O'REILLY®

Fourth Edition

Head First C#

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

Andrew Stellman
& Jennifer Greene

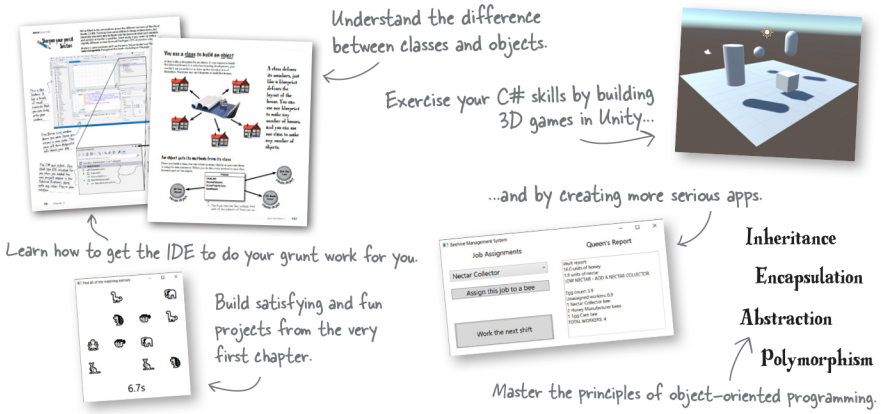
A Brain-Friendly Guide

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®

Praise for *Head First C#*

"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

"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

More Praise for *Head First C#*

"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 onwards, 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# 8.0 in a Nutshell* and *C# 8.0 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 other *Head First* books

"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

C# 8.0 in a Nutshell

C# 8.0 Pocket Reference

C# Database Basics

C# Essentials, 2nd Edition

Concurrency in C# Cookbook, 2nd Edition

Mobile Development with C#

Programming C# 8.0

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

Head First 2D Geometry

Head First Agile

Head First Ajax

Head First Algebra

Head First Android Development

Head First C

Head First Data Analysis

Head First Design Patterns

Head First EJB

Head First Excel

Head First Go

Head First HTML5 Programming

Head First HTML with CSS and XHTML

Head First iPhone and iPad Development

Head First Java

Head First JavaScript Programming

Head First Kotlin

Head First jQuery

Head First Learn to Code

Head First Mobile Web

Head First Networking

Head First Object-Oriented Analysis and Design

Head First PHP & MySQL

Head First Physics

Head First PMP

Head First Programming

Head First Python

Head First Rails

Head First Ruby

Head First Ruby on Rails

Head First Servlets and JSP

Head First Software Development

Head First SQL

Head First Statistics

Head First Web Design

Head First WordPress

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 #4

User Interfaces

In the last Unity Lab you started to build a game, using a prefab to crate GameObject instances that appear at random points in your game's 3D space and fly in circles. This Unity Lab picks up where the last one left off, allowing you to apply what you've learned about interfaces in C# and more.

Your program so far is an interesting visual simulation. The goal of this Unity Lab is to **finish building the game**. It starts off with a score of zero. Billiard balls will start to appear and fly around the screen. When the player clicks on a ball, the score goes up by 1 and the ball disappears. More and more balls appear; once 15 balls are flying around the screen, the game ends. For your game to work, your players need a way to start it and to play again once the game is over, and they'll want to see their score as they click on the balls. So you'll add a **user interface** that displays the score in the corner of the screen, and shows a button to start a new game.

Add a score that goes up when the player clicks a ball

You've got a really interesting simulation. Now it's time to turn it into a game. **Add a new field** to the GameController class to keep track of the score—you can add it just below the OneBallPrefab field:

```
public int Score = 0;
```

Next, **add a method called ClickedOnBall to the GameController class**. This method will get called every time the player clicks on a ball:

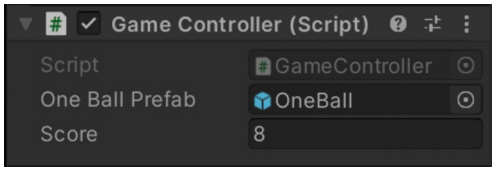
```
public void ClickedOnBall()
{
    Score++;
}
```

Unity makes it really easy for your GameObjects to respond to mouse clicks and other input. If you add a method called OnMouseDown to a script, Unity will call that method any time the GameObject it's attached to is clicked. **Add this method to the OneBallBehaviour class**:

```
void OnMouseDown()
{
    GameController controller = Camera.main.GetComponent<GameController>();
    controller.ClickedOnBall();
    Destroy(gameObject);
}
```

The first line of the OnMouseDown method gets the instance of the GameController class, and the second line calls its ClickedOnBall method, which increments its Score field.

Now run your game. Click on Main Camera in the hierarchy and watch its Game Controller (Script) component in the Inspector. Click on some of the rotating balls—they'll disappear and the Score will go up.



there are no Dumb Questions

Q: Why do we use Instantiate instead of the new keyword?

A: Instantiate and Destroy are ***special methods that are unique to Unity***—you won't see them in your other C# projects. The Instantiate method isn't quite the same thing as the C# `new` keyword, because it's creating a new instance of a prefab, not a class. Unity does create new instances of objects, but it needs to do a lot of other things, like making sure that it's included in the update loop. When a GameObject's script calls Destroy(gameObject) it's telling Unity to destroy itself. The Destroy method tells Unity to destroy a GameObject—but not until after the update loop is complete.

Q: I'm not clear on how the first line of the OnMouseDown method works. What's going on there?

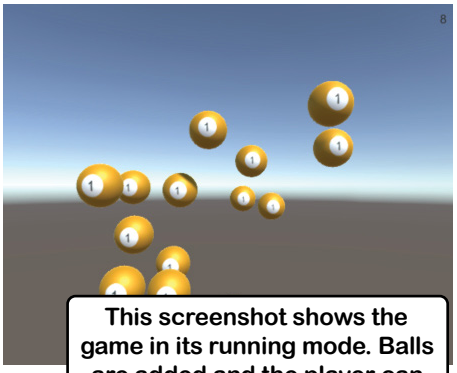
A: Let's break down that statement piece by piece. The first part should be pretty familiar: it declares a variable called `controller` of type GameController, the class that you defined in the script that you attached to the Main Camera. In the second half, we want to call a method on the GameController attached to the Main Camera. So we use Camera.main to get the Main Camera, and GetComponent<GameController>() to get the instance of GameController that we attached to it.

Add two different modes to your game

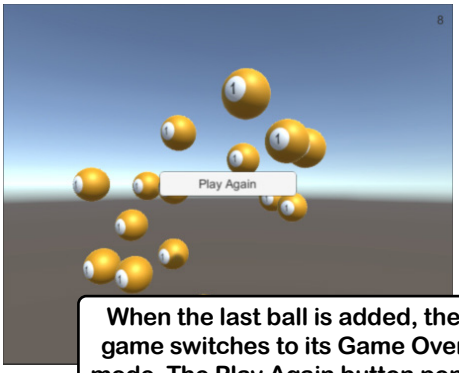
Start up your favorite game. Do you immediately get dropped into the action? Probably not—you're probably looking at a start menu. Some games let you pause the action to look at a map. Many games let you switch between moving the player and working with an inventory, or show an animation while the player is dying that can't be interrupted. These are all examples of **game modes**.

Let's add two different modes to your billiard ball game:

- **Mode #1: The game is running.** Balls are being added to the scene, and clicking on them makes them disappear and the score go up.
- **Mode #2: The game is over.** Balls are no longer getting added to the scene, clicking on them doesn't do anything, and a "Game over" banner is displayed.



This screenshot shows the game in its running mode. Balls are added and the player can click on them to score.



When the last ball is added, the game switches to its Game Over mode. The Play Again button pops up and no more balls get added.

You'll add two modes to your game. You already have the "running" mode, so now you just need to add a "game over" mode.

Here's how you'll add the two game modes to your game:

1. ***Make GameController.AddABall pay attention to the game mode.***

   Your new and improved AddABall method will check if the game is over, and will only instantiate a new OneBall prefab if the game is not over.

2. ***Make OneBallBehaviour.OnMouseDown only work when the game is running.***

   When the game is over, we want the game to stop responding to mouse clicks. The player should just see the balls that were already added continue to circle until the game restarts.

3. ***Make GameController.AddABall end the game when there are too many balls.***

   AddABall also increments its NumberOfBalls counter, so it goes up by 1 every time a ball is added. If the value reaches MaximumBalls, it sets GameOver to true to end the game.

In this lab, you're building this game in parts, and making changes along the way. You can download the code for each part from the book's GitHub repository: https://github.com/head-first-csharp/fourth-edition.

Add game mode to your game

Modify your GameController and OneBallBehaviour classes to **add modes to your game** by using a Boolean field to keep track of whether or not the game is over.

1 ***Make GameController.AddABall pay attention to the game mode.***

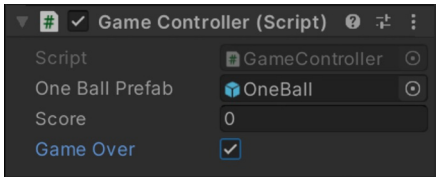
We want the GameController to know what mode the game is in. When we need to keep track of what an object knows, we use fields. Since there are two modes—running and game over—we can use a Boolean field to keep track of the mode. **Add the GameOver field** to your GameController class:

```
public bool GameOver = false;
```

The game should only add new balls to the scene if the game is running. Modify the AddABall method to add an `if` statement that only calls Instantiate if GameOver is not true:

```
public void AddABall()
{
    if (!GameOver)
    {
        Instantiate(OneBallPrefab);
    }
}
```

Now you can test it out. Start your game, then **click on Main Camera** in the Hierarchy window.



Check the Game Over box while the game is running to toggle the GameController's GameOver field. If you check it while the game is running, Unity will reset it when you stop the game.

Set the GameOver field by unchecking the box in the Script component. The game should stop adding balls until you check the box again.

2 ***Make OneBallBehaviour.OnMouseDown only work when the game is running.***

Your OnMouseDown method already calls the GameController's ClickedOnBall method. Now **modify OnMouseDown in OneBallBehaviour** to use the GameController's GameOver field as well:

```
void OnMouseDown()
{
    GameController controller = Camera.main.GetComponent<GameController>();
    if (!controller.GameOver)
    {
        controller.ClickedOnBall();
        Destroy(gameObject);
    }
}
```

Run your game again and test that balls disappear and the score goes up only when the game is not over.

3 ***Make GameController.AddABall end the game when there are too many balls.***

The game needs to keep track of the number of balls in the scene. We'll do this by **adding two fields** to the GameController class to keep track of the current number of balls and the maximum number of balls:

```
public int NumberOfBalls = 0;
public int MaximumBalls = 15;
```

Every time the player clicks on a ball, the ball's OneBallBehaviour script calls GameController.ClickedOnBall to increment (add 1 to) the score. Let's also decrement (subtract 1 from) NumberOfBalls:

```
public void ClickedOnBall()
{
    Score++;
    NumberOfBalls--;
}
```

Now **modify the AddABall** method so that it only adds balls if the game is running, and ends the game if there are too many balls in the scene:

```
public void AddABall()
{
    if (!GameOver)
    {
        Instantiate(OneBallPrefab);
        NumberOfBalls++;
        if (NumberOfBalls >= MaximumBalls)
        {
            GameOver = true;
        }
    }
}
```

The GameOver field is true if the game is over and false if the game is running. The NumberOfBalls field keeps track of the number of balls currently in the scene. Once it hits the MaximumBalls value, the GameController will set GameOver to true.

Now test your game one more time by running it and then clicking on Main Camera in the Hierarchy window. The game should run normally, but as soon as the NumberOfBalls field is equal to the MaximumBalls field, the AddABall method sets its GameOver field to true and ends the game.

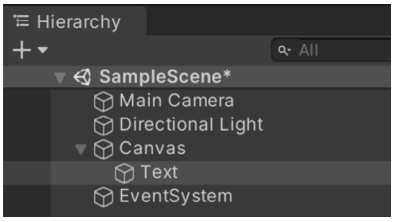
Once that happens, clicking on the balls doesn't do anything because OneBallBehaviour.OnMouseDown checks the GameOver field and only increments the score and destroys the ball if GameOver is false.

Your game needs to keep track of its game mode. Fields are a great way to do that.

Add a UI to your game

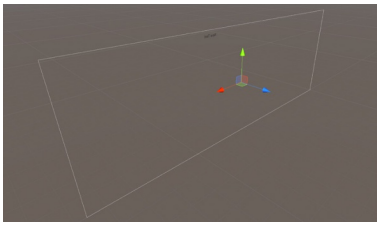
Almost any game you can think of—from Pac Man to Super Mario Brothers to Grand Theft Auto 5 to Minecraft—features a **user interface (or UI)**. Some games, like Pac Man, have a very simple UI that just shows the score, high score, lives left, and current level. Many games feature an intricate UI incorporated into the game's mechanics (like a weapon wheel that lets the player quickly switch between weapons). Let's add a UI to your game.

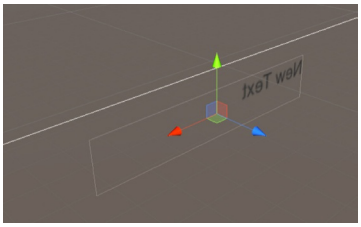
Choose *UI* >> *Text* from the GameObject menu to add a 2D Text GameObject to your game's UI. This adds a Canvas to the Hierarchy, and a Text under that Canvas:



When you added the Text to your scene, Unity automatically added Canvas and Text GameObjects. Click the triangle (▼) next to Canvas to expand or collapse it—the Text GameObject will appear and disappear because it's nested under Canvas.

Double-click on Canvas in the Hierarchy window to focus on it. It's a 2-D rectangle. Click on its Move Gizmo and drag it around the scene. It won't move! The Canvas that was just added will always be displayed, scaled to the size of the screen and in front of everything else in the game.





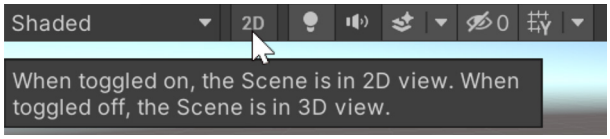
Did you notice an EventSystem in the Hierarchy? Unity automatically added it when you created the UI. It manages mouse, keyboard, and other inputs and sends them back to GameObjects—and it does all of that automatically, so you won't need to work directly with it.

Then double-click on Text to focus on it—the editor will zoom in, but the default text ("New Text") will be backward because the Main Camera is pointing at the back of the Canvas.

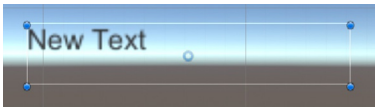
A Canvas is a two-dimensional GameObject that lets you lay out your game's UI. Your game's Canvas will have two GameObjects nested under it: the Text GameObject that you just added will be in the upper-right corner to display the score, and there's a Button GameObject to let the player start a new game.

Use the 2D view to work with the Canvas

The **2D button** at the top of the Scene window toggles 2D view on and off:



Click the 2D view—the editor flips around its view to shows the canvas head-on. **Double-click on Text** in the Hierarchy window to zoom in on it.



Use the mouse wheel to zoom in and out in 2D view.

You can **click the 2D button to switch between 2D and 3D**. Click it again to return to the 3D view.

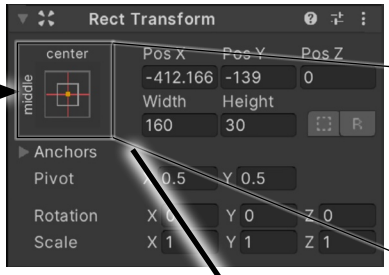
Set up the Text that will display the score in the UI

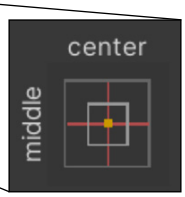
Your game's UI will feature one Text GameObject and one Button. Each of those GameObjects will be **anchored** to a different part of the UI. For example, the Text GameObject that displays the score will show up in the upper-right corner of the screen (no matter how big or small the screen is).

Click on Text in the Hierarchy window to select it, then look at the Rect Transform component. We want the Text in the upper-right corner, so **click the Anchors box** in the Rect Transform panel.

The Text is anchored to a specific point in the 2D Canvas.

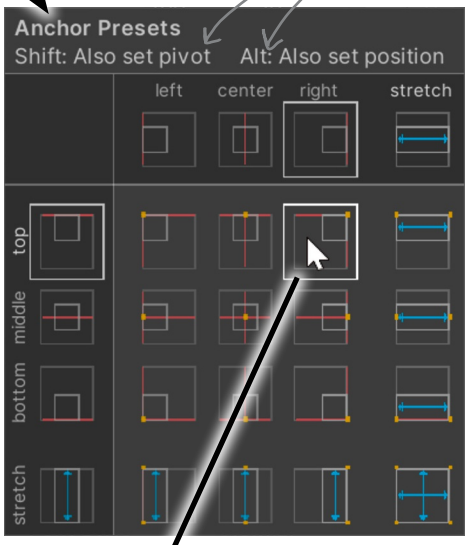
Since the Text only "lives" in the 2D Canvas, it uses a Rect Transform (it has that name because its position is relative to the rectangle that makes up the Canvas). Click on the Anchors box to display the anchor presets.

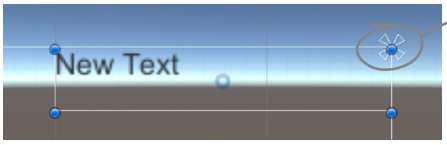




Make sure you hold down both Shift and Alt (Option on a Mac) so you set both the pivot and position.

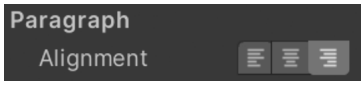
The Anchor Presets window lets you anchor your UI GameObjects to various parts of the Canvas. **Hold down Alt and Shift** (or Option+Shift on a Mac) and **choose the top right anchor preset**. Click the same button you used to bring up the Anchor Presets window. The Text is now in the upper-right corner of the Canvas—double-click on it again to zoom into it.



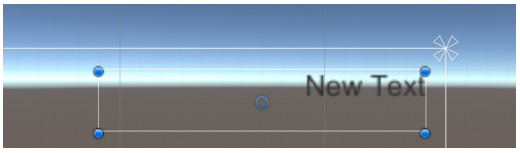


You set the anchor pivot to the top right. The Text's position is the anchor's position relative to the Canvas.

Let's add a little space above and to the right of the Text. Go back to the Rect Transform panel and **set both Pos X and Pos Y to −10** to position the text 10 units to the left and 10 down from the top-right corner. Then **set the Alignment on the Text component to right**, and use the box at the top of the Inspector to **change the GameObject's name to Score**.



Your new Text should now show up in the Hierarchy window with the name Score. It should now be right-aligned, with a small gap between the edge of the Text and the edge of the Canvas.

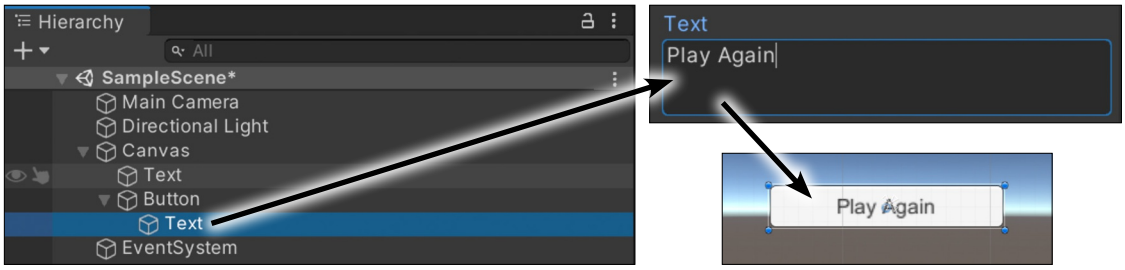


Add a button that calls a method to start the game

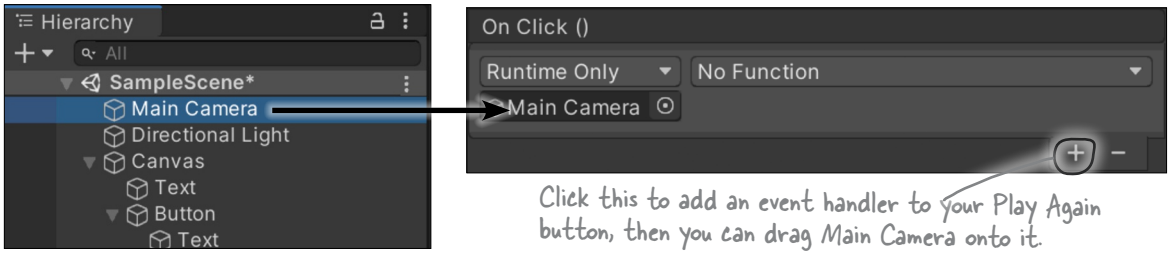
When the game is in its "game over" mode, it will display a button labeled Play Again that calls a method to restart the game. **Add an empty StartGame method** to your GameController class (we'll add its code later):

```
public void StartGame()
{
    // We'll add the code for this method later
}
```

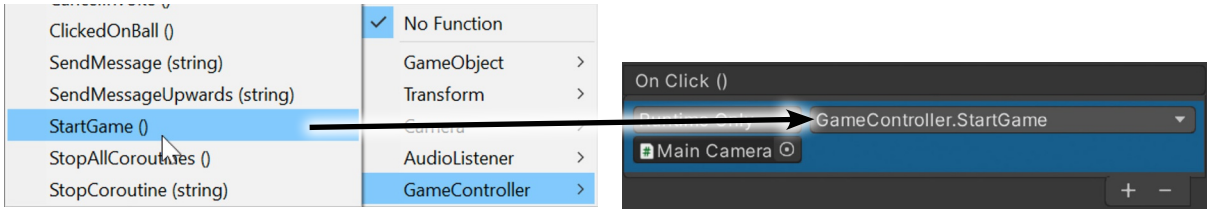
Click on Canvas in the Hierarchy window to focus on it. Then **choose UI >> Button** from the GameObject menu to add a Button. Since you're already focused on the Canvas, the Unity editor will add the new Button and anchor it to the center of the Canvas. Did you notice that Button has a triangle next to it in the Hierarchy? Expand it—there's a TextGameObject nested under it. Click on it and set its text to `Play Again`.



Now that the Button is set up, we just need to make it call the StartGame method on the GameController object attached to the Main Camera. A UI button is ***just a GameObject with a Button component***, and you can use its On Click () box in the Inspector to hook it up to an event handler method. Click the + button at the bottom of the On Click () box to add an event handler, then **drag Main Camera onto the None (Object) box**.



Now the Button knows which GameObject to use for the event handler. Click the No Function dropdown and choose **GameController >> StartGame**. Now when the player presses the button, it will call the StartGame method on the GameController object hooked up to the Main Camera.



Make the Play Again button and Score Text work

Your game's UI will work like this:

- The game starts in the game over mode.
- Clicking the Play Again button starts the game.
- Text in the upper-right corner of the screen displays the current score.

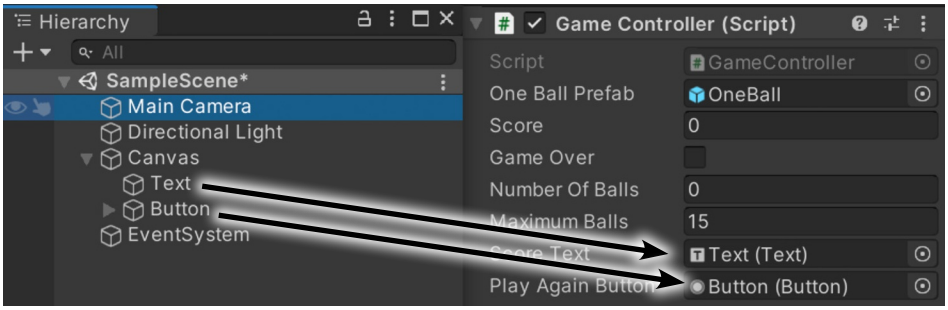
You'll be using the Text and Button classes in your code. They're in the UnityEngine.UI namespace, so **add this `using` statement** to the top of your GameController class:

```
using UnityEngine.UI;
```

Now you can add Text and Button fields to your GameController (just above the OneBallPrefab field):

```
public Text ScoreText;
public Button PlayAgainButton;
```

Click on Main Camera in the Hierarchy window. **Drag the Text GameObject** out of the Hierarchy and ***onto*** the Score Text field in the Script component, **then drag the Button GameObject *onto*** the Play Again Button field.

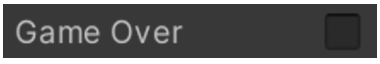


Go back to your GameController code and **set the GameController field's default value to true**:

```
public bool GameOver = true;
```
← Change this from false to true.

Now go back to Unity and check the Script component in the Inspector.

Hold on, something's wrong!



The Unity editor still shows the Game Over checkbox as unchecked—it didn't change the field value. Make sure to check the box so your game starts in the game over mode:

Now the game will start in the game over mode, and the player can click the Play Again button to start playing.

Unity remembers your scripts' field values.

When you wanted to change the GameController.GameOver field from false to true, it wasn't enough to change the code. When you add a Script component to Unity, it keeps track of the field values, and it won't reload the default values unless you reset it from the context menu ( ).

Finish the code for the game

The GameController object attached to the Main Camera keeps track of the score in its Score field. **Add an Update method to the GameController class** to update the Score Text in the UI:

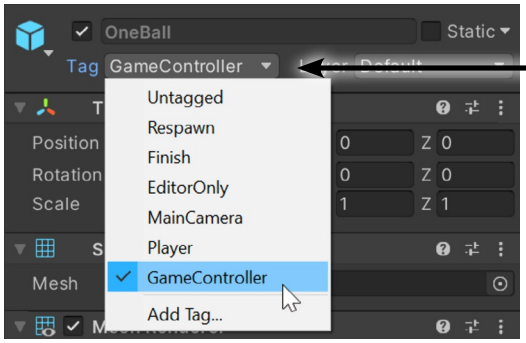
```
void Update()
{
    ScoreText.text = Score.ToString();
}
```

Next, **modify your GameController.AddABall method** to enable the Play Again button when it ends the game:

```
if (NumberOfBalls >= MaximumBalls)
{
    GameOver = true;
    PlayAgainButton.gameObject.SetActive(true);
}
```

Every GameObject has a property called gameObject that lets you manipulate it. You'll use its SetActive method to make the Play Again button visible or invisible.

There's just one more thing to do: get your StartGame method working so that it starts the game. It needs to do a few things: destroy any balls that are currently flying around the scene, disable the Play Again button, reset the score and number of balls, and set the mode to "running." You already know how to do most of those things! You just need to be able to find the balls in order to destroy them. **Click on the OneBall prefab in the Project window and set its tag**:



A tag is a keyword that you can attach to any of your GameObjects that you can use in your code when you need to identify them or find them. When you click on a prefab in the Project window and use this dropdown to assign a tag, that tag will be assigned to every instance of that prefab that you instantiate.

Now you have everything in place to fill in your StartGame method. It uses a `foreach` loop to find and destroy any balls left over from the previous game, hides the button, resets the score and number of balls, and changes the game mode:

```
public void StartGame()
{
    foreach (GameObject ball in GameObject.FindGameObjectsWithTag("GameController"))
    {
        Destroy(ball);
    }
    PlayAgainButton.gameObject.SetActive(false);
    Score = 0;
    NumberOfBalls = 0;
    GameOver = false;
}
```

Now run your game. It starts in "game over" mode. Press the button to start the game. The score goes up each time you click on a ball. As soon as the 15th ball is instantiated, the game ends and the Play Again button appears again.

Here's a Unity coding challenge for you! Each of your GameObjects has a **transform.Translate** method that moves it a distance from its current position. The goal of this exercise is to modify your game so that instead of using transform.RotateAround to circle balls around the Y axis, your OneBallBehaviour script uses transform.Translate to make the balls fly randomly around the scene.

- **Remove** the XRotation, YRotation, and ZRotation fields from OneBallBehaviour. **Replace them with fields** to hold the X, Y, and Z speed called XSpeed, YSpeed, and ZSpeed. They're float fields—no need to set their values.
- **Replace all of the code in the Update method** with this line of code that calls the transform.Translate method:

```
transform.Translate(Time.deltaTime * XSpeed,
                    Time.deltaTime * YSpeed, Time.deltaTime * ZSpeed);
```

  The parameters represent the speed that the ball is traveling along the X, Y, or Z axis. So if XSpeed is 1.75, multiplying it by Time.deltaTime causes ball move along the X axis at a rate of 1.75 units per second.

- **Replace the DegreesPerSecond field** with a field called Multiplier with a value of 0.75**F**—the **F** is important! Use it to update the XSpeed field in the Update method, and **add two similar lines** for the YSpeed and ZSpeed fields:

```
XSpeed += Multiplier - Random.value * Multiplier * 2;
```

  Part of this exercise is to **understand exactly how this line of code works**. Random.value is a static method that returns a random floating-point number between 0 and 1. What is this line of code doing to the XSpeed field?

  .....

  .....

  .....

- Then **add a method called ResetBall** and call it from the Start method. Add this line of code to ResetBall:

```
XSpeed = Multiplier - Random.value * Multiplier * 2;
```

  What does that line of code do?

  Before you start working on the game, figure out what these lines of code do.

  .....

  .....

  Add two more lines just like it to ResetBall that update YSpeed and ZSpeed. Then **move the line of code** that updates transform.position out of the Start method and into the ResetBall method.

- Modify the OneBallBehaviour class to **add a field called TooFar** and set it to 5. Then modify the Update method to check whether the ball went too far. You can check if a ball went too far along the X axis like this:

```
Mathf.Abs(transform.position.x) > TooFar
```

  That checks the *absolute value* of the X position, which means that it will check if transform.position.x is greater than 5F or less than –5F. Here's an `if` statement that checks if the ball went too far along the X, Y, or Z axis:

```
if ((Mathf.Abs(transform.position.x) > TooFar)
     || (Mathf.Abs(transform.position.y) > TooFar)
     || (Mathf.Abs(transform.position.z) > TooFar)) {
```

  Modify your OneBallBehaviour.Update method to use that `if` statement to call ResetBall if the ball went too far.

Here's what the entire OneBallBehaviour class looks like after updating it following the instructions in the exercise. The key to how this game works is that each ball's speed along the X, Y, and Z axes is determined by its current XSpeed, YSpeed, and ZSpeed values. By making small changes to those values, you've made your ball move randomly throughout the scene.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class OneBallBehaviour : MonoBehaviour
{
    public float XSpeed;
    public float YSpeed;
    public float ZSpeed;
    public float Multiplier = 0.75F;
    public float TooFar = 5;

    static int BallCount = 0;
    public int BallNumber;

    // Start is called before the first frame update
    void Start()
    {
        BallCount++;
        BallNumber = BallCount;

        ResetBall();
    }

    // Update is called once per frame
    void Update()
    {
        transform.Translate(Time.deltaTime * XSpeed,
                            Time.deltaTime * YSpeed, Time.deltaTime * ZSpeed);

        XSpeed += Multiplier - Random.value * Multiplier * 2;
        YSpeed += Multiplier - Random.value * Multiplier * 2;
        ZSpeed += Multiplier - Random.value * Multiplier * 2;

        if ((Mathf.Abs(transform.position.x) > TooFar)
             || (Mathf.Abs(transform.position.y) > TooFar)
             || (Mathf.Abs(transform.position.z) > TooFar))
        {
            ResetBall();
        }
    }
```

You added these fields to your OneBallBehaviour class. Don't forget to add the F to 0.75F, otherwise your code won't build.

When the ball is first instantiated, its Start method calls ResetBall to give it a random position and speed.

The Update method first moves the ball, then updates the speed, and finally checks if it went out of bounds. It's OK if you did these things in a different order.

Exercise Solution

```
    void ResetBall()
    {
        XSpeed = Random.value * Multiplier;
        YSpeed = Random.value * Multiplier;
        ZSpeed = Random.value * Multiplier;

        transform.position = new Vector3(3 - Random.value * 6,
            3 - Random.value * 6, 3 - Random.value * 6);
    }

    void OnMouseDown()
    {
        GameController controller = Camera.main.GetComponent<GameController>();
        if (!controller.GameOver)
        {
            controller.ClickedOnBall();
            Destroy(gameObject);
        }
    }
}
```

We reset the ball when it's first instantiated or if it flies out of bounds by giving it a random speed and position. It's OK if you set the position first.

Here are our answers to the questions—did you come up with similar answers?

```
XSpeed += Multiplier - Random.value * Multiplier * 2;
```

By increasing or decreasing the ball's speed along all three axes, we're giving each ball a wobbly random path.

What is this line of code doing to the XSpeed field?

Random.value * Multiplier * 2 finds a random number between 0 and 1.5. Subtracting that from Multiplier gives us a random number between –0.75 and 0.75. Adding that value to XSpeed causes it to either speed up or slow down a small amount for each frame.

```
XSpeed = Multiplier - Random.value * Multiplier * 2;
```

What does that line of code do?

It sets the XSpeed field to a random value between –0.75 and 0.75. This causes some balls to start going forward along the X axis and others to go backward, all at different speeds.

Did you notice that you *didn't have to make any changes to the GameController class*? That's because you didn't make changes to the things that GameController does, like managing the UI or the game mode. If you can make a change by modifying one class but not touching others, that can be a sign that you designed your classes well.

Get creative!

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

- Is the game too easy? Too hard? Try changing the parameters that you pass to InvokeRepeating in your GameController.Start method. Try making them fields. Play around with the MaximumBalls value, too. Small changes in these values can make a big difference in gameplay.
- We gave you texture maps for all of the billiard balls. Try adding different balls that have different behaviors. Use the scale to make some balls bigger or smaller, and change their parameters to make them go faster or slower, or move differently.
- Can you figure out how to make a "shooting star" ball that flies off really quickly in one direction and is worth a lot if the player clicks on it? How about making a "sudden death" 8 ball that immediately ends the game?
- Modify your GameController.ClickedOnBall method to take a score parameter instead of incrementing the Score field and add the value that you pass. Try giving different values to different balls.

If you change fields in the OneBallBehaviour script, don't forget to reset the Script component of the OneBall prefab! Otherwise, it will remember the old values.

The more practice you get writing C# code, the easier it will get. Getting creative with your game is a great opportunity to get some practice!

BULLET POINTS

- Unity games display a **user interface (UI)** with controls and graphics on a flat, two-dimensional plane in front of the game's 3D scene.
- Unity provides a set of **2D UI GameObjects** specifically made for building user interfaces.
- A **Canvas** is a 2D GameObject that lets you lay out your game's UI. UI components like Text and Button are nested under a Canvas GameObject.
- The **2D button** at the top of the Scene window toggles 2D view on and off, which makes it easier to lay out a UI.
- When you add a **Script component** to Unity, it keeps track of the field values, and it won't reload the default values unless 2D reset it from the context menu.
- A **Button** can call any method in a script that's attached to a GameObject.
- You can use the Inspector to **modify field values** in your GameObjects' scripts. If you modify them while the game is running, they'll reset to saved values when it stops.
- The **transform.Translate** method moves a GameObject a distance from its current position.
- A **tag** is a keyword that you can attach to any of your GameObjects that you can use in your code when you need to identify them or find them.
- The **GameObject.FindGameObjectsWithTag method** returns a collection of GameObjects that match a given tag.