O'REILLY®

Fourth Edition

Head First C#

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

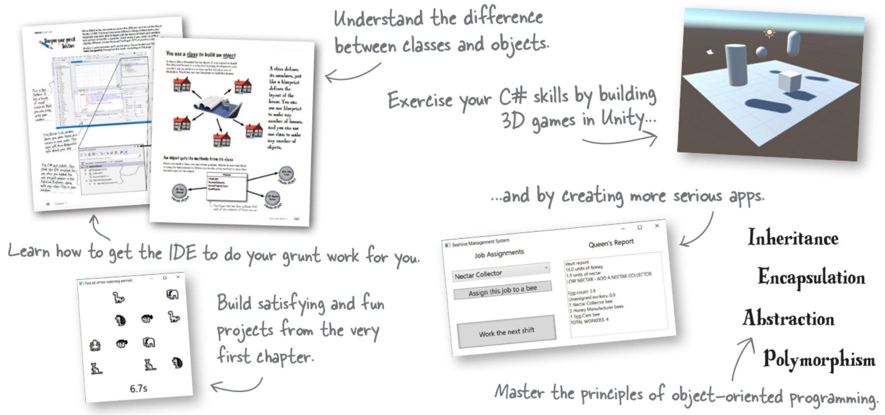
Andrew Stellman
& Jennifer Greene

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!


Understand the difference between classes and objects.
Exercise your C# skills by building 3D games in Unity...
...and by creating more serious apps.
Learn how to get the IDE to do your grunt work for you.
Build satisfying and fun projects from the very first chapter.
Inheritance
Encapsulation
Abstraction
Polymorphism
Master the principles of object-oriented programming.

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.

"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

.NET

US $64.99 **CAN $85.99**
ISBN: 978-1-491-97670-8

O'REILLY®

Head First C#

Fourth Edition

WOULDN'T IT BE DREAMY IF THERE WAS A C# BOOK THAT WAS MORE FUN THAN MEMORIZING A DICTIONARY? IT'S PROBABLY NOTHING BUT A FANTASY...

Andrew Stellman
Jennifer Greene

Beijing • Boston • Farnham • Sebastopol • Tokyo
O'REILLY®

Head First C#

Fourth Edition

by Andrew Stellman and Jennifer Greene

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

ISBN: 978-1-491-97670-8

[LSI] [2020-11-13]

Unity Lab

robots

In the previous Unity Labs, you practiced your C# skills and learned the basics of Unity by building a series of games. In this lab, you'll take the next step by laying the groundwork for building a larger game.

In this lab, you'll build a version of one of the oldest and most established video games in history. It's gone by many names since it was originally developed in the 1970s: Chase, Escape!, Zombies, Daleks. Once the version developed in 1984 was included with Unix distributions, many people came to know it by its most common name: **robots**.

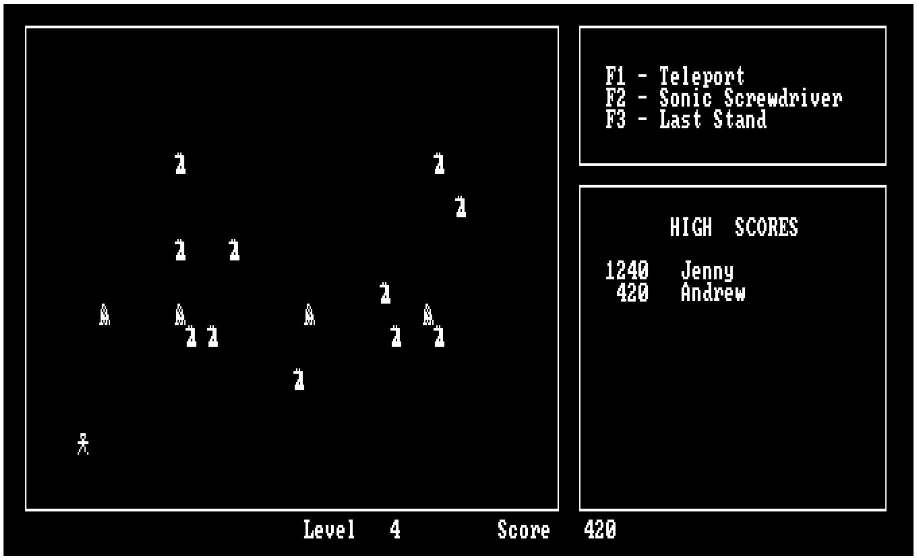
In this Unity Lab, you'll create a Unity version of the classic game. You'll start by importing assets you created in the last Unity Lab, and you'll use them to build the basic gameplay for the game. The game that you build in this Lab will be the foundation for the remaining labs in the book.

The history of **robots**

The game you're about to build has a long history, starting in the 1970s:

- Someone at Dartmouth College whose identity remains unknown to this day—many suspect it was originally written by Mac Oglesby, a prolific game writer at Dartmouth at the time—used a programming language called Dartmouth Basic to write a game called **Chase**.
- Back in the 1970s, a really common way to distribute video games was to print their source code in books and magazines. A version of Chase appeared in the January-February 1976 edition of *Creative Computing* magazine. You can view it on the Internet Archive—it starts on page 76:
  https://archive.org/details/Creative_Computing_v02n01_Jan-Feb1976
- The BASIC source code for the game was published in *More BASIC Computer Games*, a 1979 book edited by David H. Ahl. You can view the code at the Internet Archive:
  https://archive.org/details/More_BASIC_Computer_Games/page/n39/mode/2up
- In 1984, Allan R. Black wrote a Unix version called **robots** and posted it to the Usenet group net.sources.games. Here's the original post: *http://bit.ly/robots-usenet-post*
- A bunch of clones of the game appeared on various computing platforms in the early 1980s. Home computer variations of the game were published under different names, including Daleks, Escape!, and Zombies.
- In 1985, Robert Paauwe wrote a shareware version of the game and called it Daleks. You can **play this game in your browser**: *https://archive.org/details/msdos_Daleks_1985*

One of the authors of this book had David H. Ahl's books growing up, and they were a major influence on his early experience as a programmer!



```
F1 - Teleport
F2 - Sonic Screwdriver
F3 - Last Stand

HIGH  SCORES

1240   Jenny
 420   Andrew

Level   4        Score   420
```

You can play the 1985 Shareware version called Daleks in your browser on the Internet Archive. This game predates W-A-S-D keyboard controls—you use the number pad to move, space to wait, F1 to teleport, F2 to zap nearby enemies, and F3 to stand still and wait for all enemies to move.

- In the almost half-century since Chase was originally coded on a mainframe computer at Dartmouth, many, many versions of this game have been written.

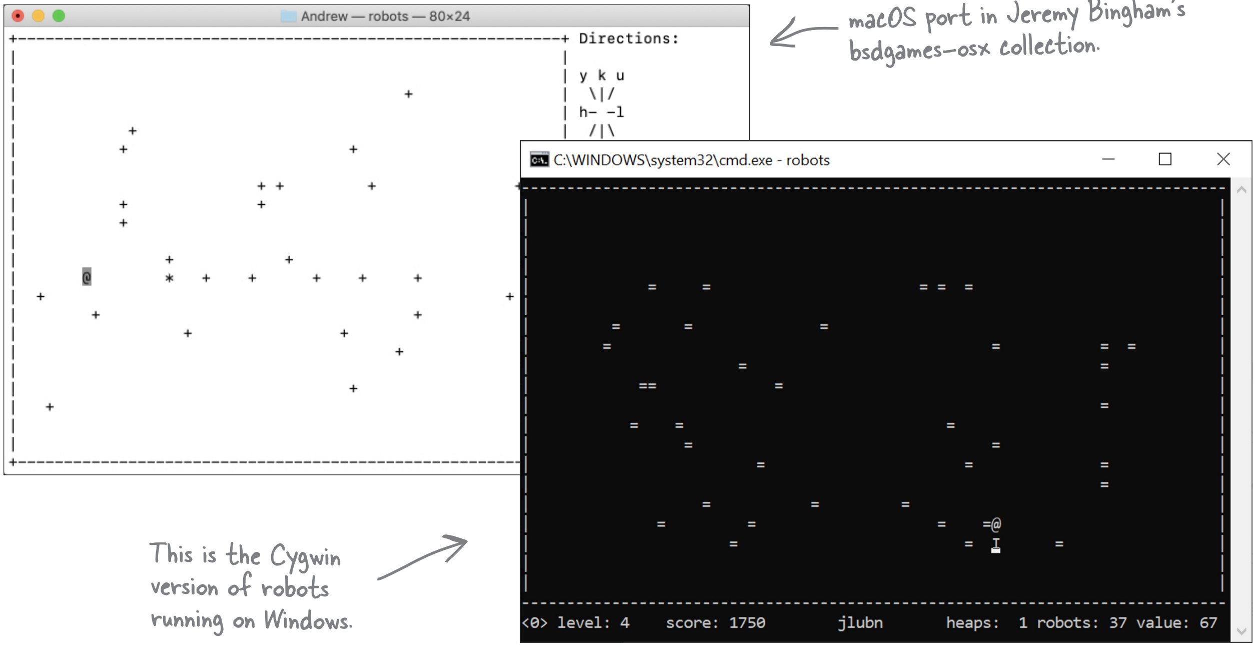
Now <u>you</u> get to join the long tradition and build your own version of this classic game!

The rules of the game

Robots is a **turn-based game**. The goal of the game is to evade the robots and destroy as many as possible by causing them to collide with each other. The version of this game most closely resembles Allan R. Black's robots game, which was included in *BSD Games*, a collection of classic text-based games distributed with many Unix distributions. You can install it on Windows using Cygwin (*https://cygwin.org/*), or on macOS using Jeremy Bingham's Homebrew port of BSD games (*https://github.com/ctdk/bsdgames-osx*).

Here are the rules of the version of the game that you'll build in this Unity Lab:

- ★ Robots is played on a two-dimensional playing area. The player is spawned at a random point in the play area. The robots are spawned at random points as well, with more robots appearing on each consecutive level.
- ★ Each turn, the player can move one space in one of eight directions (up, down, left, right, and four diagonals). The robots then all move one space towards the player.
- ★ If any of the robots hits the player, the game is over. If any two robots hit each other or a dead robot, they die.
- ★ The player can teleport to a random location, but teleporting is dangerous because the player could teleport onto a robot and die.
- ★ The player can wait for a turn without moving, so the robots move towards the player.
- ★ The player gets 10 points for each robot that's destroyed.
- ★ The player can wait for the end, staying still while all robots keep moving. If all of the robots destroy themselves, the player gets an extra 1-point wait bonus.



```
Directions:

 y k u
  \|/
 h- -l
  /|\
```

```
<0> level: 4     score: 1750        jlubn       heaps:  1 robots: 37 value: 67
```

Here's a screenshot of the macOS port in Jeremy Bingham's bsdgames-osx collection.

This is the Cygwin version of robots running on Windows.

Import assets from your Unity Lab 8 project

Have you had a look at the Unity Asset Store? If not, take a few minutes to browse: https://assetstore.unity.com/ – you can find amazing free and paid assets there that you can include in your Unity projects.

One of the most valuable parts of Unity is the ability to import assets into your project. You can get them from the Asset Store. You can also **package up your own assets**. Since you already built a perfectly good scene with a player, NavMesh, materials, and UI, it makes sense to reuse it. Let's do that now.

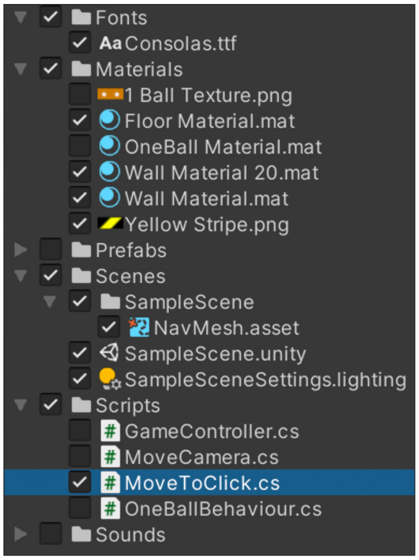
Export the assets from your previous Unity Lab

Open the main project from Unity Lab 8, **select the Assets folder** in the Project window, and choose **Export Package...** from the Assets menu. Unity will display an *Exporting package* window—**check the boxes** next to ***all*** of the assets except for the Sounds and Prefabs folders, ball material texture, and the GameController, MoveCamera, and OneBallBehaviour scripts (but make sure you check the MoveToClick script). Then **click the Export button** to save the assets to a file that ends with the .unitypackage extension. This will create a new file that contains all of the selected assets.

Now you have a Unity package with assets to reuse in your new project.

You can download the asset package from our GitHub page: ***https://github.com/head-first-csharp/fourth-edition***

We named it Unity_Lab_10_robots_starting_assets.unitypackage, and you can find it in the Unity_Labs/Asset_Packages folder.



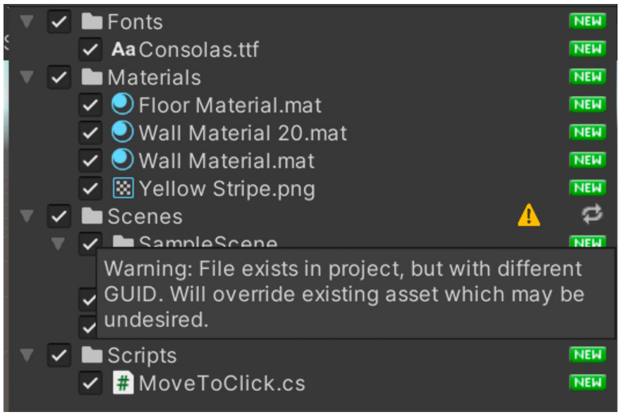
```
▼ ✓ Fonts
    ✓ Aa Consolas.ttf
▼ ✓ Materials
      1 Ball Texture.png
    ✓ Floor Material.mat
      OneBall Material.mat
    ✓ Wall Material 20.mat
    ✓ Wall Material.mat
    ✓ Yellow Stripe.png
▶   Prefabs
▼ ✓ Scenes
  ▼ ✓ SampleScene
        ✓ NavMesh.asset
    ✓ SampleScene.unity
    ✓ SampleSceneSettings.lighting
▼ ✓ Scripts
      GameController.cs
      MoveCamera.cs
    ✓ MoveToClick.cs
      OneBallBehaviour.cs
▶   Sounds
```

Import assets into a new Unity project

Now that you have the assets from the last project neatly packaged up into a file, **create a new Unity project**. Choose **Import Package >> Custom Package** from the Assets menu, and select the .unitypackage file that you exported. Notice that it doesn't give you the option to import individual GameObjects, just the scene, and there's a warning triangle next to the Scenes folder. Hover over it—it's a warning that it will overwrite the scene, SampleScene.

Click All and Import to import all of the assets into your project. Unity will warn you that the open scene has been modified externally and ask if you want to reload it—tell it to reload the scene. You should now see the play area.

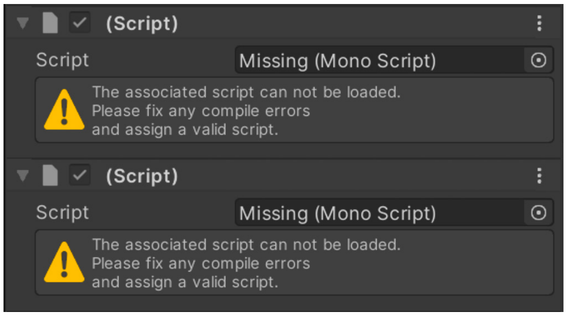
You'll also see a warning at the bottom of the Unity editor window—it's a compiler error on the MoveToClick script telling you that the type or namespace 'GameController' could not be found. That makes sense: you're not dropping any billiard balls in this game, so you didn't include the GameController script in the assets that you exported.



```
▼ ✓ Fonts                                   NEW
    ✓ Aa Consolas.ttf                       NEW
▼ ✓ Materials                               NEW
    ✓ Floor Material.mat                    NEW
    ✓ Wall Material 20.mat                  NEW
    ✓ Wall Material.mat                     NEW
    ✓ Yellow Stripe.png                     NEW
▼ ✓ Scenes                                ⚠
  ▼ ✓ SampleScene                           NEW
    Warning: File exists in project, but with different
    GUID. Will override existing asset which may be
    undesired.
▼ ✓ Scripts                                 NEW
    ✓ MoveToClick.cs                        NEW
```

Add a new GameController script

Select Main Camera in the Hierarchy window. There are two Script components that say "Missing (Mono Script)" – these were scripts attached to Main Camera that you didn't include in your export package.



```
(Script)
Script    Missing (Mono Script)
The associated script can not be loaded.
Please fix any compile errors
and assign a valid script.
(Script)
Script    Missing (Mono Script)
The associated script can not be loaded.
Please fix any compile errors
and assign a valid script.
```

You skipped some scripts (and a few other files) when you exported the asset packages from the Unity Lab 8 project, so when you click on Main Camera you can see the missing scripts in the inspector.

Click on the context menu icon (⋮) in both of those Script components and **choose Remove Component**.

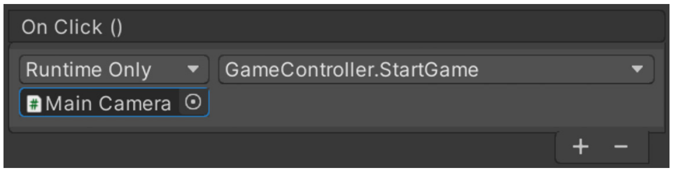
Add a **new script called GameController to the Main Camera** and put it in the Scripts folder. Here's the code:

```
using UnityEngine.UI;

public class GameController : MonoBehaviour {
    public bool GameOver { get; private set; } = true;
    public int Score { get; private set; }
    public Button PlayAgainButton;

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

This is the same StartGame method from the GameController script in the last chapter. Make sure you drag the PlayAgain button onto the Play Again Button field in the Script component and hook its On Click event up to the GameController.StartGame method.

Once the new GameController script is attached to the Main Camera, click the select button (⊙) next to the Play Again Button field in the Script component, choose the Scene tab, and **set the value of the field to Button**. Then expand Canvas in the Hierarchy window and **set the Button's OnClick event** to call MainCamera.GameController.

Now play your game. The Play Again! button appears. When you click the button it disappears, and you can move your player around the play area by clicking on the floor.



```
On Click ()
Runtime Only    GameController.StartGame
Main Camera
```

BRAIN POWER

We started this project by importing the assets from Unity Lab 8. Do you think we made the right choice? Would it have made more sense to start a new project from the ground up? There are no right or wrong-answers—take some time to think about the pros and cons of each approach.

Add a static class to find random points on the NavMesh

Your player and the robot enemies will spawn at random locations on the NavMesh. We'll position them this by setting their transform.position properties in their Start methods. But we don't want two exact copies of same code in their scripts, so we'll create a static class called RandomPointHelper for the GameObjects to call like this:

```
    transform.position = RandomPointHelper.RandomPointOnMesh(transform.position, 100);
```

Add **a script called RandomPointHelper**—it's just a skeleton for now, but you'll fill in the methods soon:

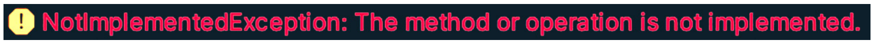
```
using UnityEngine;
using UnityEngine.AI;

public static class RandomPointHelper
{
    public static Vector3 RandomPointOnMesh(
              Vector3 startingPoint, int numberOfSteps)
    {
        throw new System.NotImplementedException();
    }

    private static Vector3 SampleRandomMeshPosition(
              Vector3 startingPoint)
    {
        throw new System.NotImplementedException();
    }
}
```

We need to add two classes at the same time. You'll solve a puzzle to get RandomPointHelper working, and you'll test it using PlayerBehaviour. That means we need a RandomPointHelper class that will compile, even if it doesn't work yet. That's where a class skeleton can be really useful. A skeleton is an outline of a class that has the methods and fields, but none of the code. Instead, you'll have stub code that serves as a placeholder until you write the real code. In this case, we had each of the two methods in RandomPointHelper throw a NotImplementedException.

Now **add a script called PlayerBehaviour** to your Player GameObject and **add this statement** to its Start method to use your new static RandomPointHelper class set its position to a random point on the NavMesh:

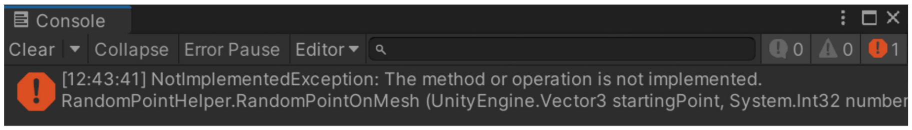
```
transform.position = RandomPointHelper.RandomPointOnMesh(
                                  transform.position, 100) + Vector3.up;
```

Run your game. It still runs, even though it throws an exception! Look for the message at the bottom of the editor:



```
NotImplementedException: The method or operation is not implemented.
```

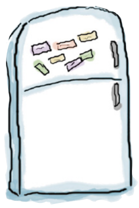
Open the Console window to see more detail about the error. The errors button (!) at the top of the window toggles the error messages, and the number next to it shows how many error messages your game has logged.



```
[12:43:41] NotImplementedException: The method or operation is not implemented.
RandomPointHelper.RandomPointOnMesh (UnityEngine.Vector3 startingPoint, System.Int32 number
```

This is telling you that the error happened in the RandomPointHelper.RandomPointOnMesh method—which makes sense, because it throws an exception. Unlike the other programs you've written, ***your Unity game won't crash when the RandomPointOnMesh method throws an exception***. Instead, it will just stop executing the Start method, but it will continue running the game.

Code Magnets



We printed out the RandomPointHelper class and stuck it on the refrigerator with magnets, but someone slammed the door and **six of the magnets fell off**—they're in a jumble at the bottom! Can you rearrange them to make a working RandomPointHelper class? Pay special attention to the snippet with this code:

```
NavMesh.SamplePosition(randomPoint, out hit, 6f, NavMesh.AllAreas)
```

That's a static method provided by Unity that finds a random point on the NavMesh within 1 unit of `randomPoint`. Can you get this code working ***even if you're not 100% clear on how it works yet***?

Don't just do this Code Magnets puzzle on paper! Try filling in the skeleton RandomPointHelper class with the working code as you solve the puzzle. You've written enough code by now that seeing it in the IDE will help you make sense of it. You can test your solution by running the code, since your PlayerBehaviour.Start method calls RandomPointHelper to set the player's starting position.

```
using UnityEngine;
using UnityEngine.AI;
```

```
public static class RandomPointHelper
{
```

```
public static Vector3 RandomPointOnMesh(
         Vector3 startingPoint, int numberOfSteps)
{
```

```
    randomPointOnMesh = SampleRandomMeshPosition(randomPointOnMesh);
}
```

```
private static Vector3 SampleRandomMeshPosition(Vector3 startingPoint)
{
```

```
if (NavMesh.SamplePosition(randomPoint, out hit, 6f, NavMesh.AllAreas))
{
```

```
Vector3 randomPointOnMesh = startingPoint;
for (int i = 0; i < numberOfSteps; i++)
{
```

```
}
```

```
}
```

```
    return randomPointOnMesh;
}
```

```
Vector3 newPoint = startingPoint;
```

```
Vector3 randomPoint = startingPoint + Random.insideUnitSphere * 6f;
NavMeshHit hit;
```

```
return newPoint;
```

```
    newPoint = hit.position;
}
```

Code Magnets Solution

Here's the working RandomPointHelper class with the jumbled code snippets arranged so the class works. If you make the RandomPointHelper class in your project match this code, then when you start your game the PlayerBehaviour.Start method will call its RandomPointOnMesh method to position the player at a different random point every time you start the game.

```
using UnityEngine;
using UnityEngine.AI;

public static class RandomPointHelper
{

    public static Vector3 RandomPointOnMesh(
                Vector3 startingPoint, int numberOfSteps)
    {

        Vector3 randomPointOnMesh = startingPoint;
        for (int i = 0; i < numberOfSteps; i++)
        {

            randomPointOnMesh = SampleRandomMeshPosition(randomPointOnMesh);
        }

        return randomPointOnMesh;
    }

    private static Vector3 SampleRandomMeshPosition(Vector3 startingPoint)
    {

        Vector3 newPoint = startingPoint;

        Vector3 randomPoint = startingPoint + Random.insideUnitSphere * 6f;
        NavMeshHit hit;

        if (NavMesh.SamplePosition(randomPoint, out hit, 6f, NavMesh.AllAreas))
        {

            newPoint = hit.position;
        }

        return newPoint;

    }

}
```

Once you got the code in the right order, theStart method sets the player's position to a random point on the NavMesh. SamplePosition function isn't the only code you haven't seen before. The Random. insideUnitSphere property is also new… but you didn't need to be familiar with either of them to solve the puzzle.

BRAIN POWER

Can you think of why it makes sense for your game to keep running when the Start method throws an exception? You've probably run into glitches while playing games—is there a connection between game glitches and this kind of exception handling?

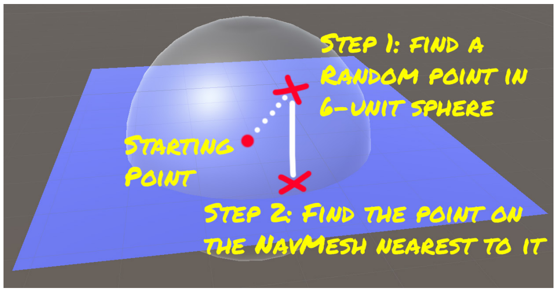
Random Walks Up Close

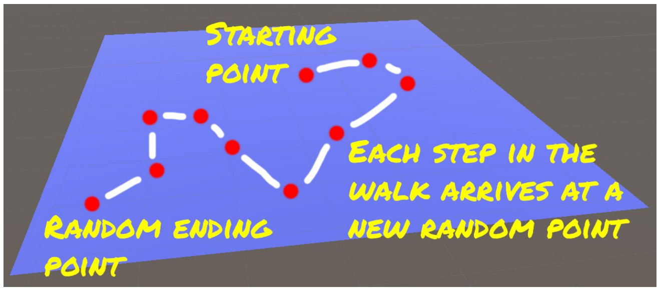
The **static NavMesh object** is a useful tool in Unity's navigation system. We used its **NavMesh.SamplePosition method**, which takes a starting point and finds the closest position to it on the scene's NavMesh. So if you have a Vector3 variable called `point` with a location, this code will find the nearest point on the scene's NavMesh within 6 units of that location:

```
bool success = NavMesh.SamplePosition(point, out hit, 6.0f, NavMesh.AllAreas)
```

If there's a position within 1.0f units of point, NavMesh.SamplePosition returns true; if not, it returns false. It uses the out keyword to return the result as a NavMeshHit type. (This is just like how the ScreenPointToRay method returned its result—if you need a reminder of how that works, flip back to the last Unity tutorial.)

Unity gives us another really useful tool**. The Random.insideUnitSphere property** returns a Vector3 at a random location somewhere inside the 1-unit sphere centered at (0, 0, 0). We can combine it with NavMesh.SamplePosition to find a random point on the NavMesh by doing a **random walk**. Here's how that works:


Step 1: find a random point in 6-unit sphere
Starting point
Step 2: Find the point on the NavMesh nearest to it



Step 1: The walk starts out at a point somewhere on the NavMesh called startingPoint. We do some vector arithmetic that uses Random.insideUnitSphere to find a point in our scene that is within 6 units in any direction of startingPoint:

```
    Vector3 randomPoint = startingPoint + Random.insideUnitSphere * 6f;
```

Did you notice two different vector arithmetic operations? First we multiplied Random.insideUnitSphere by 6, which gives us a random location inside a 6-unit sphere centered at (0, 0, 0). Then we added it to startingPoint, which means we've not got a new Vector3 location called randomPoint that's within 6 units of the starting point. That point could be anywhere in the 6-unit sphere centered on startingPoint, which means it's very likely above or below the NavMesh.

Step 2: Now we can use NavMesh.SamplePosition to find the nearest point on the NavMesh:

```
    NavMeshHit hit;
    bool result = NavMesh.SamplePosition(randomPoint, out hit, 10f, NavMesh.AllAreas);
```

If result is true, then hit.position will contain the nearest point on the mesh to the random point. Now you have a random point on the NavMesh that's near the starting point.

Repeat: If we want to find a point that's farther away, we'll just repeat this over and over, doing a series of small random steps. This is called a **random walk**—the more steps in the walk, the farther away the end point can get.

You can use Debug.DrawRay to see how this works. Create a new script and **add this code to its Update method**:

```
    Vector3 randomPoint = RandomPointHelper.RandomPointOnMesh(Vector3.zero, 100);
    Debug.DrawRay(randomPoint, Vector3.up, Color.yellow, 1.0f);
```

Drag it onto your Main Camera, then run the game and switch to the Scene view to see rays appearing all over the floor.

Add an enemy that follows the player when it moves

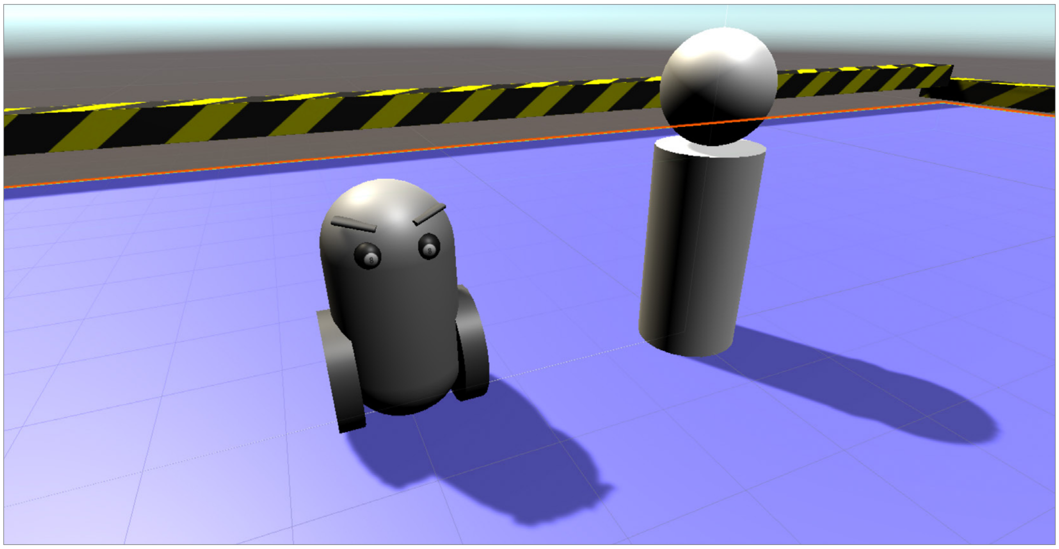
Each enemy robot will be a are the GameObjects for the play area—notice that we included the Main Camera in this list so it starts out at an angle that shows the whole floor, and we also included a second Directional Light so you can see your robot more clearly. **Create these GameObjects**:

Name	Type	Position	Rotation	Scale
Main Camera	Camera	(0, 15, -9)	(65, 0, 0)	(1, 1, 1)
Directional Light 2	Directional Light	(90, 230, 180)	(120, -30, 0)	(1, 1, 1)
▼ Robot	Capsule	(3, 1, 0)	(0, 0, 0)	(1, 1, 1)
Right Wheel	Cylinder	(-0.5, -0.5, 0)	(0, 0, 90)	(1, 0.1, 1)
Left Wheel	Cylinder	(0.5, -0.5, 0)	(0, 0, 90)	(1, 0.1, 1)
Right Eye	Sphere	(-0.2, 0.6, 0.4)	(0, 0, 0)	(0.2, 0.2, 0.2)
Left Eye	Sphere	(0.2, 0.6, 0.4)	(0, 0, 0)	(0.2, 0.2, 0.2)
Right Eyebrow	Cylinder	(-0.25, 0.8, 0.3)	(-10, -30, -100)	(0.05, 0.15, 0.1)
Left Eyebrow	Cylinder	(0.25, 0.8, 0.3)	(10, 30, 100)	(0.05, 0.15, 0.1)

Now let's add materials to make it look a little more robot-ish:

- ★ Create a material called *Robot Material* with albedo color `959595` and apply it to Robot.
- ★ Create a material called *Wheel Material* with albedo color `3C3C3C` and apply it to Right Wheel, Left Wheel, Right Eyebrow, and Left Eyebrow.
- ★ Use the same `8 Ball Texture.png` that you used in the previous Unity Labs to create a material called *8 Ball Material* and apply it to Right Eye and Left Eye.

Now you've got a deadly-looking enemy for your game.



Exercise

The robots in Unix Robots only move when the player moves, so your job is to make the robot move only when the player is moving. In this exercise, you'll build a behavior script for the Robot GameObject that detects when the player is moving and makes the robot chase the player.

Add an IsMoving property to your PlayerBehaviour script

You're going to add a bool property called IsMoving to PlayerBehaviour that only has a get accessor, but no setter. The robot will use this property to determine whether or not to move towards the player.

You can figure out if your player is moving by adding a **FixedUpdate method** that saves the current position in a private Vector3 field called lastUpdatePosition, and adding a property to see if the position has changed. Unity calls each the FixedUpdate method *exactly 50 times a second*. So if the position hasn't changed (or changed very little) in consecutive FixedUpdate calls, the player's stopped moving. You can use the **Vector3.Distance method** to check the distance between two vectors. If they're under 0.001 units apart, then that's close enough to zero for our purposes:

```
Vector3.Distance(transform.position, lastUpdatePosition) > 0.001f
```

Let's make it easier for you to do the exercise: **add a Debug.Log statement** to the first line of FixedUpdate that logs a line to the console with `true` if the player is moving, or `false` if the player is still.

MINI Sharpen your pencil

Why does the Debug.Log statement need to be on the first line of the FixedUpdate method? Why can't it be at the end?

....

....

....

Add a RobotBehaviour script that checks Player.IsMoving and moves the robot only if the player is moving

Add a NavMesh Agent to the Robot GameObject. Then **add a RobotBehaviour script** with two fields:

```
public PlayerBehaviour Player;
private NavMeshAgent agent;
```

Set the **`agent`** field in the Awake method (like in your MoveToClick script), and use the select button (◉) to set the Player field. In theUpdate method, use the PlayerBehaviour.IsMoving property to check if the player is moving. If it's true, make the robot chase the player by setting its NavMesh Agent's destination to Player.transform.position. If the player is not moving, make the robot stop by setting its NavMesh Agent's destination to the robot's transform.position. Finally, adjust the NavMesh Agent fields in both the player and robot to make them move at a speed that you like.

The FixedUpdate method

You've used the Update method in the previous labs to make changes to your GameObjects after every frame. The frame rate that your game runs at depends on the speed of the hardware it's running on, so your Update methods used Time.deltaTime to take the time elapsed since the previous frame into account. But sometimes you really want a method that is guaranteed to run at a fixed interval—and that's what the FixedUpdate method does. It runs exactly 50 times per second on any hardware, which comes in handy if you need an operation that absolutely must happen at exactly the same interval each time it runs.

Exercise Solution

The robots in Unix Robots only move when the player moves. So you had to add an IsMoving property to PlayerBehaviour that's true if the player is moving, and have RobotBehaviour use that property and only move towards the player if it's true.

Here's the updated PlayerBehaviour class with the FixedUpdate method and IsMoving property:

```
public class PlayerBehaviour : MonoBehaviour
{
    private Vector3 lastUpdatePosition;

    void Start()
    {
        transform.position = RandomPointHelper.RandomPointOnMesh(
                                          transform.position, 100) + Vector3.up;
    }

    private void FixedUpdate()
    {
        Debug.Log(IsMoving);
        lastUpdatePosition = transform.position;
    }

    public bool IsMoving =>
        (Vector3.Distance(transform.position, lastUpdatePosition) > 0.01f);
}
```

The IsMoving property returns true of the player is moving by checking if its current position changed less than 0.01 units in the last 1/50th of a second. It does that by comparing the current position with the one that was set in the last FixedUpdate call.

This is an example of using the => lambda operator to create a property, like we learned in Chapter 9.

And here's the RobotBehaviour class that makes it follow the player, but only when the player is moving:

```
using UnityEngine.AI;

public class RobotBehaviour : MonoBehaviour
{
    public PlayerBehaviour Player;
    private NavMeshAgent agent;

    void Awake()
    {
        agent = GetComponent<NavMeshAgent>();
    }

    void Start()
    {
        transform.position = RandomPointHelper.RandomPointOnMesh(
                                           transform.position, 100) + Vector3.up;
    }

    void Update()
    {
        if (Player.IsMoving) agent.SetDestination(Player.transform.position);
        else agent.SetDestination(transform.position);
    }
}
```

Did you notice that the type of this field is PlayerBehaviour, and not GameObject? When you drag the Player GameObject Unity is smart enough to look for the PlayerBehaviour script component on the Player GameObject and this field so it contains a reference to that GameObject's instance of PlayerBehaviour.

there are no Dumb Questions

Q: I know we've seen vector arithmetic several times, but I'm still not all that comfortable with it. Can you explain why you're adding Vector3.up to the random point on the mesh?

A: Absolutely. The phrase "vector math" might sound intimidating, but adding two vectors just means adding their X values, adding the Y values, and adding the Z values to find a new vector. So (3, 5, 9) + (12, 1, 2) = (3 + 12, 5 + 1, 9 +2) = (15, 6, 11). Vector3.up is the same as new Vector3(0, 1, 0) because it's exactly one unit up from position (0, 0, 0) (which is also called Vector3.zero). When you added Vector3.up to the random point on the NavMesh, you added (0, 1, 0) to it—which means you just added 1 to its Y value.

Q: Okay, I understand *what* adding Vector3.up to the random point did. But *why* did we do it?

A: The random point on the NavMesh is on the floor, with its Y coordinate equal to zero. But your Player GameObject is a cylinder, and since the scale of the cylinder isn't changed the center of the GameObject is the middle of the cylinder. So if you position Player at (0, 0, 0) the bottom half of the player will actually be sticking out below the floor. So you had to move it up along the Y-axis by 1 unit to make the bottom of the player just touch the floor.

Q: I get all of that in theory. Can I see some code?

A: Let's write some code to find out! We'll start with an arbitrary point: (19.2, 3.6, 7.1) – and add Vector3.up to it. Add these lines to your GameController.Start method to calculate the point exactly one unit up from it:

```
Vector3 v = new Vector3(19.2f, 3.6f, 7.1f);
Vector3 oneAbove = v + Vector3.up;
Debug.Log(oneAbove);
```

Now run your game and open the Console window. You'll see the answer: (19.2, 4.6, 7.1). This is an easy way to experiment with vector arithmetic.

Q: Why do we need to do a random walk? Can't we just do Random.insideUnitSphere * 1000f once to get a random point inside a sphere that contains the entire play area, and then find the nearest point on the NavMesh?

A: You could use Random.insideUnitSphere with a very large radius to find a random point, but there's a cost to it. Open up the Unity scripting reference and search for NavMesh.SamplePosition. You'll recognize some of the code on that page because the RandomPointHelper.RandomPointOnMesh function is based on it. The manual page also includes this text:

The function can get quite expensive if the search radius is really big. A good starting point for the maxDistance is 2 times the agent height.

In this context, "expensive" means that if you pass a small maxDistance argument to the NavMesh.SamplePosition method it works quickly, but as you pass larger and larger maxDistance values to the method, it gets much, much slower. So if you need to find a random point far away on the NavMesh, it's going to be a lot faster to do a random walk that consists of many small hops than try to do a single large hop.

Vector arithmetic is an important tool—luckily, it's not hard to understand the basics. The Unity manual has a really useful page to help you understand vector arithmetic. Open the Unity manual and search for Understanding Vector Arithmetic – take a few minutes and read through the page. Taking the time to understand how vector arithmetic work snow will make things much easier later on.

MINI Sharpen your pencil Solution

Why does the Debug.Log statement need to be on the first line of the FixedUpdate method? Why can't it be at the end?

The Debug.Log statement needs to come before the lastUpdatePosition field is set. If it's moved to the end of the FixedUpdate method, it will always call IsMoving when lastUpdatePosition is equal to transform.position – which will cause it to always print false.

Turn Robot into a prefab and spawn multiple enemies

Make your **Robot GameObject into a prefab**, the same way you did in your other Unity projects. First, create a folder called Prefabs in the Project window, then **drag Robot into the Prefabs folder and rename it RobotPrefab**. Once you do that, Robot will turn blue in the Hierarchy window. **Delete it** from the scene.

Now you can **modify your GameController class** to add a public GameObject field called RobotPrefab, and have it spawn enemies at random points on the NavMesh by calling a new ResetLevel method from its StartGame method:

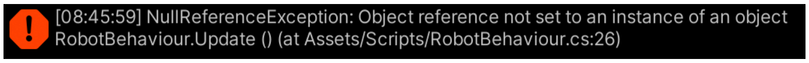
```
public GameObject RobotPrefab;

void Start() {
    ResetLevel(5);
 }

public void StartGame()
{
    PlayAgainButton.gameObject.SetActive(false);
    Score = 0;
    GameOver = false;
    ResetLevel(5);
}

private void ResetLevel(int numberOfRobots)
{
    for (int i = 0; i < numberOfRobots; i++)
    {
        Instantiate(RobotPrefab);
    }
}
```

The game starts in "game over" mode so the player can decide when to start playing, but we still want them to see robots.

The ResetLevel method instantiates robots—but it doesn't delete any existing ones in the scene. So the Start method adds 5 robots, then when you press the button the StartGame method adds 5 more robots. How do you think you'll fix that so the game starts with 5 robots?

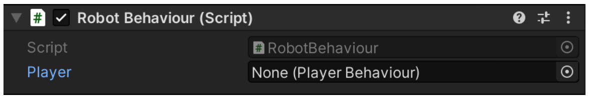
Use the select button (◉) to set the Robot field in GameController to your new RobotPrefab, then run the game. It looks like it worked! But hold on—when you move the player, the robots aren't chasing it anymore.

What happened? Let's use Unity's debugging tools to find out. **Open the Console window**—you'll see an exception:



```
[08:45:59] NullReferenceException: Object reference not set to an instance of an object
RobotBehaviour.Update () (at Assets/Scripts/RobotBehaviour.cs:26)
```

Double-click on one of the exceptions in the Console window to jump to the line of code that caused the problem—it's the line in RobotBehaviour.cs that checks the Player.IsMoving property. So let's check Player. Click on Robot in the Prefabs folder to view the prefab in the Inspector. Then click the Open Prefab button in the Inspector.



Robot Behaviour (Script)
Script: RobotBehaviour
Player: None (Player Behaviour)

The Player field in the script component was reset to None. But hold on… something's wrong! ***The Unity editor won't let you set the Player field in the prefab to reference the Player GameObject in the scene.*** When you click the select button next to the Player field, it doesn't show RobotPrefab in its *Select PlayerBehaviour* window, and Unity won't let you drag Player onto the field!

We'll need to find another way to set the Player field.

BRAIN POWER

Why do you think Unity won't let you set a field in a prefab to reference a GameObject in the scene?

Make the robots find the Player GameObjet using its tag

When you did the project in the last Unity Lab, you tagged added the "Player" tag to the Player GameObject. So when you exported that asset and imported it into your current project, the Player GameObject still had its tag. Let's take advantage of that tag to find the Player. Start by **making the RobotBehaviour.Player field private**:

```
        private PlayerBehaviour Player;
```

We always use camelCase and for private fields, so we need to fix the name of the **P**layer field so it starts with a lowercase letter… but that field is used elsewhere in the code! How can we make sure to change every instance of it?

Renaming is the simplest way to **refactor** your code

Throughout the book, we've been **refactoring**, or changing the structure of your code without altering its behavior. Great developers refactor their code all the time, because they're constantly looking for ways to improve it. One of the easiest ways to improve your code is to **rename your variables, methods, and properties**.

We've seen how the Visual Studio IDE makes it really easy for us to rename variables. Here's a quick recap:

★ When you click on a field or variable, the IDE highlights other places where it's used in the code.

★ You can rename a variable, field or method. Just **right-click on the field** (or method, variable, etc) that you want to rename. Then **choose Rename** from the right-mouse menu.

★ The IDE will keep the field and all of the other occurrences of it highlighted, but the highlight color will change. Now you can **start typing the new field name**: `playerBehvaiour`.

★ As you type the new name, all of the occurrences ***are updated at the same time***. This makes it really easy to tell that you're making the change that you intend to make.

So go ahead and **rename the Player field to playerBehaviour**. The IDE changes the field name, and also the two occurrences of it in the first line of the Update method.

Now you can use the GameObject.FindObjectWithTag method to set the field in the Awake method:

```
void Awake()
{

    agent = GetComponent<NavMeshAgent>();
    playerBehaviour = GameObject.FindGameObjectWithTag("Player")
                                .GetComponent<PlayerBehaviour>();
}
```

This is really similar to the FindGameObjectsWithTag method thatt you used in the last Unity Lab, except it just returns a single GameObject.

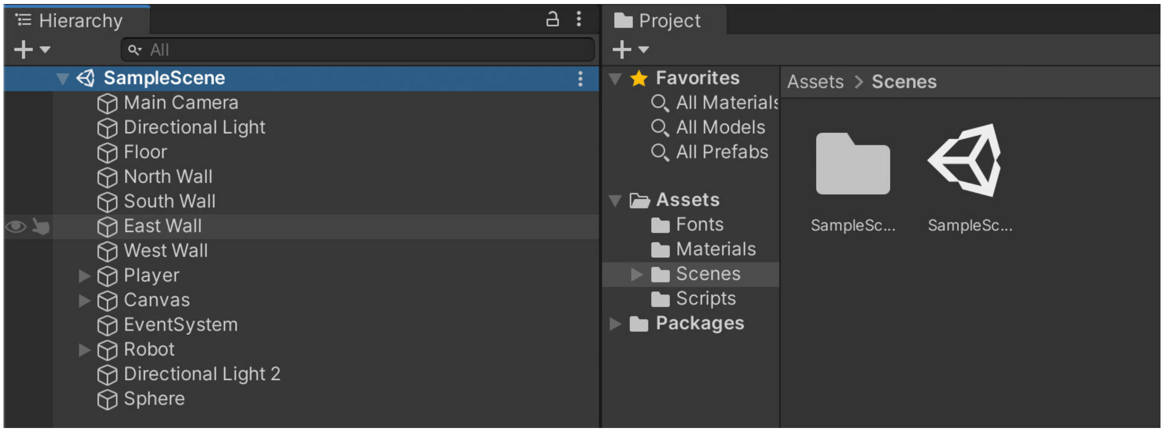
Run your game again. Now all of the robots will chase the player GameObject whenever it's moving.

The IDE makes it really easy to rename a variable, class member, even a class. Just right-click on it, choose Rename, and start typing. The IDE will automatically rename it everywhere in your code.

Behind the Scenes

Scenes

A **scene** contains the game environment and UI for your game—so it shouldn't surprise you that you've been working with scenes in all of the Unity projects so far. When you create a new Unity project, it creates a folder called Scenes with a scene called SampleScene. Go ahead and look in your project right now—you'll see SampleScene in the Scenes folder, and at the top of the Hierarchy window:

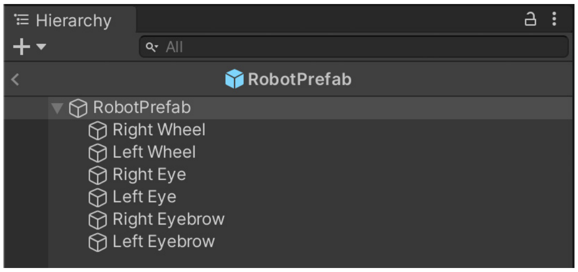


When a Unity game has multiple levels, each of those levels typically lives in its own scene. A scene contains **instances of GameObjects**, including any cameras (like Main Camera your scene) and lights (including Directional Light). When you view a scene in the Hierarchy window, it shows you all of those instances, and when you click on them you can views details about each instance in the Inspector.

Unity instantiates the GameObjects in your scene automatically, doing exactly the same thing behind the scenes that it does when you call the Instantiate method from your code. The GameObjects in your scene are instantiated along with the scene, but prefabs are instantiated separately.

So why Unity wouldn't let you drag a GameObject from the Hierarchy onto your Prefab?

Because your game can contain more than one scene, and the same prefab can be instantiated in different scenes. Unity won't allow you to drag a GameObject from a scene onto a prefab because that prefab might be instantiated in a different scene where that object doesn't exist.



This is also why the Hierarchy window changes when you double-click on a prefab in the Project window, and you have to click the back button (<) to get back to the scene hierarchy. The prefab has its own hierarchy that exists outside of any scene. Unity is preventing you from dragging objects instantiated in a scene into a prefab that might be instantiated in a different scene, so they wouldn't exist when the prefab needs them.

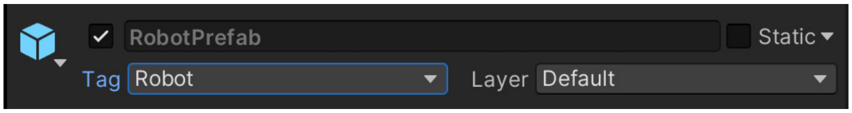
Add collision detection

Now that you've got the robots following the player, it's time to do a little collision detection. When two robots collide with each other, they should both die. When a robot collides with the player, the game should end.

You need a Rigidbody in order to detect collisions, so **add a Rigidbody component to Player**. However, the GameObject is not going to be responding to forces from the physics engine, because we want the NavMesh Agent controlling the player's movement. So **uncheck Use Gravity and check Is Kinematic**, which will allow it to use the physics engine to detect collisions but not rebound from those collisions or respond to other forces.

Then **add a RigidBody to RobotPrefab** – make sure it's kinematic but does not respond to gravity.

Now **add a new tag called Robot**, and tag the RobotPrefab with it.

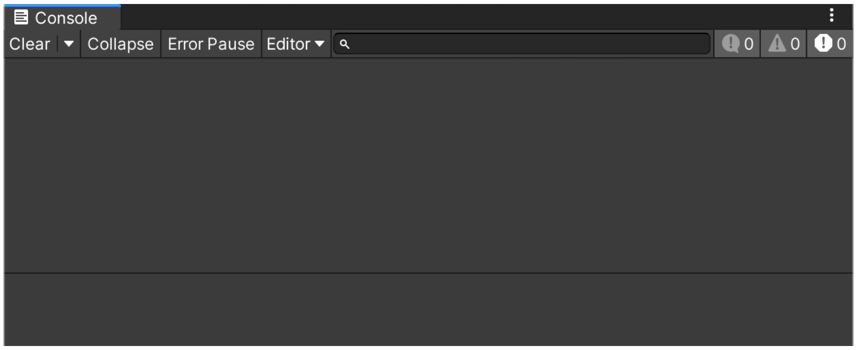

RobotPrefab
Static
Tag Robot
Layer Default

Now **add an OnCollisionEnter method** to the RobotBehaviour class, just like you did in the last Unity Lab:

```
void OnCollisionEnter(Collision other)
{
    if (other.gameObject.CompareTag("Player") || other.gameObject.CompareTag("Robot"))
    {
         Debug.Log($"{gameObject.tag} collided with {other.gameObject.tag}");
    }
}
```

And if you still have **any other Debug.Log statements** in PlayerBehaviour or RobotBehaviour, delete them.

Now that you've code that runs when your robot collides with something, **run your game and open up the Console window**. Make the player run into robots, and move the player around so they run into themselves. Once the player has collided with a few robots, have a look at the Console window:



If you keep playing your game and making many robots collide with the player, you might see a small number of log messages, but definitely not the many, many collisions that we would expect to see.

Were you expecting something different, like a bunch of messages in the Console window? It seems that Debug.Log is never getting called.

Why isn't the OnCollisionEnter method being called when a robot touches the player?

Use triggers to fix the collision detection problem

Let's have a quick look at the **NavMesh Agent page in the Unity manual**. Open the Unity Manual and search for NavMesh Agent. It's worth reading the whole page, but here's the part that's relevant to our current problem:

> Agents reason about the game world using the NavMesh and they know how to avoid each other as well as other moving obstacles.

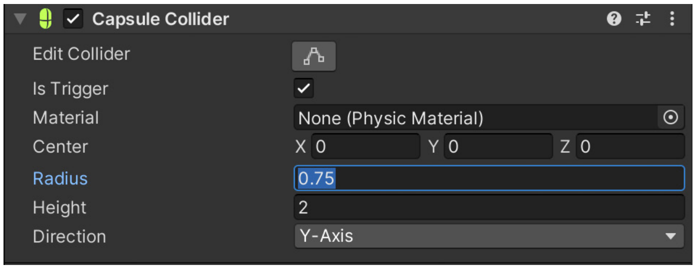
So of course there are no collisions! The NavMesh Agents are avoiding each other, so they'll get very close but they never collide. That means collision detection won't work for this game. What we really want to do is ***detect when the robots or the player are very close to each other.***

This is where **triggers** come in handy. A trigger lets you use Unity's physics engine to detect when one object enters the space of another object—without creating a collision. A trigger is the shape of a solid object but only detects when another object passes through it. Luckily, triggers are just as easy to use as collisions. Open RobotPrefab again and look for its Capsule Collider component.

You need RigidBody components to do collision detection.

*Collision and trigger detection rely on the physics engine, so make sure that you **add a RigidBody component** to both GameObjects involved in the collision, or your OnTriggerEnter method won't ever get called.*

You may not have noticed it at the time, but the Capsule Collider was created automatically when you first created the Capsule for the robot's body. **Check the Is Trigger box** to turn it into a trigger. Then click on the Radius label and drag your mouse cursor up and down. You'll see the yellow outlines of a capsule grow and shrink as the number goes up and down—this is the capsule-spaced shape that will detect other objects. **Set the radius to 0.75** so that the radius for triggers it's just outside of the robot's wheels.



Capsule Collider
Edit Collider
Is Trigger ✓
Material: None (Physic Material)
Center: X 0 Y 0 Z 0
Radius: 0.75
Height: 2
Direction: Y-Axis

you can use triggers to detect when one GameObject enters the space of another without actually colliding.

Now go back to the RobotBehaviour class and **replace your OnCollisionEnter method declaration**.Change its name to OnTriggerEnter, and change the parameter type to Collider:

```
void OnTriggerEnter(Collider other)
```

Open up the Console window and run your game. **Now your game is detecting collisions**—or, at least, it's detecting triggers when the GameObjects get very close to each other, which is just fine for this game.

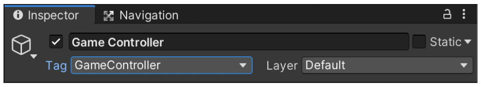
OnCollisionEnter takes a Collision parameter, while OnTriggerEnter takes a Collider parameter. The reason is that when you use a collision, you can detect the specific points where the two bodies contacted each other. There's no way to do that if you're using a trigger—which makes sense because there was no contact between the GameObjects.

Still not detecting triggers? Make sure your RobotPrefab has a Rigidbody.

Now that your game is detecting collisions, you're ready to **make your robots deadly** by modifying the GameController, MoveToClick, and RobotBehaviour classes.

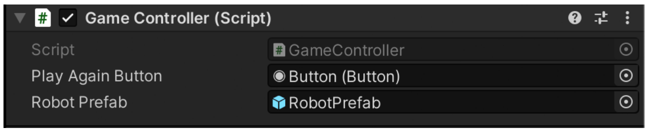
Move the GameController into its own GameObject

1. Add an empty GameObject to the scene by right-clicking on SampleScene and **choosing Create Empty** from the GameObject menu. This will add a GameObject named *GameObject* at the bottom of the Hierarchy window that only has a Transform component.
2. **Rename your empty GameObject** *Game Controller* and **set its tag** to GameController. You don't have to create a new tag—there's already a GameController tag in Unity's default list of tags.



Game Controller
Tag GameController
Layer Default

3. **Remove the GameController** script component from Main Camera. Then **drag the GameController script** onto the empty Game Controller GameObject you just added. Set the Play Again Button and Robot Prefab fields.



Game Controller (Script)

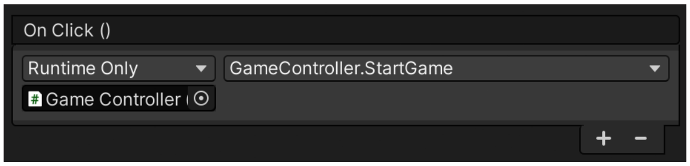
Script GameController
Play Again Button Button (Button)
Robot Prefab RobotPrefab

You should now have a Game Controller (Script) component in your new Game Controller GameObject, and not in the Main Camera.

4. **Add a public PlayerDied method** to the GameController class that sets the GameOver property to true.

Modify the MoveToClick, PlayerBehaviour, and RobotBehaviour classes

1. **Modify the MoveToclick class** to use the GameController tag to find the GameObject the GameController is attached to (it still expects GameController to be attached to Main Camera).
2. **Modify the PlayerBehaiviour class** to add private GameController and NavMeshAgent fields and **set them in the Awake method**, just like you did in the MoveToClick class (you can even copy and paste them—don't forget to add the `using UnityEngine.AI;` line at the top of the file). Add a **public method called StopMoving** that sets the NavMesh Agent's position to the current position. **Add an Update method** that calls it if the game is over.
3. Modify both the PlayerBehaviour and RobotBehaviour classes to **add a private GameController field** and set it in the Awake method, so it matches the one in PlayerBehaviour.
4. **Modify RobotBehaviour.OnTriggerEnter** to call GameController.PlayerDied if the robot collides with the player, but do nothing if it collides with another robot. While you're there, delete the if statement that calls Debug.Log.
5. **Change the Button's OnClick event** to use the Game Controller GameObject to call GameController.StartGame.



On Click ()

Runtime Only
GameController.StartGame
Game Controller

When you use the select button bring up the Select Object window, you'll find Game Controller on the Scene tab.

6. If you've done everything above, the game will end when the player hits a robot—but it's not quite right. The "Play again!" button appears, and the game stops responding to mouse clicks, but the player keeps moving until it reaches its destination. ***Can you figure out how to stop the player moving as soon as the game is over?***

Exercise Solution

Now that your game is detecting collisions, you're ready to **make your robots deadly** by modifying the GameController, MoveToClick, and RobotBehaviour classes. Once you moved the GameController to its own empty object, you had to modify MoveToClick and RobotBehaviour to use it to manage the "Game Over" game mode.

Here's everything that you needed to change in the PlayerBehaviour class (don't forget the `using` line):

```
public class PlayerBehaviour : MonoBehaviour
{
    private Vector3 lastUpdatePosition;
    public NavMeshAgent agent;
    private GameController gameController;

    void Awake()
    {
        agent = GetComponent<NavMeshAgent>();
        gameController = GameObject.FindGameObjectWithTag("GameController")
            .GetComponent<GameController>();
    }

    public void StopMoving() => agent.SetDestination(transform.position);

    void Update()
    {
        if (gameController.GameOver) StopMoving();
    }

    // The rest of the PlayerBehaviour class stays the same
```

This is just like the code you already added to RobotBehaviour to use a tag to find the player, except it's using the GameController tag to find the empty object that the game controller is attached to.

The reason that the player didn't stop moving after it collided with a robot is that the NavMesh Agent's destination was still set to the point that the player clicked. So you can make sure the player stops by setting its NavMesh Agent's destination to the current position if the game is over.

And the PlayerDied method that you added to the GameController class:

```
public class GameController : MonoBehaviour
{
    public bool GameOver { get; private set; } = true;
    public int Score { get; private set; }
    public Button PlayAgainButton;
    public GameObject RobotPrefab;

    public void PlayerDied()
    {
        GameOver = true;
        PlayAgainButton.gameObject.SetActive(true);
    }

        /* The rest of the GameController class is the same as it was before */
    }
}
```

The GameController.PlayerDied method resets the game to "Game Over" mode by setting its GameOver field and displaying the "Play again!" button.

We've been attaching the GameController to the Main Camera because it's added by default to the SampleScene scene in the 3D Unity template, so we know it will never get destroyed. Many developers prefer to put the GameController in its own GameObject, which is why we chose to do that in this Unity Lab. There's no right or wrong answer—both methods have upsides and downsides. Which way do you prefer?

Here's everything that changed in the RobotBehaviour class.

```
using UnityEngine.AI;

public class RobotBehaviour : MonoBehaviour
{
    private PlayerBehaviour playerBehaviour;
    private NavMeshAgent agent;
    private GameController gameController;

    void Awake()
    {
        agent = GetComponent<NavMeshAgent>();
        playerBehaviour = GameObject.FindGameObjectWithTag("Player")
            .GetComponent<PlayerBehaviour>();
        gameController = GameObject.FindGameObjectWithTag("GameController")
            .GetComponent<GameController>();
    }

    // The Start and Update methods didn't change at all

    void OnTriggerEnter(Collider other)
    {
        if (other.gameObject.CompareTag("Player"))
        {
            gameController.PlayerDied();
        }
    }
}
```

The gameController field is set just like in the PlayerBehaviour class.

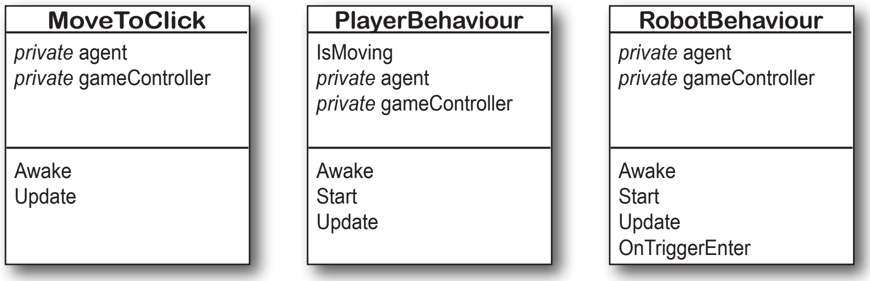
Changing the if statement to only check for player collisions and replacing the Debug.Log statement with a call to the GameController. PlayerDied method causes the game to end when a robot collides with the player.

Here's everything that you needed to change in the MoveToClick class:

```
public class MoveToClick : MonoBehaviour
{
    public NavMeshAgent agent;
    private GameController gameController;

    void Awake()
    {
        agent = GetComponent<NavMeshAgent>();
        gameController = GameObject.FindGameObjectWithTag("GameController")
            .GetComponent<GameController>();
    }

    void Update()
    {
        if (!gameController.GameOver && Input.GetMouseButtonDown(0))

    // The rest of the MoveToClick class didn't change
```

Use inheritance to improve your player and robot classes

Your game is really coming along. We've said throughout the book that we should constantly be looking for ways to **refactor** our code by finding ways to improve its structure without changing its behavior. There's a really good candidate for refactoring in our three classes that drive player and robot behavior. ***Can you spot it?***



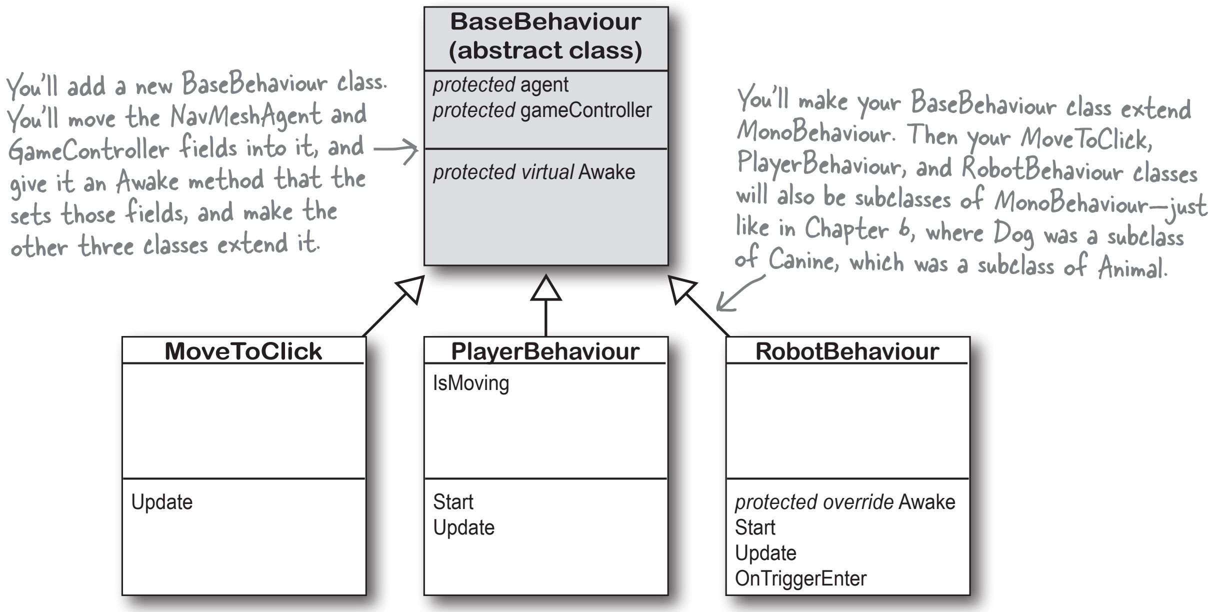
The MoveToClick, PlayerBehaviour, and RobotBehaviour all have fields to store references to the GameObject's NavMesh Agent and the GameController instance, with identical code in their Awake methods to set them.

In Chapter 6 we learned about **inheritance** and **abstract classes**. Now that you've used it to build a few projects and games, take a closer look at the class declaration for any of the classes that Unity added:

```
public class PlayerBehaviour : MonoBehaviour
```

You know what that colon does. Your Unity classes use inheritance to extend a superclass called MonoBehaviour.

We can take advantage of that by extending the hierarchy. Let's ***use inheritance to refactor the duplicate code*** into its own class. We'll call that class BaseBehaviour. Here's what the new class model will look like:



BaseBehaviour (abstract class)
protected agent
protected gameController
protected virtual Awake
You'll add a new BaseBehaviour class. You'll move the NavMeshAgent and GameController fields into it, and give it an Awake method that the sets those fields, and make the other three classes extend it.
You'll make your BaseBehaviour class extend MonoBehaviour. Then your MoveToClick, PlayerBehaviour, and RobotBehaviour classes will also be subclasses of MonoBehaviour—just like in Chapter 6, where Dog was a subclass of Canine, which was a subclass of Animal.
MoveToClick
Update
PlayerBehaviour
IsMoving
Start
Update
RobotBehaviour
protected override Awake
Start
Update
OnTriggerEnter

Refactor your classes to extend BaseBehaviour

Start refactoring your code by **creating a new BaseBehaviour class that extends MonoBehaviour**. It should have the same agent and gameController fields and Awake method as PlayerBehaviour and MoveToClick.

Since your BaseBehaviour class extends MonoBehaviour, any subclasses of it will also extend MonoBehvaiour. Plus, it should never get instantiated—that makes it a perfect candidate for an abstract class, just like we learned in Chapter 6.

```
using UnityEngine;
using UnityEngine.AI;

public abstract class BaseBehaviour : MonoBehaviour
{
    protected NavMeshAgent agent;
    protected GameController gameController;

    protected virtual void Awake()
    {
        agent = GetComponent<NavMeshAgent>();
        gameController = GameObject.FindGameObjectWithTag("GameController")
            .GetComponent<GameController>();
    }
}
```

Your PlayerBehaviour and MoveToClick classes will use the inherited Awake method exactly as it is, but your RobotBehaviour class will need to override it because its Awake method also needs to set its private playerBehaviour field. So make sure you use the virtual keyword.

Now you can **modify the PlayerBehaviour declaration** to extend BaseBehaviour instead of MonoBehaviour:

```
public class PlayerBehaviour : BaseBehaviour
```

And **modify MoveToClick** to extend BaseBehaviour as well:

```
public class MoveToClick : BaseBehaviour
```

Now **delete the agent field, gameController field, and Awake method** from both PlayerBehaviour and MoveToClick. If you don't delete them. If you see the warnings about hiding inherited members (the same ones that you saw in the JewelThief project in Chapter 6), then you know you've forgotten to delete one of them.

The RobotBehaviour class needs a bit more work, because in addition to setting the agent and gameController fields, the RobotBehaviour.Awake method sets the private playerBehaviour field. That's why you used the `virtual` keyword when you declared the BaseBehaviour.Awake method.

Make RobotBehaviour extend BaseBehaviour, and **delete the agent and gameController fields**, but this time **override the Awake method and use the base keyword** to call the Awake method in the base class.

```
public class RobotBehaviour : BaseBehaviour
{
    private PlayerBehaviour playerBehaviour;
    private bool alive = true;

    protected override void Awake()
    {
        playerBehaviour = GameObject.FindGameObjectWithTag("Player")
            .GetComponent<PlayerBehaviour>();
        base.Awake();
    }
```

The RobotBehaviour.Awake method sets the private playerBehaviour field, then uses the base keyword to call RobotBehaviour.Awake to set its protected agent and gameController fields.

Now run your game again. It should work exactly like before—but with a new and improved refactored class model.

Make the robots die when they collide with each other

The goal of the game is for the player to deactivate as many robots as possible by getting them to collide with each other. So GameController needs to keep track of how many robots there are, and how many of them are still alive. So let's add two private `int` fields to GameController to keep track of them:

```
private int totalRobots;
private int liveRobots;
```

If the player kills all of the robots on the level, the game should reset the level with even more robots, so it needs a method for RobotBehaviour to call every time a robot dies. **Add a public method called RobotDied** that decrements liveRobots and resets the level when the last robot dies:

```
public void RobotDied()
{
    liveRobots--;
    if (liveRobots == 0)
    {
        ResetLevel(totalRobots + 1);
    }
}
```

Now **modify the ResetLevel method** so that it destroys all of the robot GameObjects and resets the liveRobots and totalRobots field before instantiating new robots. You can use the GameObject. FindGameObjectsWithTag method to find the robot GameObjects and destroy them.

```
private void ResetLevel(int numberOfRobots)
{
    foreach (GameObject robot in GameObject.FindGameObjectsWithTag("Robot"))
    {
        Destroy(robot);
    }

    liveRobots = numberOfRobots;
    totalRobots = numberOfRobots;

    for (int i = 0; i < numberOfRobots; i++)
    {
        Instantiate(RobotPrefab);
    }
}
```

Are your robots bouncing off of each other and moving around when the player isn't? Make sure the Robot Prefab's Rigidbody has "Use Gravity" unchecked and "Is Kinematic" checked—and do the same for the Player's Rigidbody. Now neither the player or the robots will respond to physics forces.

Finally, let's add **state** to RobotBehaviour: each robot can either be alive or dead. Add a private bool field to RobotBehaviour to keep track of each robot's state:

```
private bool alive = true;
```

Modify the first line of the RobotBehaviour.Update method so that the robot only moves while it's alive by adding **`alive && !gameController.GameOver &&`** to the conditional check in the if statement:

```
void Update()
{
    if (alive && !gameController.GameOver && playerBehaviour.IsMoving)
        agent.SetDestination(playerBehaviour.transform.position);
    else agent.SetDestination(transform.position);
}
```

We want the robots to die if they collide, so modify RobotBehaviour.OnTriggerEnter to set the robot's state to dead and call GameController.RobotDied if it hits another robot—and to keep the player safe if it hits a dead robot:

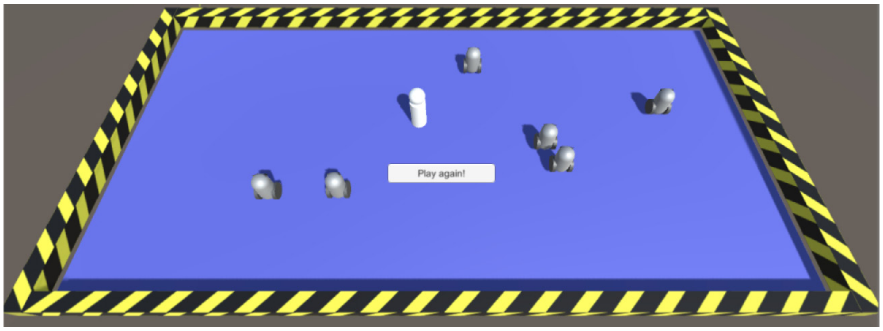
```
void OnTriggerEnter(Collider other)
{
    if (alive && other.gameObject.CompareTag("Player"))
    {
        gameController.PlayerDied();
    }
    else if (alive && other.gameObject.CompareTag("Robot"))
    {
        alive = false;
        gameController.RobotDied();
    }
}
```

Let's do one more thing. It's difficult to see which robots are dead, so let's make them change color when they die. **Add these two lines** to RobotBehaviour.OnTriggerEnter ***right before*** it calls GameController.RobotDied:

```
Renderer renderer = gameObject.GetComponent<Renderer>();
renderer.material.color = Color.black;
```

This gets the Renderer component, which is the Unity object that determines how a GameObject is drawn on the screen. Its material property contains a reference to a Material object, and you can use that object to change the material while your game is running. Try putting a breakpoint on the second line and inspecting the value of renderer.material.name—it will show you the name of the material you used for the robot.

Run your game. ***Hold on, something's not right.*** There are six robots when you start the game. Shouldn't there only be five? When you click the Play Again button, it resets to five?! Weird! Even a small glitch that doesn't affect gameplay is worth tracking down, because ***good developers can't stand mysteries in their code***.



Why is the game creating six robots when it first starts, and not five? (You may need to adjust your Main Camera's position and rotation to see the whole play area.)

Use Debug.Log to track down the problem.

Let's debug this with Debug.Log and the Console window. Add this statement to RobotBehaviour.OnTriggerEnter right after it calls GameController.RobotDied: `Debug.Log($"I died at {transform.position}");`

Now run your game. As soon as the game starts, your Debug.Log statement writes several identical lines to the console after each robot dies. Try adding more Debug.Log statements—that might help you gain some insight.

Can you use the debug information to come up with a theory about what went wrong?

Unity Events Way Up Close

To fix the glitch that spawned too many robots, we need to answer some questions:

- **What caused each robot's OnTriggerEvent method to fire before its Start method?** The Start method is supposed to be the very first thing that Unity calls when it instantiates a GameObject. Why did Unity call the OnTriggerEnter before the Start method?
- **Why doesn't the glitch reappear when the player starts the game?** The start button calls ResetLevel method, which instantiates the robots in exactly the same way. So why did the Start method work for *those* robots?

To answer these questions, we need to really dig into how Unity calls the methods on the GameObjects in a scene. Games—not just Unity games, but many video games—run in a big loop, which has a sensible name: it's called the **game loop**. (Most of the games that you've built so far in this book have a game loop.)

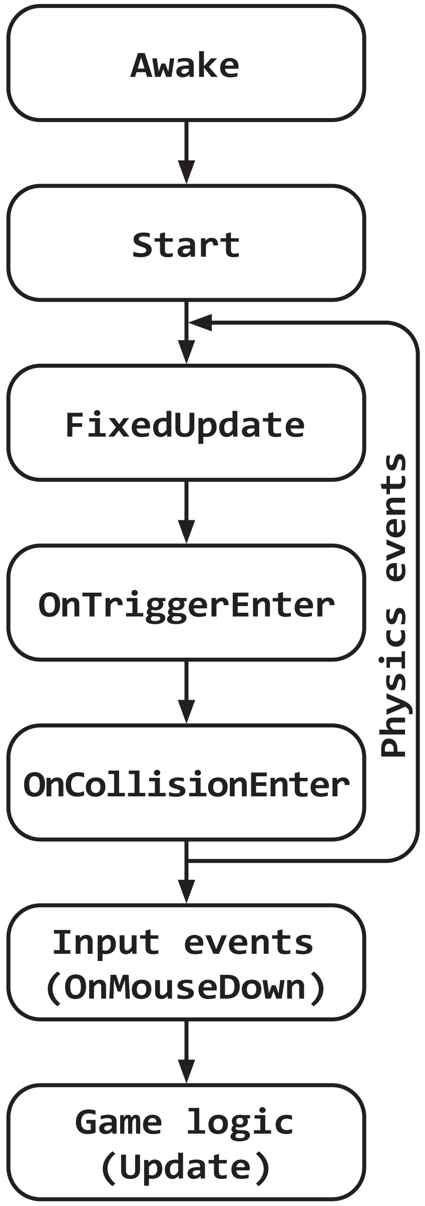
Unity's game loop **runs once for each frame**, calling methods on all of the GameObjects in the scene in a specific order. You can see a diagram of Unity's game loop. Search the Unity manual for ***Order of Execution for Event Functions*** to find a page that goes into a lot of detail about how Unity calls the GameObject methods. There's a diagram of the loop at the bottom of the page—it has methods you haven't seen yet, so we excerpted the relevant ones in a smaller diagram.

We can find answers to our big unanswered questions in these diagrams. Here's what Unity does for every frame:

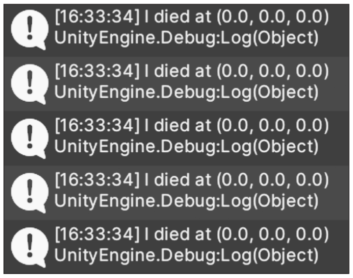
- Unity starts processing a frame by calling the Awake methods for any GameObjects instantiated since the last frame, then their Start methods.
- Then it calls the physics events, including FixedUpdate, OnTriggerEnter, and OnCollisionEnter. They'll often happen multiple times per frame. This is why FixedUpdate may be called more frequently than Update.
- Unity handles input, then it executes the game logic, which includes calling each GameObject's Update method.

When your game first ran ResetLevel it was called from GameController.Start, so after it finished it went on to run the triggers, killing all of the robots. When the last robot died, **its OnTriggerEnter method called ResetLevel during the physics update, so after it reset the level it went on to the input events and game logic before calling any Start methods**. It looped back to initialization after it finished the frame, and then called each robot's Start method to set the position. The second time ResetLevel was called, it happened during the game logic. So when the frame ended, Unity called the Start method, so each robot moved to random positions **before the physics update could detect the collisions**.

And that tells us how to fix the glitch. We can keep the robots from dying when the game first starts by ***repositioning them before the OnTriggerEnter event fires***.



Fix the glitch that shows the wrong number of robots



```
[16:33:34] I died at (0.0, 0.0, 0.0)
UnityEngine.Debug:Log(Object)
[16:33:34] I died at (0.0, 0.0, 0.0)
UnityEngine.Debug:Log(Object)
[16:33:34] I died at (0.0, 0.0, 0.0)
UnityEngine.Debug:Log(Object)
[16:33:34] I died at (0.0, 0.0, 0.0)
UnityEngine.Debug:Log(Object)
[16:33:34] I died at (0.0, 0.0, 0.0)
UnityEngine.Debug:Log(Object)
```

When you added the Debug.Log statement to OnTriggerEnter, did you notice anything interesting about the messages that it printed to the Console window? Every robot that **died printed the same position**—they all printed the message **`I died at (3.0, 1.0, 0.0)`**. Do you recognize that vector? That's the starting position we set when we created the GameObjects for RobotPrefab.

- The reason the robots all die is that each new RobotPrefab instance is created at position (3, 1, 0).
- Since they're all Rigidbodies, they can't all occupy the same space, so they bump into each other and nudge each other around
- This causes **all of their triggers to fire** before the first frame of the game even runs.
- That causes every robot instance's RobotBehaviour.OnTriggerEnter method to run, killing the robot <u>before</u> the Start method has a chance to set its transform.position to a random position on the NavMesh.

So let's fix this by positioning each robot to a random point on the NavMesh as soon as it's instantiated, and before the trigger has a chance to fire. We'll also make some small improvements to the gameplay: we'll make it stop the player's movement, and we'll make sure that no robot is instantiated within 5 units of the player.

```
private void ResetLevel(int numberOfRobots)
{
    foreach (GameObject robot in GameObject.FindGameObjectsWithTag("Robot"))
    {
        Destroy(robot);
    }

    liveRobots = numberOfRobots;
    totalRobots = numberOfRobots;

    PlayerBehaviour player = GameObject.FindGameObjectWithTag("Player")
        .GetComponent<PlayerBehaviour>();
    player.StopMoving();

    for (int i = 0; i < numberOfRobots; i++)
    {
        var robot = Instantiate(RobotPrefab);
        Vector3 position;
        do
        {
            position = RandomPointHelper
                 .RandomPointOnMesh(robot.transform.position, 100) + Vector3.up;
            robot.transform.position = position;
        } while (Vector3.Distance(position, player.transform.position) < 5);
    }
}
```

We'll need this player reference to get the player's position We'll also use it to stop the player moving between levels.

This do loop keeps the robots from being spawned too close to the player, so that the player gets a fighting chance—and so the game doesn't end because a robot was spawned on top of the player. It uses RandomPointHelper to find a random position on the NavMesh, and keeps repeating until it finds a position that's more than 5 units away from the player.

Now that ResetLevel is setting the robot positions, **delete RobotBehaviour.Start method** because you don't need it anymore. **Now run your game.** Only five robots appear at the start of the game.

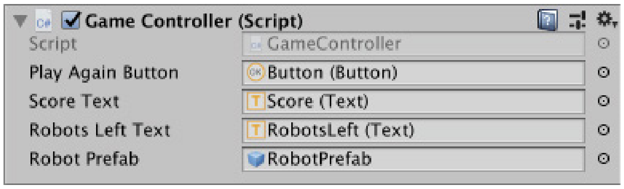
Fix the UI

When you imported the assets from the last Unity Lab, it included the Play Again button, which you've been using. It also included two Text GameObjects, Score and Timer. Let's make them work. First, **rename the Text called Time to RobotsLeft**. You'll use it to show the number of robots left in the play area.

Modify GameController to **add public Text fields**:

```
public Text ScoreText;
public Text RobotsLeftText;
```

Use the select button (◉) to set the fields in the Game Controller script component.

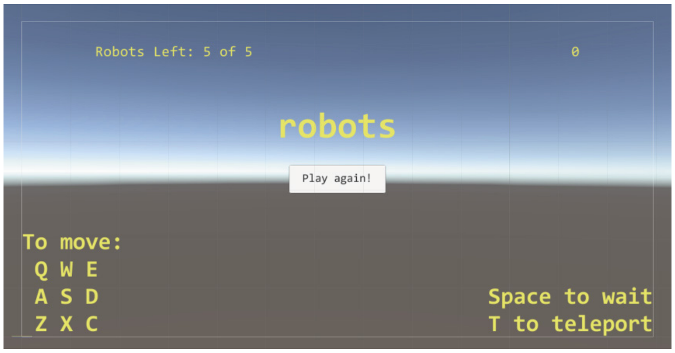


Game Controller (Script)	
Script	GameController
Play Again Button	Button (Button)
Score Text	Score (Text)
Robots Left Text	RobotsLeft (Text)
Robot Prefab	RobotPrefab

Now you can **add a GameController.Update method** that keeps the text up to date. While you're editing GameController, modify GameController.RobotDied to increase the score whenever a robot dies:

```
private void Update()
{
    ScoreText.text = Score.ToString();
    RobotsLeftText.text = $"Robots left: {liveRobots} of {totalRobots}";
}
```

```
public void RobotDied()
{
    Score += 10;
    liveRobots--;
    if (liveRobots == 0)
    {
        ResetLevel(totalRobots + 1);
    }
}
```



Finally, add a title and instructions to the "Game Over" screen. **Add several Text GameObjects** to your Canvas, giving them whatever colors and text you want.

Then **give them all the `Game Over Screen` tag**, and modify the GameController.Update method to use their `enabled` properties to make them visible only when the game is over. Use the FindObjectsOfType method, which works exactly like the other methods that find GameObjects, and enable them if the game is over.

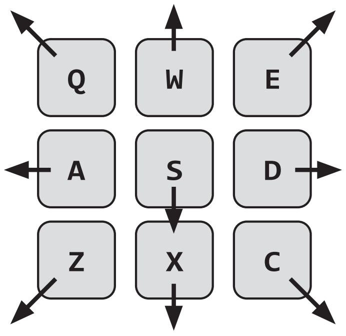
```
private void Update()
{
    ScoreText.text = Score.ToString();
    RobotsLeftText.text = $"Robots left: {liveRobots} of {totalRobots}";

    foreach (Text text in GameObject.FindObjectsOfType<Text>())
    {
        if (text.CompareTag("Game Over Screen")) text.enabled = GameOver;
    }
}
```

When you need to check a GameObject's tag, use its `CompareTag` method instead of doing a string comparison. It's faster, and it can help you prevent bugs because it throws an exception if the tag doesn't exist.

The original robots game was turn-based, but yours is real-time. The goal of this exercise is to modify your game to **turn it into a turn-based game** that works just like the original robots game did, using the tools that you've used in previous Unity Labs to add keyboard controls.

How the controls will work

You'll use the standard WASD keyboard controls plus the diagonal directions Q, E, Z, and C, to move your player in eight directions. You'll make both S and X move the player down. You'll make the space key make the player wait for .5 seconds while the robots move, and you'll make the T key teleport the player.



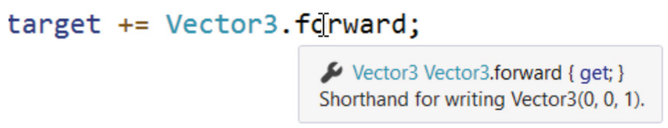
Modify PlayerBehaviour.IsMoving to use a private float field called `waiting`

Add a **private float field called `waiting`** to PlayerBehaviour. You'll use this field to make the player wait for robots to move when the space key is hit. Then **modify the IsMoving property** to return true if waiting is greater than zero.

Modify the PlayerBehaviour.Update method to respond to the keyboard

Right now the PlayerBehaviour.Update method just stops the player from moving if the game is over. Your job is to add logic to the method that responds to keyboard presses. Here's what it will do:

- If the `waiting` field is greater than zero, the player is still waiting for the robots to finish moving after pressing the spacebar. **Subtract** the amount of time that's elapsed since the last frame **from `waiting`**.
- If the game <u>isn't over</u> and the player <u>isn't moving or waiting</u>, the Update method should look for key presses.
- Use `Input.GetKey(KeyCode.Space)` to **detect if the player is pressing space**, then add 0.5f to `waiting`.
- If the player presses Q, W, E, A, S, D, Z, X, or C, **set a Vector3 variable to the player's new position**, and then **set the NavMesh agent's destination** to that position.
- You can use **vector addition** to find the new position with a combination of Vector3.forward, Vector3.back, Vector3.up, and Vector3.down. XML documentation with their values pops up when you hover over them:



```
target += Vector3.forward;
```

Vector3 Vector3.forward { get; }
Shorthand for writing Vector3(0, 0, 1).

- Teleporting is dangerous to the player, so let's make it extra dangerous. If the T key is down, move the player to a new random position **whether or not the player is moving**. If the user keeps holding down the T key, the Player GameObject should keep teleporting to a new location each frame. That will make teleporting dangerous, and also looks interesting.

Don't be a code hoarder.

Does it feel weird to delete a class that you put so much work into? Code hoarding is a really common trap for developers. But if there's a statement, method, or even an entire class that doesn't do anything anymore, you should feel comfortable deleting it. When you need to fix a bug later, you'll be glad not to have to debug through it!

Delete the MoveToClick class

You don't want the game to respond to mouse clicks anymore, so **remove** the MoveToClick script component from Player, then **delete the class**.

Exercise Solution

Changing your game from real-time to turn-based was just a matter of making the player wait until the current move is done before initiating the next one—and you could do that all just by modifying the PlayerBehaviour class. Here's everything that you needed to change in it:

Making the IsMoving property return true if the player is waiting will cause the robots to move and the player to stay still any time the waiting field has a positive value. Update subtracts the delta time from waiting, so setting waiting to 0.5f causes the player to wait for the robots to move for half a second.

```
private float waiting = 0f;

public bool IsMoving =>
        (waiting > 0f) ||
        (Vector3.Distance(transform.position, lastUpdatePosition) > 0.01f);

void Update()
{
    if (gameController.GameOver) StopMoving();
    if (waiting >= 0f) waiting -= Time.deltaTime;
    if (!gameController.GameOver && !IsMoving)
    {
        Vector3 target = transform.position;

        if (Input.GetKey(KeyCode.T))
        {
            Start();
            target = transform.position;
        }
        else if (Input.GetKey(KeyCode.Q))
            target += Vector3.forward + Vector3.left;
        else if (Input.GetKey(KeyCode.W))
            target += Vector3.forward;
        else if (Input.GetKey(KeyCode.E))
            target += Vector3.forward + Vector3.right;
        else if (Input.GetKey(KeyCode.D))
            target += Vector3.right;
        else if (Input.GetKey(KeyCode.C))
            target += Vector3.back + Vector3.right;
        else if (Input.GetKey(KeyCode.S) || Input.GetKey(KeyCode.X))
            target += Vector3.back;
        else if (Input.GetKey(KeyCode.Z))
            target += Vector3.back + Vector3.left;
        else if (Input.GetKey(KeyCode.A))
            target += Vector3.left;
        else if (Input.GetKey(KeyCode.Space))
            waiting = .5f;

        agent.SetDestination(target);
    }
}
```

The Start method sets the player to a random position, so we reused it. This could be clever, but it could also be dangerous. What if we need to modify the Start method later? What bugs could that cause?

If we detected the E key, we want the player to go up and to the right. Vector3.forward is (0, 0, 1), so adding it to the current position will add 1 to the Z position, which will move the Player GameObject 1 unit up. Vector3.right is (1, 0, 0), so adding it will add 1 to the X position, which will move the GameObject 1 unit to the right. Adding both of them to the current position moves the player diagonally up and to the right.

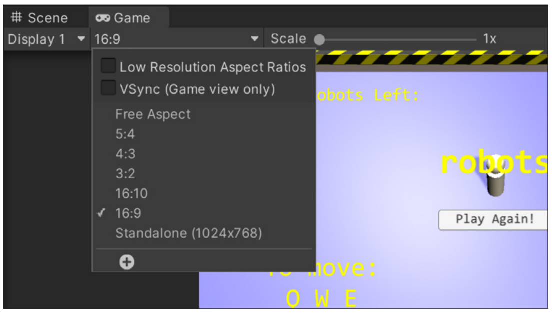
If you remember your geometry classes from school, you might have realized that moving 1 unit to up and 1 unit to the right actually moves the player more than 1 unit diagonally. That's what we want in this case, because that's how the original robots game works, too.

If the game feels a little sluggish, adjust the Acceleration property in the NavMesh Agents attached to Player and RobotPrefab. Try setting it to 50. You can also make the game more or less difficult by changing their Speed properties.

Use an orthographic camera

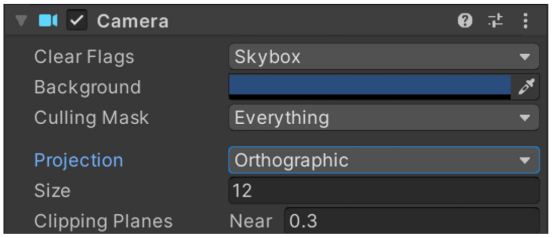
Take a close look at the screenshot of the `robots` game at the beginning of this Unity Lab. It's a text rendering of a top-down view: the player is represented by an **@**, each live robots is represented by a **+**, and a "junk heap" of dead robots is represented by a *****. We're using graphics and not text, but we can still give this game a similar top-down look. We'll do it by changing Main Camera to an **orthographic** projection.

In the third Unity Lab we learned about the difference between perspective and isometric views, and used the Scene Gizmo to switch to an isometric view that showed all objects of equal size no matter how far away they were. When you change a Unity camera to use an orthographic projection, it does the same thing, showing all objects as having an equal size no matter their distance from the camera.



★ Start by **using the aspect ratio dropdown** in the Game window to change the Game view to 16:9 (mainly so your screen will match our screenshots).

★ Then select Main Camera, and use the Inspector window to modify the Camera component. **Change Projection to Orthographic, and Size to 12**.



Clear Flags: Skybox
Background
Culling Mask: Everything
Projection: Orthographic
Size: 12
Clipping Planes: Near 0.3

★ Then adjust the Main Camera's Transform component to **give it position (0, 20, 0) and rotation (90, 0, 0)**. You should see the entire play area in the Camera Preview.

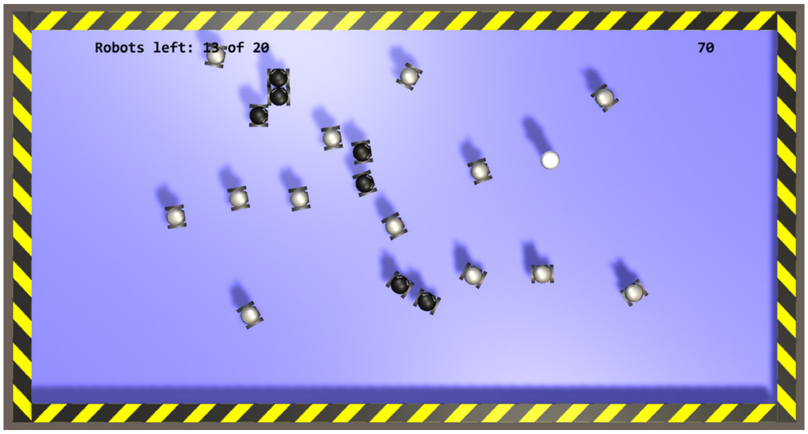
★ Now let's do a little experimenting. Start your game, select Main Camera, then click the Y label next to Position in the Transform component and drag up and down. It's probably not the way you expected it to, right? What's going on?

When you move a perspective camera up and down along the Y-axis, everything gets closer or farther away—or, more specifically, everything gets smaller and larger. Objects in an orthographic camera are always the same size no matter how far away they are from the camera. So moving the camera farther away may change things like lighting, but it won't make the entire scene get smaller. Try setting the camera's Y position to 2.1—it will cut off the top of the robots, so you'll just see a hole where the plane of the camera hits.

Now just do one more thing to make your game really similar to the robots. In the GameController.StartGame method, have it start the game with 20 robots by changing the last line of the method to `ResetLevel(20);` – and change the GameController.RobotDied method to add three robots when it resets the level:

```
ResetLevel(totalRobots + 3);
```

Now your game will have a lot of robots, just like the original.



Get creative!

There are so many ways that you can get practice writing code by improving this game. Here are some ideas:

- Add sounds when robots die, when they move, and when the player moves or teleports.
- See if you can implement the "Wait for the end" feature of the original `robots` game: when the player presses W, wait until either the player dies or all of the robots on the level die, then award an extra "wait bonus" point.
- Try out different camera angles with the perspective camera. Can you add a script to the Main Camera that moves it around in a way that improves the game?
- **Here's a coding challenge:** See if you can figure out how to make the robots get faster as the levels increase. *(And here's a hint: you already have a reference to the NavMesh Agent.)*
- Experiment with different ways to do turn-based versus real-time controls. Can you use InvokeRepeating to make the robots move on a timer instead of waiting for the player? How does that affect the game's dynamics?

BULLET POINTS

- Use the **Export Package** feature to export some or all of the assets from one Unity project, and the **Import Package** feature to import them into another one.
- A **skeleton** is an outline of a class that has the methods and fields, but instead of functional code, it has stub code that serves as a placeholder until you write the real code.
- Unity games won't crash when a method throws an exception. Instead, it will just **stop executing that method**, but continue running the game.
- The **NavMesh.SamplePosition method** takes a starting point and finds the closest position to it on the scene's NavMesh.
- The **Random.insideUnitSphere property** returns a Vector3 at a random location somewhere inside the 1-unit sphere centered at (0, 0, 0). You can multiply it by a float to get a random location inside a larger sphere.
- You can **find a random point on a mesh** by doing a random walk, where you use Random.insideUnitSphere to find a random point near the current location, feed it into NavMesh.SamplePosition to find the nearest point on the NavMesh, and repeat.
- The **FixedUpdate method** is like the Update method, except that it runs exactly 50 times per second, no matter what the frame rate of the game is.
- **Adding vectors** means adding their X values, adding the Y values, and adding the Z values to find a new vector.
- One of the easiest ways to improve your code is to **rename your variables, methods, and properties**. The Visual Studio IDE lets you rename all instances of a variable at the same time. It's a simple way to **refactor**.
- The Unity editor **won't let you set a prefab field to a GameObject in a scene** because it might not be available if the prefab is being used in a different scene.
- Collision and trigger detection rely on the physics engine. You must **add a RigidBody component to both GameObjects** involved in the collision, or your OnTriggerEnter method won't ever get called.
- NavMesh agents know how to avoid each other and obstacles. Use **triggers** to detect when one GameObject enters the space of another without actually colliding.
- Unity always executes event methods like Start and Update in a specific order. The **Order of Execution for Event Functions** manual page has details on that order.
- The **FindObjectsOfType method** works like the other methods that find GameObjects and returns a collection of GameObjects that match a specific type.
- When you need to check a GameObject's tag, use its **CompareTag method** instead of doing a string comparison.
- A camera with an **orthographic projection** does not show perspective, so objects always appear the same size no matter how far away they are from the camera.