at a random point. Stop and start the game a few times. The ball should spawn at a different point in the scene each time.

Use the debugger to understand Random.value

You've used the Random class in several projects—specifically, its static Random.Shared property. You used it to scatter the animals in the animal matching game, pick random cards, and roll for damage. Unity's Random class is not the same class! Hover over the Random keyword in Visual Studio:

These classes are both called Random, but if you hover over them in Visual Studio to see the tooltip, you'll see that the one you used earlier is in the System namespace. Now you're using the Random class in the <u>UnityEngine namespace</u>.

```
// Update is called once per frame
@ Unity Message | 0 references
void Update()
{
    transform.position = new Vector3(3 - Random.value * 6,
        3 - Random.value * 6, 3 - Random.value * 6);
}
```

class UnityEngine.Random
Easily generate random data for games.

```
int
{
    int total = 0;
    for (int i = 0; i < numberOfRows; i++) total += Random.Shared.Next(1, 7);
    return total;
}
```

class System.Random
Represents a pseudo-random number generator, which is an algorithm that produces a sequence of numbers that meet certain statistical requirements for randomness.

GitHub Examples and Documentation

This is from the code you wrote earlier to to roll for damage in Owen's app.

You can see from the code that this new Random class is different from the one you used before. Earlier you called Random.Next to get a random value, and that value was a whole number. This new code uses **Random.value**, but that's not a method—it's actually a property.

Unity may prompt you to enable debugging, just like in the last Unity Lab.

Use the Visual Studio debugger to see the kinds of values that this new Random class gives you. Click the "Attach to Unity" button (▶ Attach to Unity ▾ in Windows, ▶ ☐ Debug ▸ ⊙ Attach to Unity in macOS) to attach Visual Studio to Unity. Then **add a breakpoint** to the line you added to the Start method.

Now go back to Unity and **start your game**. It should break as soon as you press the Play button. Hover your cursor over Random.value—make sure it's over **value**. Visual Studio will show you its value in a tooltip:

Keep Visual Studio attached to Unity and restart your game a few times. You'll get a new random number between 0 and 1 each time you restart it.

```
 8        void Start()
 9        {
●10           transform.position = new Vector3(3 - Random.value * 6,
11         ▸| 3 - Random.value * 6, 3 - Random.value * 6);
12        }
13
```
Random.value   0.03689707

Keep Visual Studio attached to Unity, then go back to the Unity editor and **stop your game** (in the Unity editor, not in Visual Studio). Start your game again. Do it a few more times. You'll get a different random value each time. That's how UnityEngine.Random works: it gives you a new random value between 0 and 1 each time you access its value property.

Press Continue (▶ Continue ▾) to resume your game. It should keep running—the breakpoint was only in the **Start method**, which is *just called once for each GameObject instance*, so it won't break again. Then go back to Unity and stop the game.

You can't edit scripts in Visual Studio while it's attached to Unity, so click the square Stop Debugging button to detach the Visual Studio debugger from Unity.

Turn your GameObject into a prefab

In Unity, a **prefab** is a GameObject that you can instantiate in your scene. Over the past few chapters you've been working with object instances, and creating objects by instantiating classes. Unity lets you take advantage of objects and instances, so you can build games that reuse the same GameObjects over and over again. Let's turn your 1 ball GameObject into a prefab.

Watch it!

Visual Studio won't let you edit code while it's attached to Unity.

If you try to edit your code but find that Visual Studio won't let you make any changes, that means Visual Studio is probably still attached to Unity! Press the square Stop Debugging button to detach it.

GameObjects have names.. Change the name of your GameObject to *OneBall*. Start by **selecting your sphere**, by clicking on it in the Hierarchy window or in the scene. Then use the Inspector window to **change its name to OneBall**.
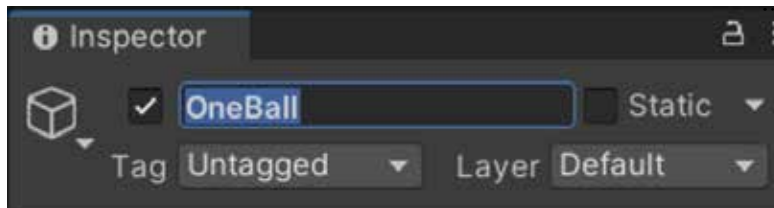
You can also rename a GameObject by right-clicking on it in the Hierarchy window and choosing Rename.
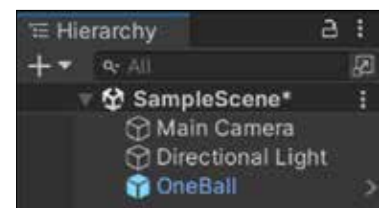
Now you can turn your GameObject into a prefab. **Drag OneBall from the Hierarchy window into the Prefabs folder**.
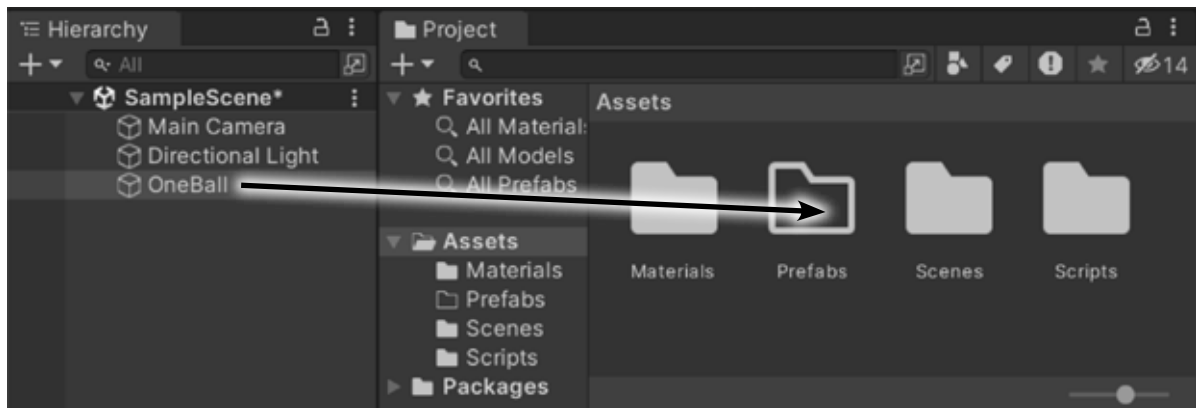
OneBall should now appear in your Prefabs folder. Notice that ***OneBall is now blue in the Hierarchy window***. This indicates that it's now a prefab—Unity turned it blue to tell you that an instance of a prefab is in your hierarchy. That's fine for some games, but for this game we want all of the instances of the balls to be created by scripts.

Right-click on OneBall in the Hierarchy window **and <u>delete</u> the OneBall GameObject from the scene**. You should now only see it in the Project window, and not in the Hierarchy window or the scene.

When a GameObject is blue in the Hierarchy window, Unity is telling you it's a prefab instance (delete it!).

Have you been <u>saving your scene</u> as you go? Save early, save often!

Create a script to control the game

The game needs a way to add balls to the scene (and eventually keep track of the score, and whether or not the game is over).

Right-click on the Scripts folder in the Project window and **create a new script called GameController**. Your new script will use two methods available in any GameObject script:

★ **The Instantiate method creates a new instance of a GameObject.** When you're instantiating GameObjects in Unity, you don't typically use the **new** keyword like you saw in Chapter 2. Instead, you'll use the Instantiate method, which you'll call from the AddABall method.

★ **The InvokeRepeating method calls another method in the script over and over again.** In this case, it will wait one and a half seconds, then call the AddABall method once a second for the rest of the game.

Here's the source code for it:

What's the type of the second argument that you're passing to InvokeRepeating?

```
public class GameController : MonoBehaviour
{
    public GameObject OneBallPrefab;

    void Start()
    {
        InvokeRepeating("AddABall", 1.5F, 1);
    }

    void AddABall()
    {
        Instantiate(OneBallPrefab);
    }
}
```

Unity's InvokeRepeating method calls another method over and over again. Its first parameter is a string with the name of the method to call ("invoke" just means calling a method).

This is a method called AddABall. All it does is create a new instance of a prefab.

You're passing the OneBallPrefab field as a parameter to the Instantiate method, which Unity will use to create an instance of your prefab.

Brain Power

Unity will only run scripts that are attached to GameObjects in a scene. The GameController script will create instances of our OneBall prefab, but we need to attach it to something. Luckily, we already know that a camera is just a GameObject with a Camera component (and also an AudioListener). The Main Camera will always be available in the scene. So…what do you think you'll do with your new GameController script?

Attach the GameController script to the Main Camera

Your new GameController script needs to be attached to a GameObject to run. Luckily, the Main Camera is just another GameObject—it happens to be one with a Camera component and an AudioListener component—so let's attach your new script to it. **Drag your GameController** script out of the Scripts folder in the Project window and **onto the Main Camera** in the Hierarchy window.
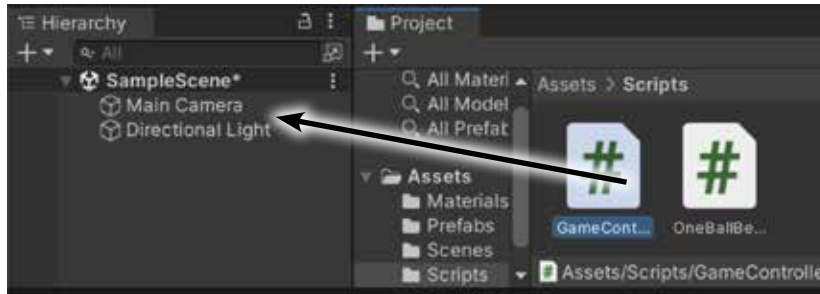

> You learned about public versus private fields in Chapter 5. When a script class has a public field, the Unity editor shows that field in the Script component in the Inspector. It adds spaces between uppercase letters to make its name easier to read.

Look in the Inspector—you'll see a component for the script, exactly like you would for any other GameObject. The script has a *public field called OneBallPrefab*, so Unity displays it in the Script component.
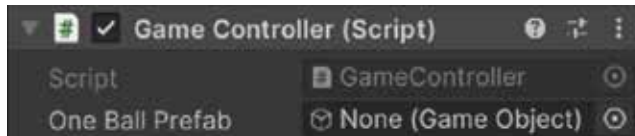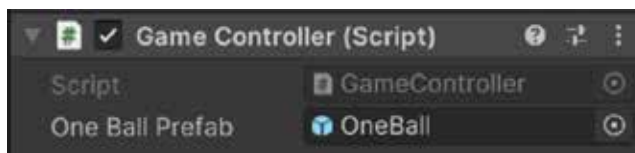

Here's the OneBallPrefab field in your GameController class. Unity added spaces before uppercase letters to make it easier to read (just like we saw in the last lab).

The OneBallPrefab field still says None, so we need to set it. **Drag OneBall out of the Prefabs folder** and onto the **box next to the One Ball Prefab label**.
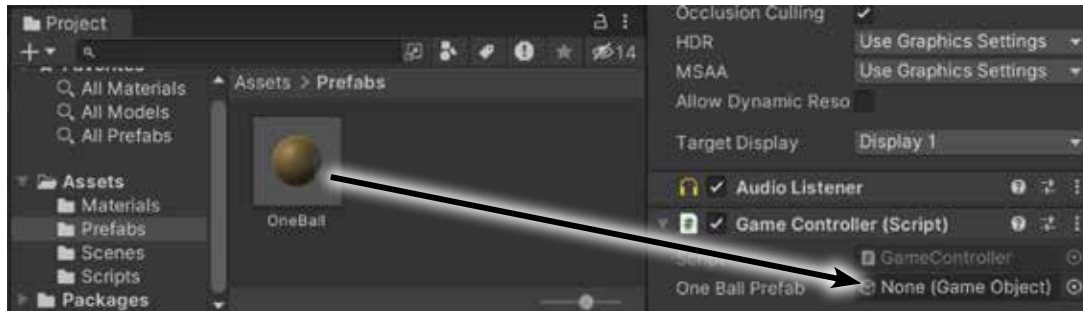

Now the GameController's OneBallPrefab field contains a *reference* to the OneBall prefab:

Go back to the code and **look closely at the AddABall method**. It calls the Instantiate method, passing it the OneBallPrefab field as an argument. You just set that field so that it contains your prefab. So every time GameController calls its AddABall method, it will *create a new instance of the OneBall prefab*.

Press Play to run your code

Your game is all ready to run. The GameController script attached to the Main Camera will wait 1.5 seconds, then instantiate a OneBall prefab every second. Each instantiated OneBall's Start method will move it to a random position in the scene, and its Update method will rotate it around the Y axis every 2 seconds using OneBallBehaviour fields (just like in the last Lab). Watch as the play area slowly fills up with rotating balls:
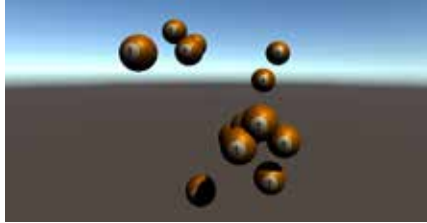

> **Unity calls every GameObject's Update method before each frame. That's called the <u>update loop</u>.**

When you instantiate GameObjects in your code, they show up in the Hierarchy window when you run your game.

Watch the live instances in the Hierarchy window

Each of the balls flying around the scene is an instance of the OneBall prefab. Each of the instances has its own instance of the OneBallBehaviour class. You can use the Hierarchy window to track all of the OneBall instances—as each one is created, a "OneBall(Clone)" entry is added to the Hierarchy.
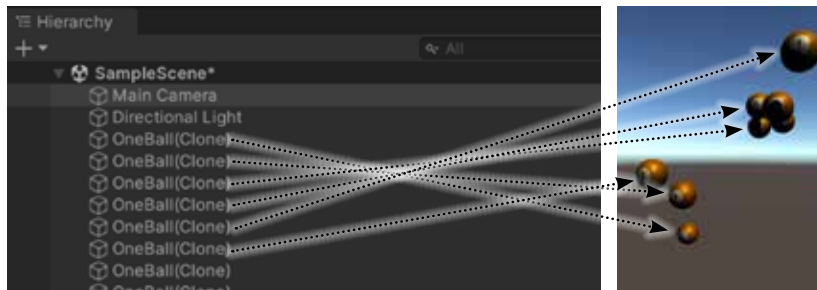
We've included some coding exercises in the Unity Labs. They're just like the exercises in the rest of the book—and remember, it's <u>not</u> cheating to peek at the solution.

Click on any of the OneBall(Clone) items to view it in the Inspector. You'll see its Transform values change as it rotates, just like in the last lab.

Exercise

Figure out how to add a BallNumber field to your OneBallBehaviour script, so that when you click on a OneBall instance in the Hierarchy and check its One Ball Behaviour (Script) component, under the X Rotation, Y Rotation, Z Rotation, and Degrees Per Second labels it has a Ball Number field:

The first instance of OneBall should have its Ball Number field set to 1. The second instance should have it set to 2, the third 3, etc. ***Here's a hint:*** *You'll need a way to keep track of the count that's shared by all of the OneBall instances. You'll modify the Start method to increment it, then use it to set the BallNumber field.*

# Use the Inspector to work with GameObject instances

Run your game. Once a few balls have been instantiated, click the Pause button—the Unity editor will jump back to the Scene view. Click one of the OneBall instances in the Hierarchy window to select it. The Unity editor will outline it in the Scene window to show you which object you selected. Go to the Transform component in the Inspector window and **set its Z scale value to 4** to make the ball stretch.

Start your simulation again—now you can track which ball you're modifying. Try changing its DegreesPerSecond, XRotation, YRotation, and ZRotation fields like you did in the last lab.

While the game is running, switch between the Game and Scene views. You can use the Gizmos in the Scene view *while the game is running*, even for GameObject instances that were created using the Instantiate method (rather than added to the Hierarchy window).

Try clicking the Gizmos button at the top of the toolbar to toggle them on and off. You can turn on the Gizmos in the Game view, and you can turn them off in the Scene view.

Toggle visibility of all Gizmos in the Scene view
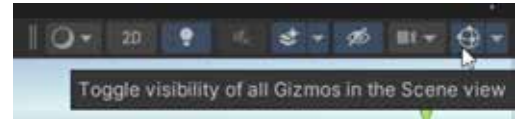
Exercise Solution

You can add a BallNumber field to the OneBallBehaviour script by keeping track of the total number of balls added so far in a static field (which we called BallCount). Each time a new ball is instantiated, Unity calls its Start method, so you can increment the static BallCount field and assign its value to that instance's BallNumber field.

```
static int BallCount = 0;
public int BallNumber;

// Start is called before the first frame update
void Start()
{
    transform.position = new Vector3(3 - Random.value * 6,
        3 - Random.value * 6, 3 - Random.value * 6);

    BallCount++;
    BallNumber = BallCount;
}
```

All of the OneBall instances share a single static BallCount field, so the first instance's Start method increments it to 1, the second instance increments BallCount to 2, the third increments it to 3, etc.

Use physics to keep balls from overlapping

Did you notice that occasionally some of the balls overlap each other?

Unity has a powerful **physics engine** that you can use to make your
GameObjects behave like they're real, solid bodies—and one thing that solid
shapes don't do is overlap each other. To prevent that overlap, you just need to tell
Unity that your OneBall prefab is a solid object.

Stop your game, then **click on the OneBall prefab in the Project window**
to select it. Then go to the Inspector and scroll all the way down to the bottom to
the Add Component button:
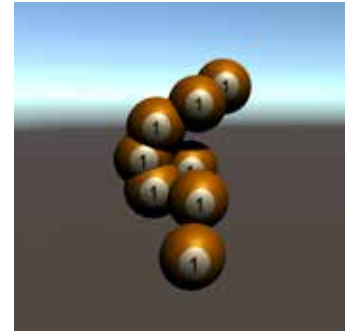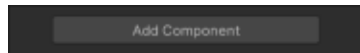
Click the button to pop up the Component window. **Choose Physics** to view the physics
components, then **select Rigidbody** to add the component.

> **Make sure that you uncheck Use Gravity. Otherwise the balls will all react to gravity and start falling, and since there's nothing for them to hit, they'll keep falling forever.**

While you're running physics experiments, here's one Galileo would appreciate. Try checking the Use Gravity box while your game is running. New balls that get created will start falling, occasionally hitting another ball and knocking it out of the way.

Run your game again—now you won't see balls overlap. Occasionally one ball will get created
on top of another one. When that happens, the new ball will knock the old one out of the way.

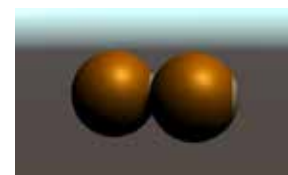**Let's run a little physics experiment** to prove that the balls really are rigid now.
Start your game, then pause it as soon as there are more than two balls created. Go to the
Hierarchy window. If it looks like this:
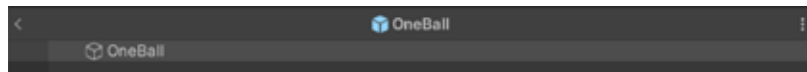
then you're editing the prefab—click the back caret (<) in the top-right corner of the
Hierarchy window to get back to the scene (you may need to expand SampleScene again).

> **You can use the Hierarchy window to delete GameObjects from your scene while the game is running.**

★ Hold down the Shift key, click the first OneBall instance in the Hierarchy window,
and then click the second one so the first two OneBall instances are selected.

★ You'll see dashes (▬) in the Position boxes in the Transform panel. **Set the
Position to (0, 0, 0)** to set both OneBall instances' positions at the same time.

★ Use Shift-click to select any other instances of OneBall, right-click, and **choose
Delete** to delete them from the scene so only the two overlapping balls are left.

★ Unpause your game—the balls can't overlap now, so instead they'll be rotating
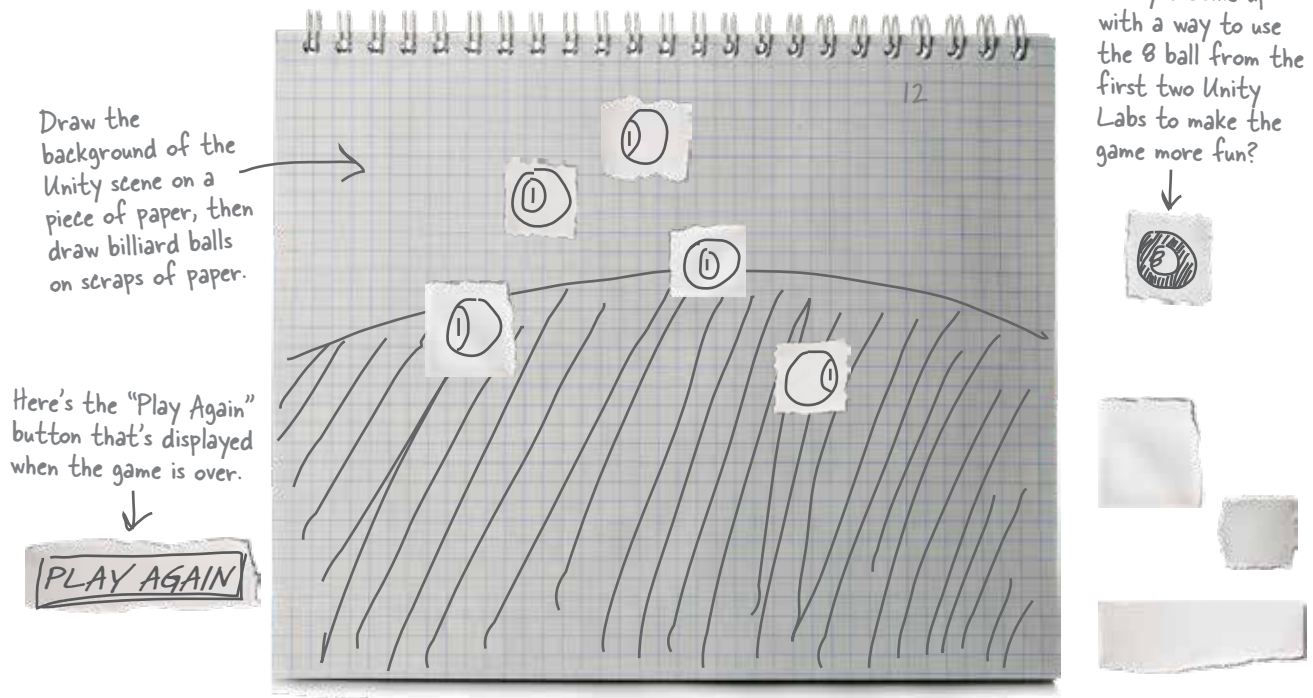next to each other.

Stop the game in Unity and Visual Studio and <u>save your scene</u>.

Get creative!

You're halfway done with the game! You'll finish it in the next Unity Lab. In the meantime, this is a great opportunity to practice your **paper prototyping** skills. We gave you a description of the game at the beginning of this Unity Lab. Try creating a paper prototype of the game. Can you come up with ways to make it more interesting?

Draw the background of the Unity scene on a piece of paper, then draw billiard balls on scraps of paper.

Here's the "Play Again" button that's displayed when the game is over.

PLAY AGAIN

Can you come up with a way to use the 8 ball from the first two Unity Labs to make the game more fun?

Bullet Points

- **Albedo** is a physics term that means the color that's reflected by an object. Unity can use texture maps for the albedo in a material.

- Unity has its own **Random class** in the UnityEngine namespace. The static Random.value method returns a random number between 0 and 1.

- A **prefab** is a GameObject that you can instantiate in your scene. You can turn any GameObject into a prefab.

- The **Instantiate method** creates a new instance of a GameObject. The Destroy method destroys it. Instances are created and destroyed at the end of the update loop.

- The **InvokeRepeating method** calls another method in the script over and over again.

- Unity calls every GameObject's Update method before each frame. That's called the **update loop**.

- You can **inspect the live instances** of your prefabs by clicking on them in the Hierarchy window.

- When you add a **Rigidbody** component to a GameObject, Unity's physics engine makes it act like a real, solid, physical object.

- The Rigidbody component lets you turn **gravity** on or off for a GameObject.