



C# Fundamentals

Today's Agenda

- **Variables & Scope** – Understanding variable types, declaration, and assignment.
Differentiating global and local scope.
- **Debugging & Errors** – Fixing common Unity errors like missing brackets and disconnected scripts.
- **Player Input & Movement** – Capturing player and moving them with Transform and Rigidbodies.
- **Collision Detection & Tags** – Using `OnTriggerEnter2D()` to detect collisions.
- **Scene Management & Transitions** – Adding scenes to **Build Settings**. Switching scenes with `SceneManager`.

Video

Alongside the slide, you can watch the accompanying video, which covers the same topics in a less detailed manner.



Set Up

Open Up Geo Quest Level 1

We're going to start fresh in a new scene and begin laying the foundations for **Geo Quest**.

Navigate to:

Assets > Geo_Quest > Scenes

Then, open **Scene_1**.

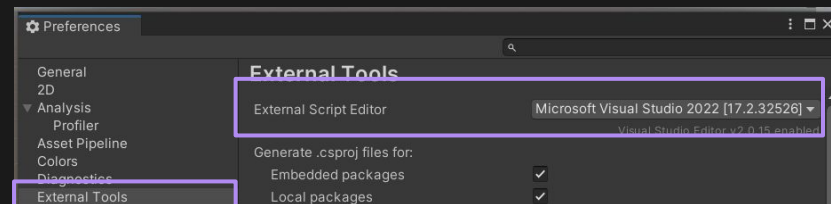
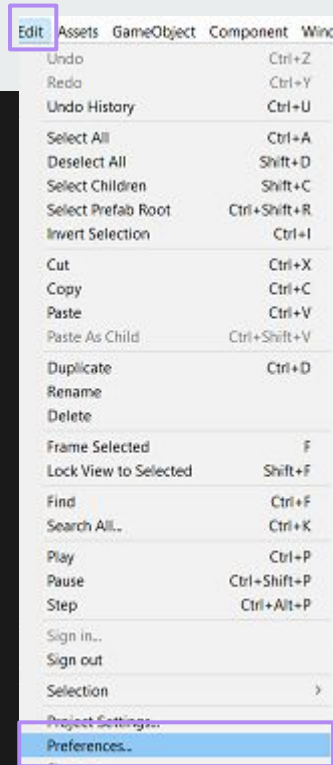


Unity Visual Studio Connected

Before we start programming, let's make sure that **Unity** and **Visual Studio** are properly connected.

1. Go to **Edit** in the **File** menu dropdown.
2. Select **Preferences**.
3. In the **Preferences** window, navigate to **External Tools**.
4. Click on the **External Script Editor** dropdown.
5. If **Visual Studio** isn't selected, choose it.

Now, all scripts will open in **Visual Studio** by default!



Script Basics

Create Script

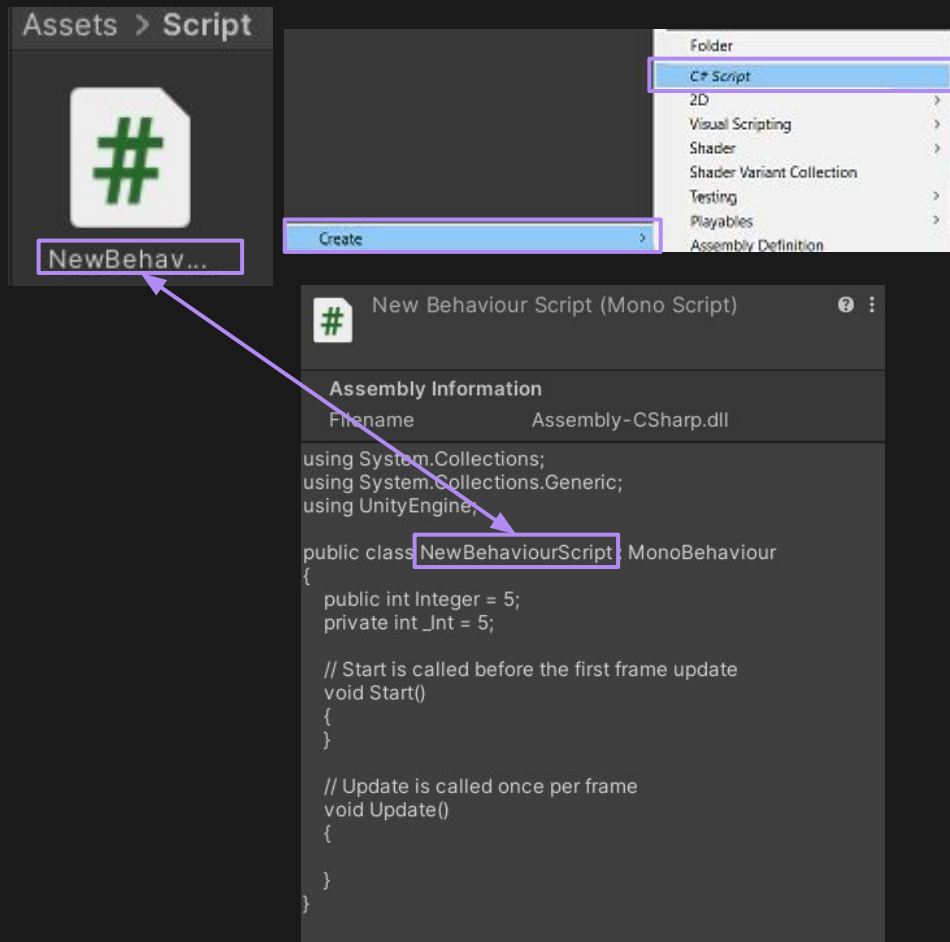
Now that we're sure scripts will open correctly, we create a **C# Script** in Unity.

In the **Project View** and create a **C# Script**.

Name it carefully the first time, Unity requires the script's filename to match the class name inside it.

- If you need to rename it later, update both the **file name** and the **class name** inside the script to avoid issues.

When selecting the script in the **Project View**, the **Inspector** will show a preview of its contents.



Challenge Create Script

Now, let's put your skills to the test!

Navigate to **Assets > Geo_Quest > Scripts**.

Create a new **C# Script** and name it **GeoController**.

Ensure the file name and class name match exactly to avoid errors.

Once done, double click on the script to open the script in **Visual Studio**.

```
Unity Script | 0 references
public class GeoController : MonoBehaviour
{
    // Start is called before the first frame update
    Unity Message | 0 references
    void Start()
    {
        ...
    }

    // Update is called once per frame
    Unity Message | 0 references
    void Update()
    {
        ...
    }
}
```



Anatomy of a Basic Script - Libraries

The **Using** section of the script connects different **libraries** to the script you are programming.

Libraries are collections of **methods/functions** that you can import to extend the script's functionality.

For example, **Start()** and **Update()** methods come from **Unity's libraries** and are included by default.

You might notice that some libraries in the **Using** section are **grayed out**, this means they are **imported but not currently used** in the script. The **UnityEngine** library, on the other hand, is active because it's required for game development.

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class NewBehaviourScript : MonoBehaviour
6  {
7      // Start is called before the first frame update
8      void Start()
9      {
10
11      }
12
13     // Update is called once per frame
14     void Update()
15     {
16
17     }
18 }
```

Unity Script | 0 references

Unity Message | 0 references

Unity Message | 0 references

Anatomy of a Basic Script - Class Name

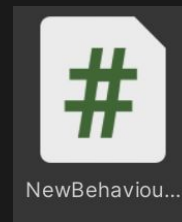
The **public class NewBehaviourScript** section defines our **class**.

A **class** is essentially a **blueprint** for a **component** that can be attached to a **GameObject**.

Everything inside its **curly braces { }** defines its **properties and behaviors**.

Just like how a **Prefab** is a template for a **GameObject**, this script is a **template** for a **component** that we will later attach to an object in Unity.

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class NewBehaviourScript : MonoBehaviour
6  {
7      // Start is called before the first frame update
8      void Start()
9      {
10         //
11     }
12
13     // Update is called once per frame
14     void Update()
15     {
16         //
17     }
18 }
```



Anatomy of a Basic Script - Curly Braces

Curly braces `{ }` are very important in C# programming. They define the **body** of a class, function, or any subsection of code.

- **Shift + [** creates `{` (opening curly brace).
- **Shift +]** creates `}` (closing curly brace).

Everything inside a **pair** of curly braces **belongs** to whatever opened them.

For example:

- The **first set** of curly braces `{ }` belongs to the **class**.
- The **second set** belongs to the **Start()** function.

Many errors in coding happen because of **missing or misplaced curly braces**, so always make sure they open and close properly!

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  [Unity Script | 0 references]
6  public class NewBehaviourScript : MonoBehaviour
7  {
8      // Start is called before the first frame update
9      [Unity Message | 0 references]
10     void Start()
11     {
12
13     }
14
15     // Update is called once per frame
16     [Unity Message | 0 references]
17     void Update()
18     {
19
20     }
21 }
```

Anatomy of a Basic Script - Extension

: **MonoBehaviour** is an **extension** of the class, meaning our script inherits all the properties and methods of **Unity's MonoBehaviour** class.

- It allows our script to be used as a **Component** on a **GameObject**.
- It gives us access to built-in Unity functions like **Start()**, **Update()**, and many more.
- By inheriting **MonoBehaviour**, our script becomes **part of Unity's GameObject system**.

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class NewBehaviourScript : MonoBehaviour
6  {
7      // Start is called before the first frame update
8      void Start()
9      {
10         //
11     }
12
13     // Update is called once per frame
14     void Update()
15     {
16         //
17     }
18 }
```

Extension Vs Library

- **Extensions (Inheritance)** → Copy all properties and methods from the class they extend. Since our script extends `MonoBehaviour`, it gains functions like `Start()`, `Update()`, and Unity's event-driven behaviors.
- **Libraries (Imports)** → Provide access to other parts of Unity's system without copying them directly. They allow interaction with features like **Sprite Renderers**, **Player Input**, and **Physics**.

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  [Unity Script | 0 references]
6  public class NewBehaviourScript : MonoBehaviour
7  {
8      // Start is called before the first frame update
9      [Unity Message | 0 references]
10     void Start()
11     {
12     }
13
14     // Update is called once per frame
15     [Unity Message | 0 references]
16     void Update()
17     {
18     }
19 }
```

Anatomy of a Basic Script - Functions or Methods

Within the body of our class, you will always start with two methods: **Start()** and **Update()**.

Start() → Runs **only once** when the object is first created or activated. Used for initialization.

Update() → Runs **every frame**, meaning it continuously executes as long as the game is running. This is where we put logic that needs to update constantly, like player movement or animations.

A **function (or method)** is a **block of code** that executes a specific action when called.

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  [Unity Script | 0 references]
6  public class NewBehaviourScript : MonoBehaviour
7  {
8      // Start is called before the first frame update
9      [Unity Message | 0 references]
10     void Start()
11     {
12     }
13
14     // Update is called once per frame
15     [Unity Message | 0 references]
16     void Update()
17     {
18     }
19 }
```


Execution of Code

When we write code, execution starts **from the top** and moves **line by line** in a sequential manner.

The computer **only executes one action at a time**, no matter how complex the program is.

However, because computers process instructions **extremely fast**, it often feels like multiple things are happening simultaneously.

```
// Start is called before the first frame update
Unity Message | 0 references
void Start()
{
    Debug.Log("Print 1");
    Debug.Log("Print 2");
    Debug.Log("Print 3");
    Debug.Log("Print 4");
    Debug.Log("Print 5");
}
```



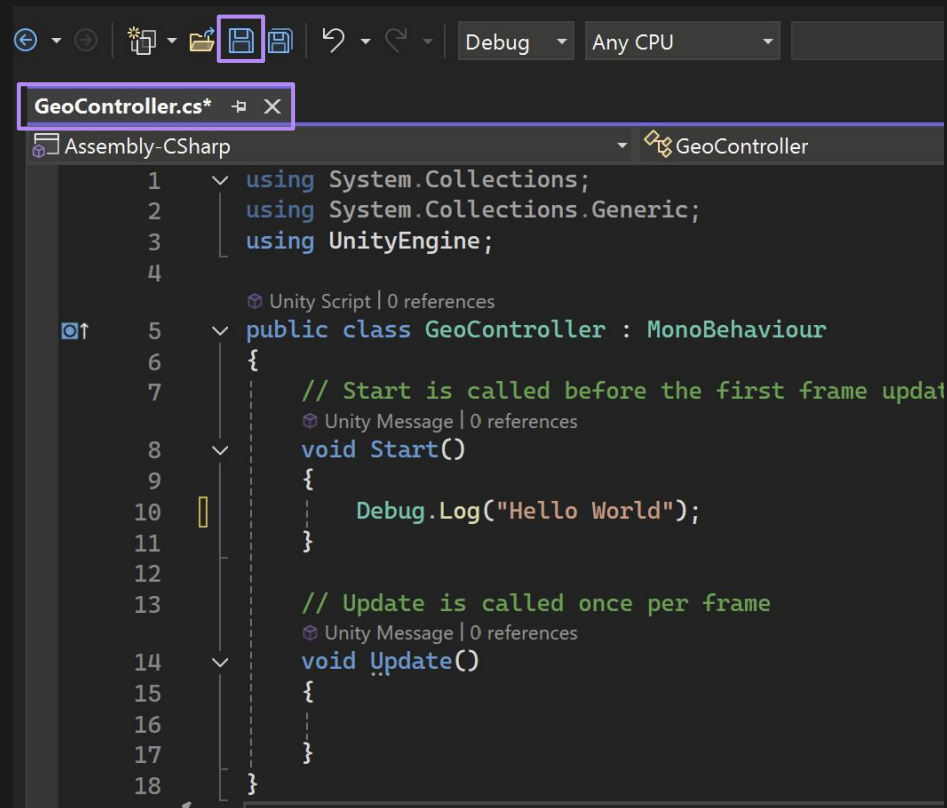
Saving

Whenever you make changes to your script, you must **save** it for those changes to take effect.

You will see if you need to save by the (*) **asterix** just like in Unity's Scenes.

When you return to **Unity**, there might be a slight delay.

This happens because **Unity is compiling** the script, checking for errors, and updating the project.



Attaching

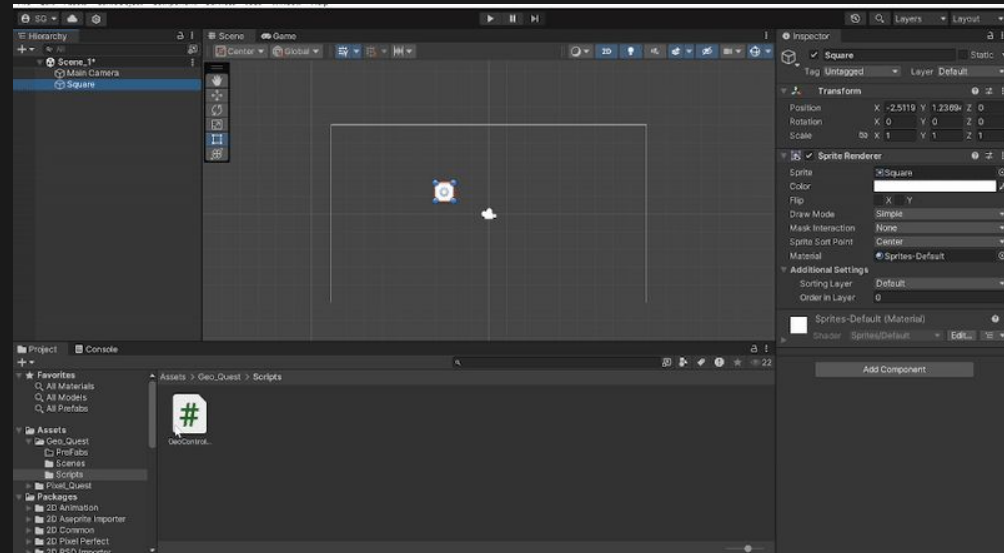
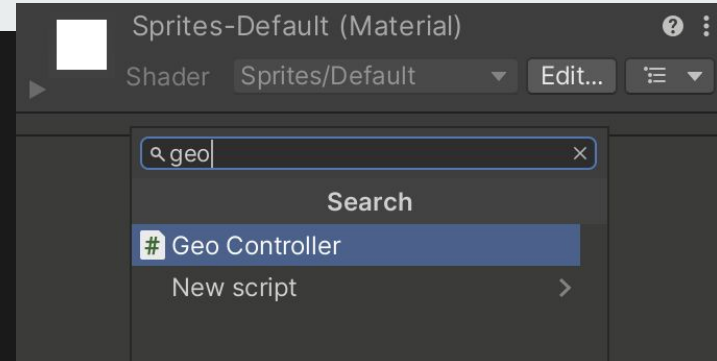
Before a script can do anything, it **must be attached to a Game Object** in the scene.

1. Using Add Component

- Select the Game Object in the **Hierarchy**.
- In the **Inspector**, click **Add Component**.
- Type the name of your script and select it.

2. Drag and Drop

- Click and **drag** the script from the **Project View**.
- Drop it into the **Inspector** of the Game Object.

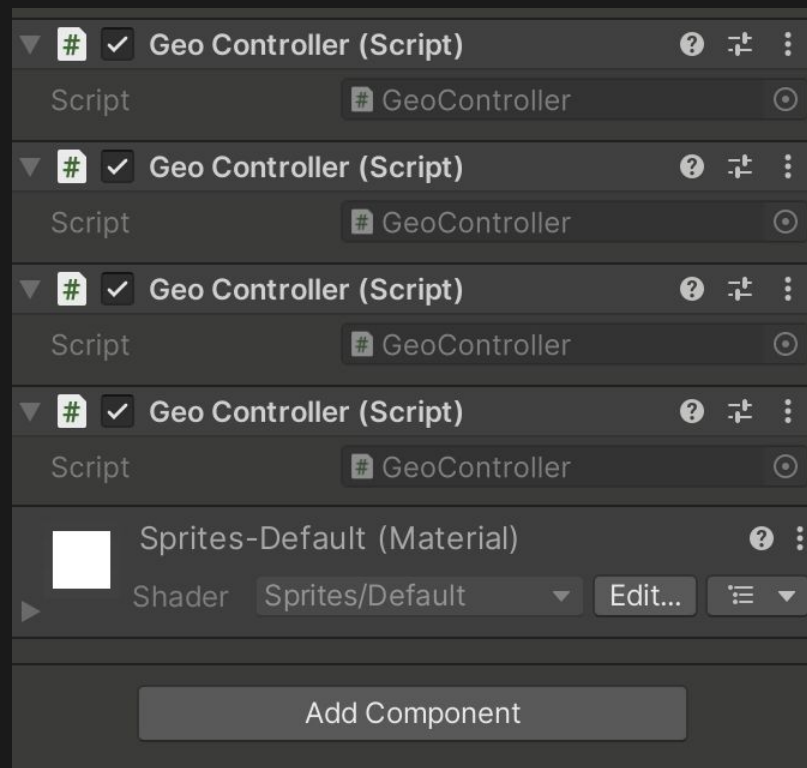


Script is a Template

Just like **Prefabs**, you can attach a script to **multiple objects** and even **multiple times** to the **same object**.

Accidentally attaching a script to the wrong object can cause unintended behavior in your game.

If something seems off, check the **Inspector** to see if the script is duplicated or attached to the wrong object!



Challenge “Hello World”

Now, let's put your skills to the test!

Create a Game Object with a Square Sprite.

Inside of GeoController Script inside the the Start Function type in `Debug.Log(“Hello World”);`

Make sure to save your changes in Visual Studio then attach the script to your new game object.

Click play and check the console for the print out.

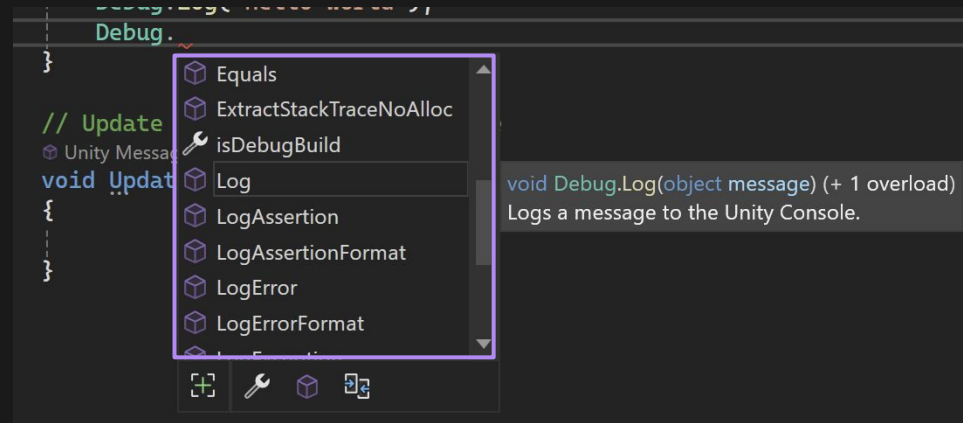


[12:43:41] Hello World
UnityEngine.Debug:Log (object)

Debug and Accessing Operator and Variables

1. **Debug** is a **Class** in Unity's **Engine** library, it contains useful functions for debugging and displaying messages in the **Console**.
2. **The Period (.)** is the **access operator**. It allows us to use functions or variables inside the **Debug** class. When you type **Debug.** in **Visual Studio**, a dropdown appears, showing available functions.
3. **Log()** is a **Method** (or Function) inside the **Debug** class. It sends a message to the **Console**, helping developers track game events.
4. **The String ("Hello World")** The text inside the parentheses is what will be displayed in the **Console**. You can replace it with other messages or even variables.

```
Debug.Log("Hello World");
```

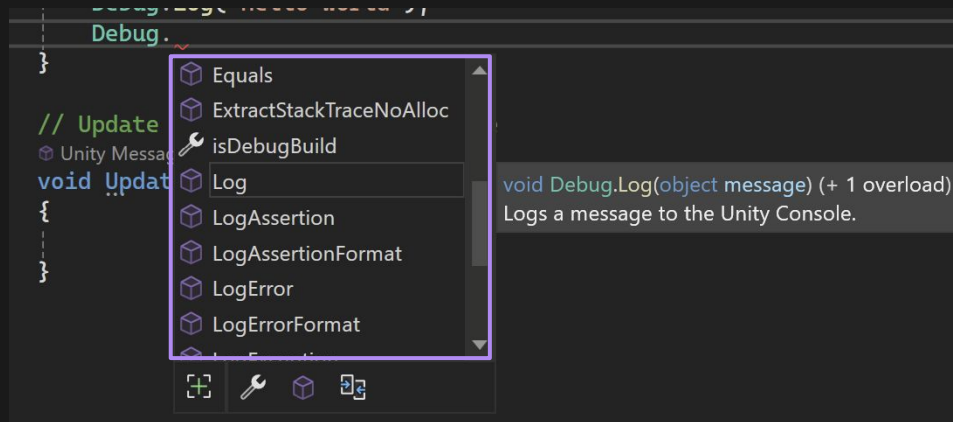
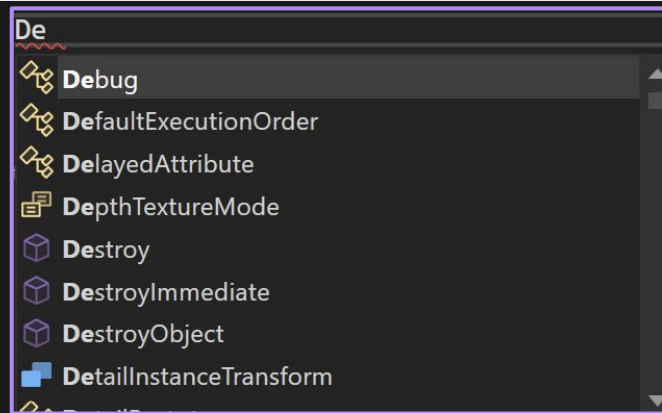


Auto Complete

Many errors in coding, especially in Unity, come from simple typos or misspellings. Using **auto-complete** can help avoid these mistakes.

When typing, Unity suggests **classes, variables, and functions**. You can navigate the list using the **Arrow Keys** (**↑**, **↓**) and press **Tab** or **Enter** to select the highlighted suggestion.

You can also **hover** and **click** with your mouse.



Errors

Erros: Semicolon

Before we start writing code, let's go over some basic problems you might encounter and should be able to fix on your own.

One of the most common issues is **missing a semicolon (;)**.

A **semicolon acts like a period in C#**, telling the computer when a statement is complete. Since **some lines of code can be long and span multiple lines**, the computer needs a clear indication that you're finished with a command.

Without a semicolon, Unity won't know where one instruction ends and the next begins, leading to errors.

```
public string nextLevel = "GeoLevel_2";
```

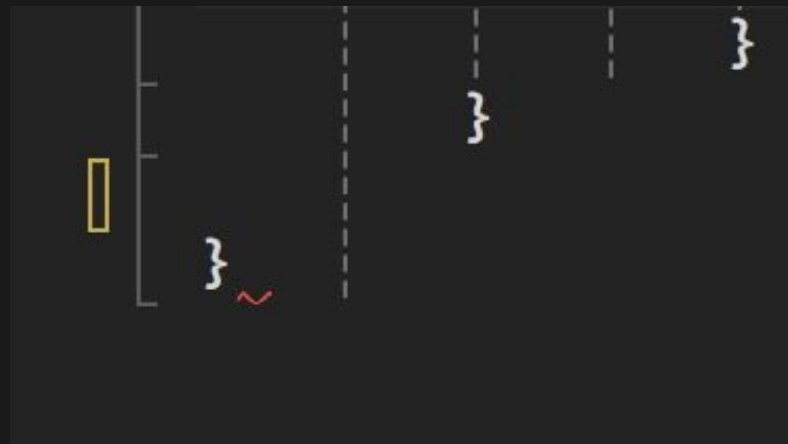
```
public string nextLevel = "GeoLevel_2" ~
```


Error: Braces

Another very common issue you will run into is **missing or having too many curly braces {}**.

Curly braces **define code blocks**, they **encapsulate** parts of your script, like functions or classes.

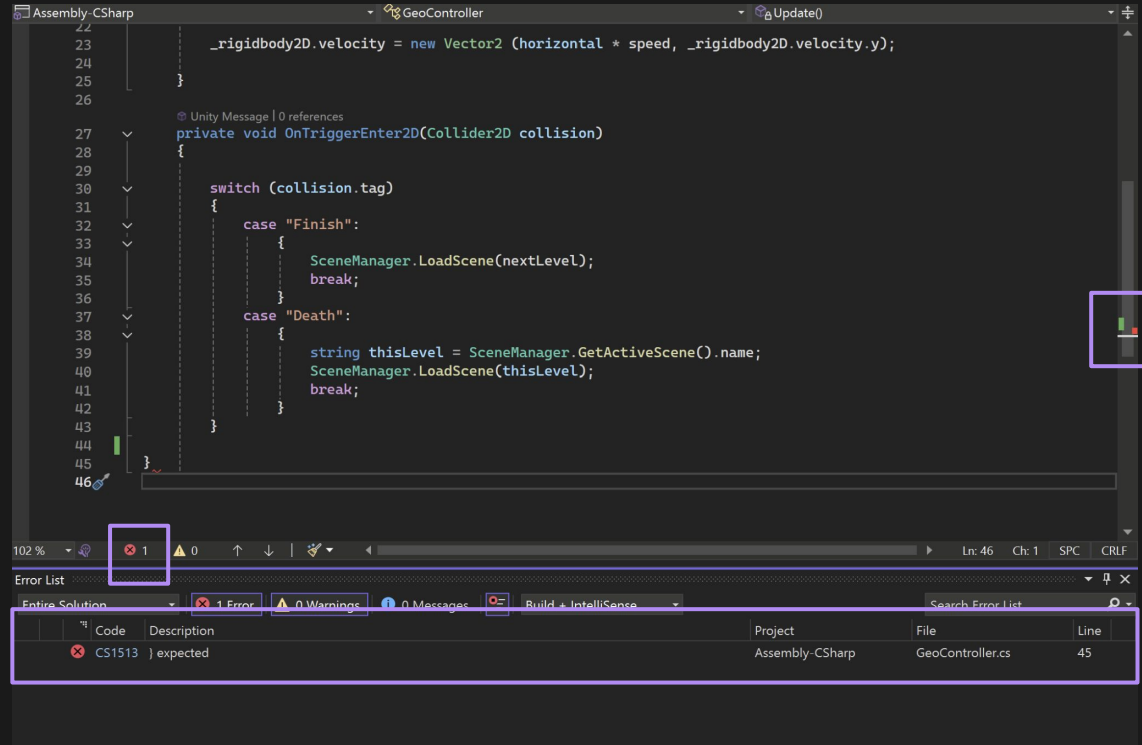
If you forget to close a brace or add an extra one, Unity won't know where a section of code begins or ends.



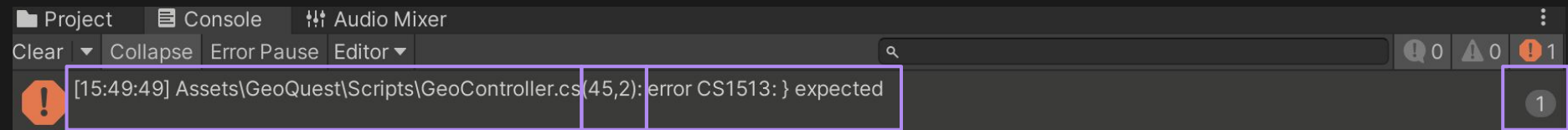
Debugging in Visual Studio

If you make an error by **deleting or changing something incorrectly**, Visual Studio will notify you in a few ways:

1. A **pop-up at the bottom** will appear, telling you how many errors you have encountered.
2. **Red indicators** will show up on the **scroll bar**, marking the locations of errors in your script.
3. If you **click the error pop-up**, it will open the **Error List**, which provides a general idea of the issue.



Errors: Console Debugging



Even if your code **saves without errors**, bugs can still appear when you **run the game**. These runtime errors usually happen when something is **not connected properly** or **goes outside its defined limits**.

To **debug** effectively, check the **Console** in Unity:

1. **Locate the file** causing the issue (e.g., `Assets\GeoQuest\Scripts\GeoController.cs`).
2. **Find the exact line and character** where the error occurs (e.g., **line 45, character 2**).
3. **Read the error message**—in this case, it might say **"} expected"**, meaning a closing brace is missing.
4. **Check how many times the error occurred**, as some issues repeat every frame.

By following these steps, you can **quickly identify and fix** the problem in your script.

Visual Studio Disconnecting

Sometimes, **Visual Studio disconnects from Unity**, causing problems like:

1. **Auto-complete breaking**—you no longer see Unity-specific classes or functions.
2. **using UnityEngine; is grayed out**, meaning it's not being recognized.

The **simple fix**:

1. **Fully exit** Visual Studio.
2. **Wait a moment** to let the computer refresh.
3. **Reopen** any script from Unity.

This should **reconnect Visual Studio to Unity**, restoring full functionality.

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

Variables

A Variable

A **variable** consists of three main parts:

1. **Data Type** – Defines what kind of information the variable stores. In this case string.
2. **Variable Name** – This is how both you and the computer identify the variable.
3. **Value** – The actual data stored inside the variable. For instance, this string variable holds "Hello World". Values can be changed unless specified otherwise.

Remember, **two variables cannot share the same name** within the same scope, or you'll get an error.

```
string String = "Hello World";
```

Data Types

There are many data types you'll use when creating scripts, but these are the most common:

- **Int (int)** – Stores whole numbers (integers). Great for keeping count of items, scores, or levels.
- **Float & Double (float, double)** – Store numbers with decimal places, useful for defining position, rotation, scale, and other precise values. We'll mostly use float.
- **Boolean (bool)** – Holds only two values: **true** or **false**. Useful for tracking states like whether a door is open or if a player has jumped.
- **Char (char)** – Stores a single character, such as 'A' or '9'. These can be combined to create strings.
- **String (string)** – A collection of characters used to store and display text, like "Hello, Player!".

```
int Integer = 1;
float Float = 1.23456789012345678901234567890f;
double Double = 1.23456789012345678901234567890d;
bool Boolean = true;
char Character = 'a';
string String = "Hello World";
```

Declaration and Assignment

In C#, **declaration** is when you create a variable, while **assignment** is when you give it a value.

For example:

Here, **variableOne** is only declared, while **variableTwo** is declared and assigned the value 5.

C# automatically assigns default values to declared variables. For instance, an int defaults to 0, so **variableOne** technically has an implicit = 0 even though we didn't specify it.

Once a variable has been assigned, you can overwrite its value using =

This allows variables to store and update data dynamically as your program runs.

```
int variableOne;
int variableTwo = 5;

// Start is called before the first frame update
[Unity Message | 0 references]
void Start()
{
    variableOne = 5;
    print(variableOne + variableTwo);
}
```


Variable Scope

Scope determines where a variable can be accessed within your script.

Global variables are declared **inside** the **class** but **outside** of any **method**. These can be accessed and modified by any method in the class.

Local variables, on the other hand, are declared **inside** a **method** and are only accessible within that method. If a variable is created inside the **Start()** method, the **Update()** method wouldn't be able to access it.

Since local variables only exist within the method they are declared in, trying to use them elsewhere will result in an error.

```
int variableOne = 4;

// Start is called before the first frame update
Unity Message | 0 references
void Start()
{
    int variableTwo = variableOne + 5;
    print(variableTwo);
}
```

Challenge Variable and Scope

Now, let's put your skills to the test!

Create a global variable that has the value of “Hello “ with the space.


Inside of the Start() function create a second variable that says “World” .

Print the two variable with the result of Hello World.



[12:43:41] Hello World
UnityEngine.Debug:Log (object)

Answer

```
string varOne = "Hello ";  
  
// Start is called before the first frame update  
 Unity Message | 0 references  
void Start()  
{  
    string varTwo = "World";  
    Debug.Log(varOne + varTwo);  
}
```

Public Vs Private Variables

Just like scope determines where a variable can be accessed within a script, the **public** and **private** keywords define whether a variable can be accessed by other scripts.

Public variables can be accessed and modified by other scripts and are also visible in the **Inspector View** in Unity. This allows you to change their values directly without modifying the script.

Private variables, on the other hand, can only be accessed and modified within the same script. They are not visible in the Inspector and are controlled only by the internal logic of the script.

If you do not specify **public** or **private**, Unity automatically sets the variable to **private**.

However, these keywords can only be applied to **global variables**, local variables inside methods do not use them.

```
public int varOne = 1;  
private int varTwo = 2;
```



Public Inspector Value

If you make a **public variable** and later decide to change its **initial value** in the script, you'll need to manually update it in the **Inspector**.

When you first attach a script to a game object, Unity **stores** the public variable values as they were at that moment. This means that even if you **change the value in the script**, the live game object will still use the old value saved in the Inspector.

To update it, you can either:

1. Manually change the value in the **Inspector**.
2. **Reset** the component by right-clicking the script in the Inspector and selecting **Reset**, which will apply the updated default values from the script.



```
public int varOne = 1421;
```

Naming Conventions

There is a **naming convention** that helps make your code more readable and organized.

1. **Classes and Functions** always start with a **capital letter** (e.g., PlayerController, MoveCharacter()).
2. **Variables** typically start with a **lowercase letter** (e.g., playerSpeed).
3. When using **multiple words** in a name, use **camel casing**, (e.g., playerHealth, PlayerCntrroller). This naming style looks like **camel humps**, making it easier to read.
4. **Private variables** often begin with an **underscore (_) to indicate they should not be accessed outside the script** (e.g., _speed, _isJumping), while **public variables** don't use an underscore (e.g., playerName).

```
public class GeoController : MonoBehaviour
{
    public int varOne = 1421;
    private int _varTwo = 2;
    ⚙ Unity Message | 0 references
    private void OnTriggerEnter2D(Collider2D collision)
    {
        //
    }
}
```

Commenting

Just like keeping your **Project** and **Hierarchy Views** organized, maintaining **clean and structured code** is just as important.

Whether you're working in a **team** or on your **own**, over time, you'll forget the details of your code. Good **documentation** helps you (or your teammates) quickly understand what each part of the script does.

A personal **organization style** can make a huge difference. Here are some common practices:

- **Use Comments (// or /* */) to explain the purpose** of your code.
- **Group related variables and methods** together for better readability.
- **Use clear and descriptive names** for variables, functions, and classes.
- **Separate sections of your code** with spacing and headers (e.g., // Movement Variables, // Player Input Handling).

```
//=====
// Variables
//=====
int variableOne = 0;
int variableTwo = 1;
int variableThree = 2;

//=====
// Base Methods
//=====

// Start is called before the first frame update
[Unity Message | 0 references]
void Start()
{
}

// Update is called once per frame
[Unity Message | 0 references]
void Update()
{
}

//=====
// Custom Methods
//=====

/*
 * Purpose: Prints out the Hello World Statment
 * Parameters: N/A
 * Return: N/A
 */
[0 references]
void PrintOutStatment()
{
    print("Hello World");
}
```

Modifying Variables

Arithmetic Operators

Your programs will need to **perform math operations** regularly, whether it's for **moving a player, adjusting health, or tracking collectibles**.

You're already familiar with the four basic arithmetic operations:

- **Addition (+)**
- **Subtraction (-)**
- **Multiplication (*)**
- **Division (/)**

```
// Start is called before the first frame update
// Unity Message | 0 references
void Start()
{
    int variableOne;
    int variableTwo = 5;

    //Adds the value of variableTwo and 5 and assigns the value to variableOne
    variableOne = variableTwo + 5;
    print(variableOne);

    //Subtracts the value of variableTwo and 5 and assigns the value to variableOne
    variableOne = variableTwo - 5;
    print(variableOne);

    //Multiplies the value of variableTwo and 5 and assigns the value to variableOne
    variableOne = variableTwo * 5;
    print(variableOne);

    //Divides the value of variableTwo and 5 and assigns the value to variableOne
    variableOne = variableTwo / 5;
    print(variableOne);

    //Modules the value of variable Two over 5, module asks how many times does the second num
    //Here the result is 0 as 5 fits into 5, one time without leaving 0 as reminder. Think of
    variableOne = variableTwo % 5;
    print(variableOne);

    //Here the result is 1 as 2 fits into 5, two times leaving 1 as remidner. Think of it as 5
    variableOne = variableTwo % 2;
    print(variableOne);
}
```

Assignment Operators

Sometimes, you need to **quickly update** the value of a variable. Instead of writing:

```
variable = variable + 2;
```

You can **condense** it using **assignment operators**:

- `variable += 2;` (adds 2 to the current value)
- `variable -= 2;` (subtracts 2 from the current value)
- `variable *= 2;` (multiplies the current value by 2)
- `variable /= 2;` (divides the current value by 2)

For **incrementing or decrementing by 1**, you can use even **shorter** notations:

- `variable++;` (adds 1 to the variable)
- `variable--;` (subtracts 1 from the variable)

```
int variableOne = 0;
int variableTwo = 5;

//Sometimes you want to change the value of a variable by itself and some other variable
variableOne = variableOne + variableTwo;
print(variableOne);

//To do this there is a shorthand that simplifies the code, changing variableOne + to +=, -=, *=
variableOne += variableTwo;
print(variableOne);

variableOne -= variableTwo;
print(variableOne);

variableOne *= variableTwo;
print(variableOne);

variableOne /= variableTwo;
print(variableOne);

//A lot of times you will want to increase/decrease a number by one for counting purpose
variableOne = variableOne + 1;
print(variableOne);

//For that the ++ or -- is shorthand for increase/decrease by one
variableOne++;
print(variableOne);

variableOne--;
print(variableOne);
```

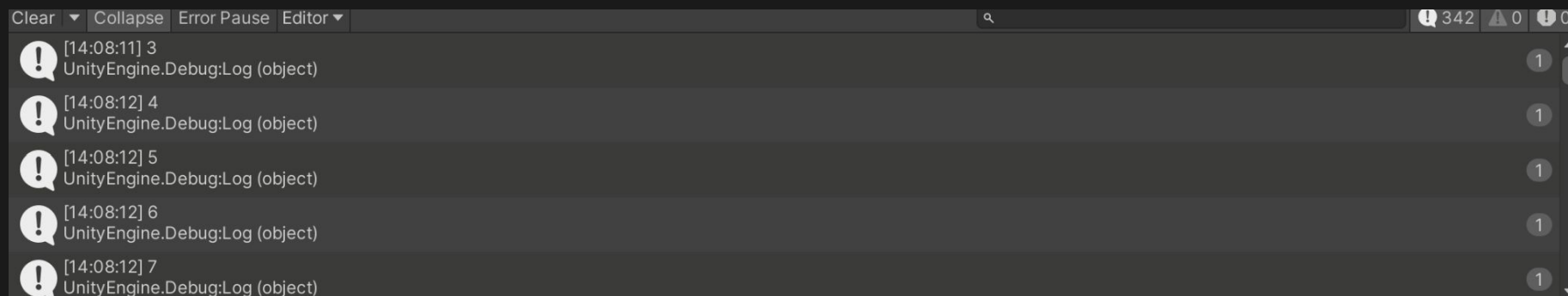
Challenge Arithmetic and Assignment

Now, let's put your skills to the test!

Let's create a global integer variable and give a value of 3.

Then inside Update function print it out using the Debug

Then add plus one. Let it run you should have this result.



Answer

```
int var = 3;

// Update is called once per frame
📦 Unity Message | 0 references
void Update()
{
    Debug.Log(var);
    var++;
}
```

Transform

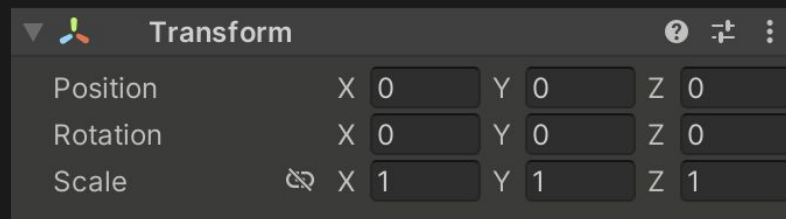
Transform

As we've discussed before, **Transform** is a component that every game object has.

Every **component** we attach to a game object is essentially a script, and we can communicate with these scripts in different ways.

If we want to access or modify the **Transform**, we can simply write **transform** and access any of its **variables**

We can **directly access transform** because **every game object has exactly one Transform component**, making it immediately available for use.



```
// Start is called before the first frame update
Unity Message | 0 references
void Start()
{
    Debug.Log(transform.position);
}
```



[14:14:03] (0.00, 0.00, 0.00)
UnityEngine.Debug:Log (object)

Vector Class

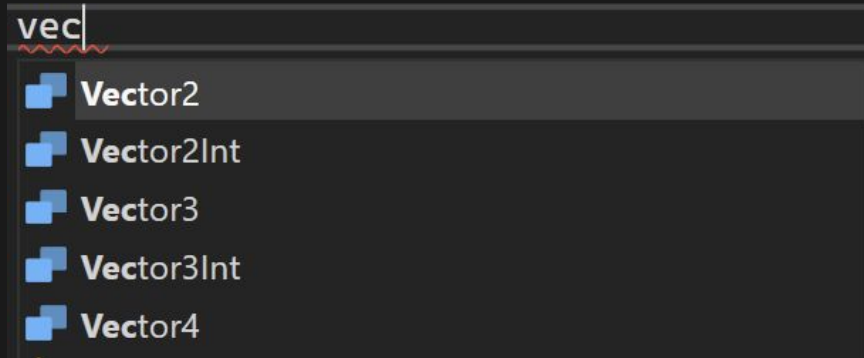
Vector Classes are collections of **2, 3, or 4 variables** stored together to represent **positions, directions, or scales** in a game.

- **Vector2** is used for **2D** values (X, Y), such as speed or UI positions.
- **Vector3** is used for **3D** values (X, Y, Z), which is what we mainly use when updating **Transform** in Unity.

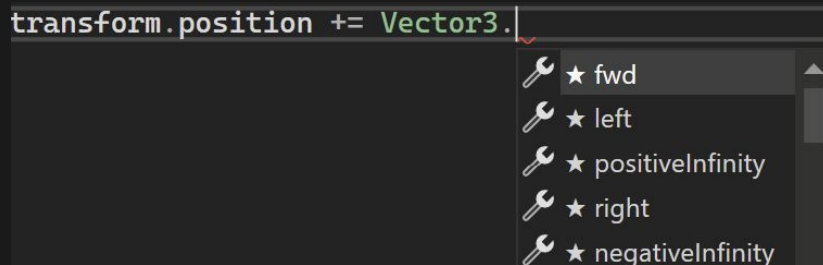
Since our **Transform** component has an **X, Y, Z location**, we need to provide movement input in that format.

This is equivalent to:
 $(0,0,0) + (-1,0,0) = (-1,0,0)$

So, if the player starts at **(0,0,0)** and we apply **Vector3.left**, the new position will be **(-1,0,0)**, effectively moving the player **one unit to the left**.



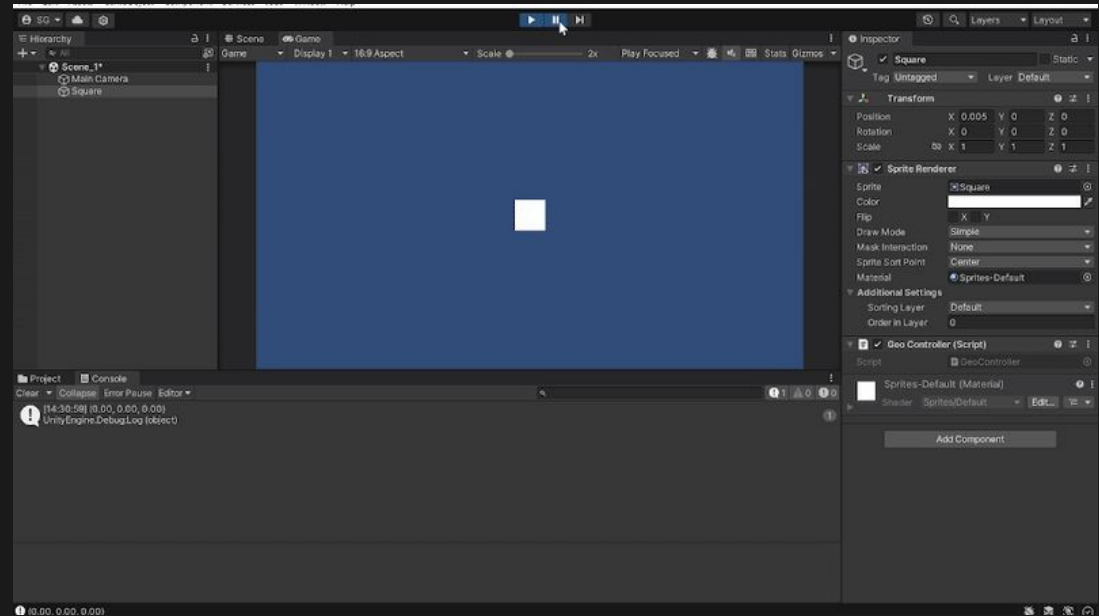
```
Vector3 vector3 = new Vector3(1, 1, 1);
```



Challenge Move Player

Now, let's put your skills to the test!

In your update function add a vector of `Vector3(0.005f, 0, 0)` to the transform.



Answer

```
// Update is called once per frame
Unity Message | 0 references
void Update()
{
    transform.position += new Vector3(0.005f, 0, 0);
}
```

Booleans & Player Input

If Statement

To move the player based on **inputs**, we first need to check **if an input is occurring**. This is done using an **if statement**, which allows the program to **branch** and behave differently based on a condition.

An **if statement** checks if something is **true** or **false**. If the condition inside the parentheses **evaluates to true**, the code inside the `{ }` **executes**. Otherwise, it **skips** that block.

Since `1 > 0` is **always true**, the first message will print, while the second one will **never execute**.

We will use **if statements** to check for **player input** and move the character accordingly.

```
if( 1 > 0 )
{
    Debug.Log("1 is bigger than 0");
}
else
{
    Debug.Log("1 is not bigger than 0");
}
```

If Statement

Additionally, **else if statements** allow for multiple conditional checks, ensuring that only the first true condition is executed.

```
//Checks if any of the statements are true, the first true statement gets executed.  
if (variable == 6)  
{  
    print("Statment Three: If");  
}  
else if (variable <= 6)  
{  
    print("Statment Three: Else If One");  
}  
else if(variable < 6)  
{  
    print("Statment Three: Else If Two");  
}  
else  
{  
    print("Statment Three: Else");  
}
```

Ternary Conditional Operator

```
//Assign variable the value of either 10 or -10 depending on if variable is 10 < 0.  
variable = variable < 0 ? 10 : -10;  
print(variable);
```

The **ternary operator** is a shorthand for an **if-else statement**, allowing value assignment based on a condition.

It follows the format: **condition ? value_if_true : value_if_false**.

The **question mark (?)** separates the condition from the true case, while the **colon (:)** separates the true and false cases.

This approach simplifies code by reducing the need for multiple lines. It is especially useful for quick conditional assignments.

Equality Operators

Equality operators **compare** two values and return either **true** or **false**.

Since `=` is used for **assignment**, we use `==` to check if **two values are equal**.

Similarly, we use `!=` to check if **two values are not equal**.

Other comparison operators:

- `>` (greater than)
- `<` (less than)
- `>=` (greater than or equal to)
- `<=` (less than or equal to)

```
Unity Script | 0 references
public class IfStatments: MonoBehaviour
{
    // Start is called before the first frame update
    Unity Message | 0 references
    void Start()
    {
        //Checks if the two sides are equal
        print(6 == 7); //False
        print(6 == 6); //True

        //Checks if the two side are not equal
        print(6 != 7); //True
        print(6 != 6); //False

        //Checks if one side is large than the other
        print(5 > 3); //True
        print(5 < 3); //False
        print(3 > 3); //Fasle

        print(5 >= 3); //True
        print(5 <= 3); //False
        print(3 >= 3); //True
    }
}
```

Logic Operators

Logical operators allow you to combine multiple conditions in a single statement.

- **&& (AND)** → Both conditions **must be true** for the result to be true. If **any condition is false**, the whole statement is false.
- **|| (OR)** → If **at least one** condition is true, the whole statement is true. If **both are false**, the statement is false.

```
// Start is called before the first frame update
@ Unity Message | 0 references
void Start()
{
    // ! Invertes the boolean value
    print(!true);    //False
    print(!false);   //True
    print(!(6 == 6)); //False

    //Asks if ALL statments are true, if either is false the result is false
    print(true && true); //True
    print(true && false); //False
    print(false && true); //False
    print(false && false); //False

    //Asks if ONE of thes statments is strue, if it is result is true
    print(true || true); //True
    print(true || false); //True
    print(false || true); //True
    print(false || false); //False

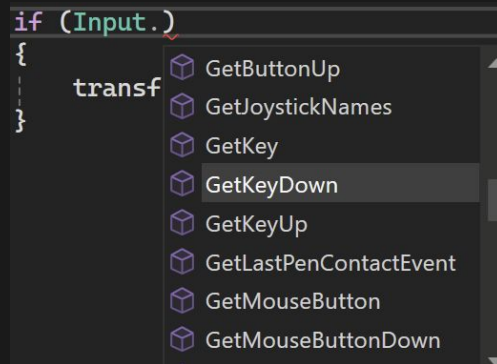
    //True    True    False
    print((6 == 6) && true || (7 < 3)); // True
}
```

Player Input

To detect player input in Unity, we use the **Input** class, which lets us check for different types of key interactions:

- `Input.GetKeyDown(KeyCode.W)`: Detects when the **W key is pressed once**.
- `Input.GetKey(KeyCode.W)`: Detects when the **W key is being held down continuously**.

Then the `KeyCode` select the key on the keyboard, in this case **W key** is pressed and moves the object **up on the Y-axis by 1 unit**.

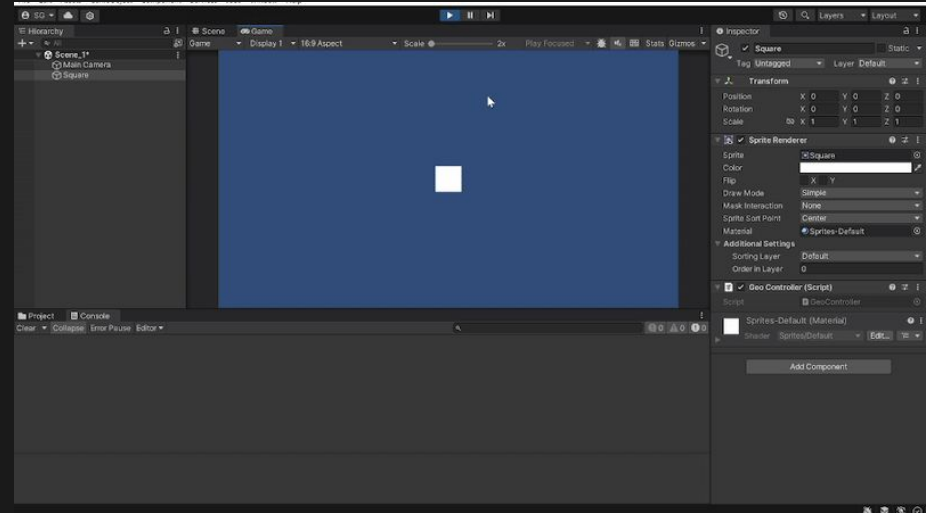


```
// Update is called once per frame
@ Unity Message | 0 references
void Update()
{
    if (Input.GetKeyDown(KeyCode.W))
    {
        transform.position += new Vector3(0, 1, 0);
    }
}
```


Challenge Inputs for the Player

Now, let's put your skills to the test!

Let follow the previous example and connect WASD to let the player move Up Left Down Right.



Answer

```
// Update is called once per frame
```

```
📦 Unity Message | 0 references
```

```
void Update()
```

```
{
```

```
    if (Input.GetKeyDown(KeyCode.W))
```

```
    {
```

```
        transform.position += new Vector3(0, 1, 0);
```

```
    }
```

```
    if (Input.GetKeyDown(KeyCode.S))
```

```
    {
```

```
        transform.position += new Vector3(0, -1, 0);
```

```
    }
```

```
    if (Input.GetKeyDown(KeyCode.A))
```

```
    {
```

```
        transform.position += new Vector3(-1, 0, 0);
```

```
    }
```

```
    if (Input.GetKeyDown(KeyCode.D))
```

```
    {
```

```
        transform.position += new Vector3(1, 0, 0);
```

```
    }
```

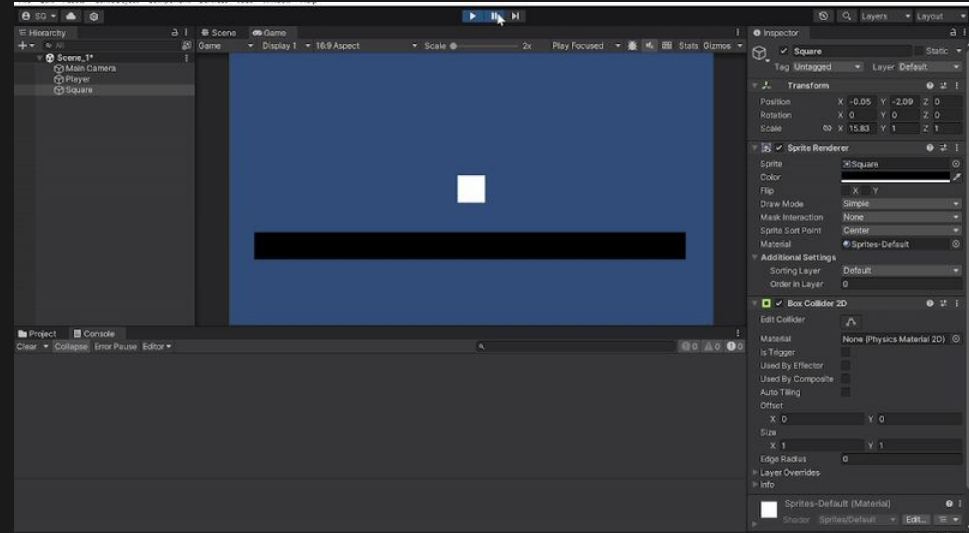
```
}
```

Challenge Geo Quest Set Up

Now, let's put your skills to the test!

Rename our Square to Player,
add a Box Collider and
Rigidbody to it

Create another square and add
box collider to it.



Communicating Between Scripts

GetComponent

Now that we have a Rigidbody, we will want to utilize it within the transform to move our player while maintaining the momentum and speed provided by the physics system.

However, unlike the transform component, Rigidbody is not automatically included in all game objects. Therefore, we need to create a variable and establish a connection to it.

Fortunately, if the script and the Rigidbody component are attached to the same game object, we can use the GetComponent method to retrieve that information:

```
GetComponent<Rigidbody2D>();
```

The `<Rigidbody2D>` specifies the type of component we are looking for.

```
private Rigidbody2D rb;  
  
// Start is called before the first frame update  
[Unity Message | 0 references]  
void Start()  
{  
    rb = GetComponent<Rigidbody2D>();  
}
```

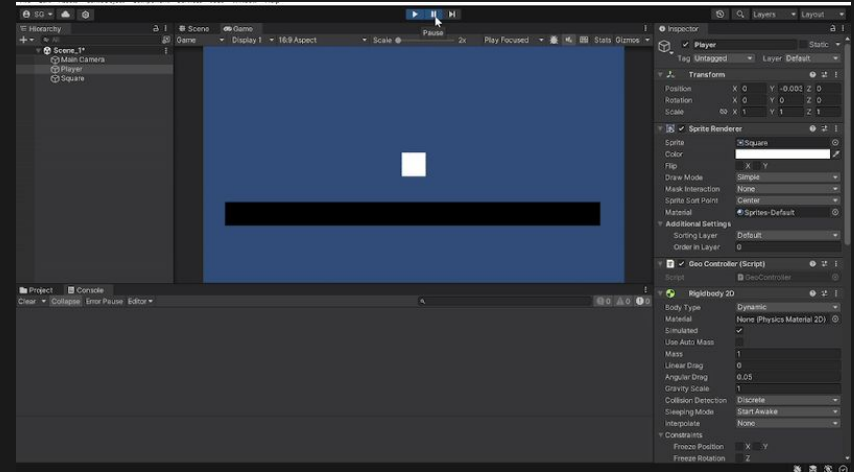


Moving By Rigidbody

Now that we have established a connection to the Rigidbody, we can define how we want it to move.

Instead of modifying the position directly, we will be adjusting the **velocity**. This means we are controlling the speed in the **X** and **Y** directions while the Rigidbody communicates with the **transform** to ensure all physics interactions remain consistent.

In this case, we move the object to the **left**. However, you may notice that the block does not fall as quickly as expected. This is because we are setting the **velocity** to **(-1, 0)**, meaning gravity starts pulling it down from zero, causing it to gradually glide downward.

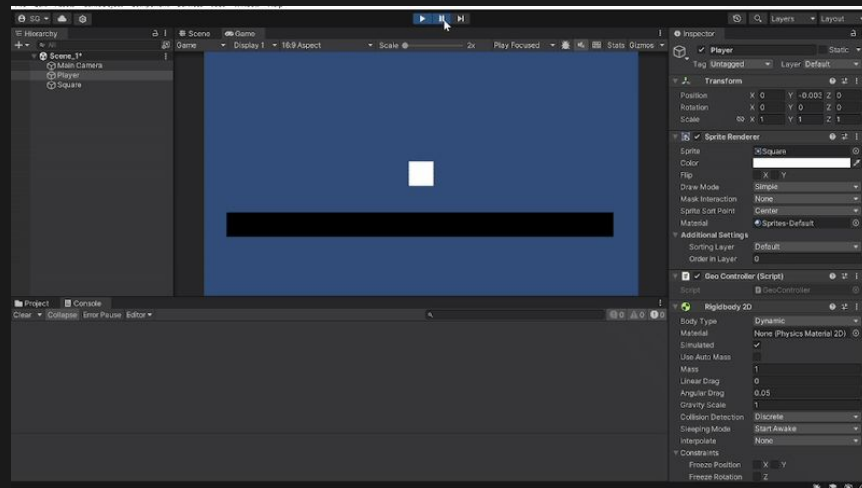


```
// Update is called once per frame
// Unity Message | 0 references
void Update()
{
    rb.velocity = Vector2.left;
}
```

Moving Rigidbody in One Direction

In this case, we update the velocity so that only the **X** direction is affected while **preserving the Y** value calculated by the Rigidbody.

As you can see, velocity is a variable that contains additional **sub-variables**, allowing us to access the X and Y values to determine the speed the Rigidbody is currently experiencing.

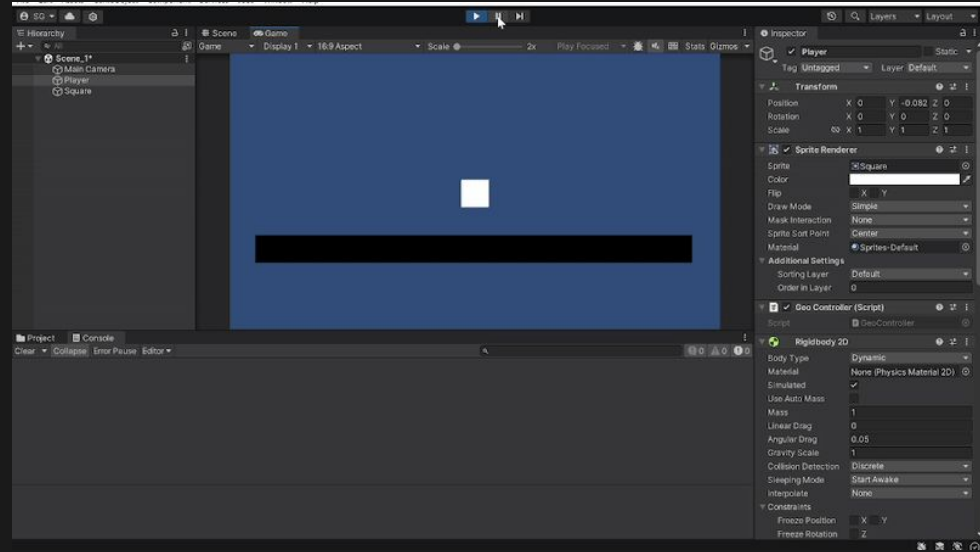


```
// Update is called once per frame
Unity Message | 0 references
void Update()
{
    rb.velocity = new Vector2(-1, rb.velocity.y);
}
```

Challenge Move the Rigidbody

Now, let's put your skills to the test!

Let's follow our example from earlier and give the player input to move the player left and right using the rigidbody



Answer

```
// Update is called once per frame
```

```
Unity Message | 0 references
```

```
void Update()
```

```
{
```

```
    if (Input.GetKeyDown(KeyCode.A))
```

```
    {
```

```
        rb.velocity = new Vector2(-1, rb.velocity.y);
```

```
    }
```

```
    if (Input.GetKeyDown(KeyCode.D))
```

```
    {
```

```
        rb.velocity = new Vector2(1, rb.velocity.y);
```

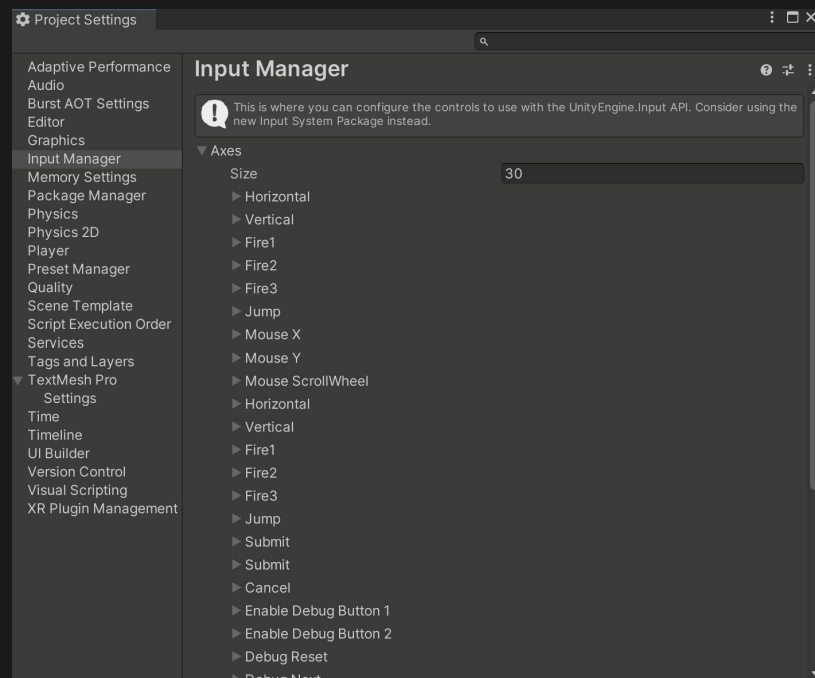
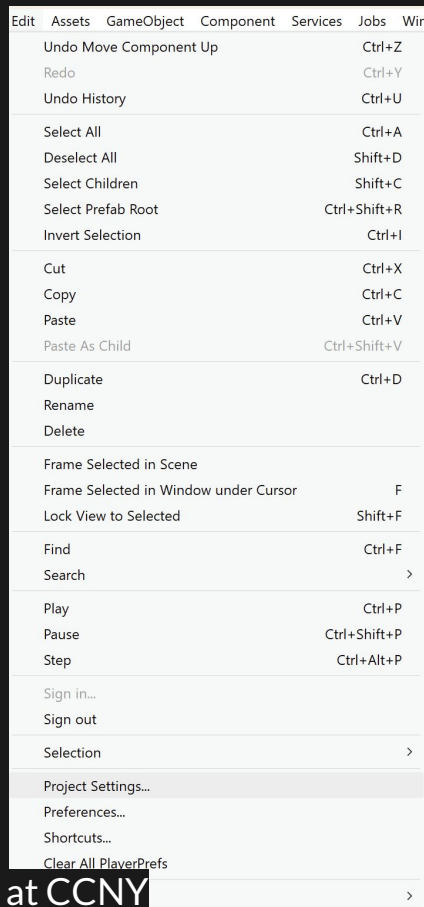
```
    }
```

```
}
```

Input Manager

Unity provides a way to simplify input handling through the **Input Manager**, a specialized system that allows you to assign multiple keys to different actions. It also enables you to define values for different keys, with the absence of input registering as 0.

You can access the **Input Manager** by navigating to **Edit** → **Project Settings**, then selecting the **Input Manager** tab.



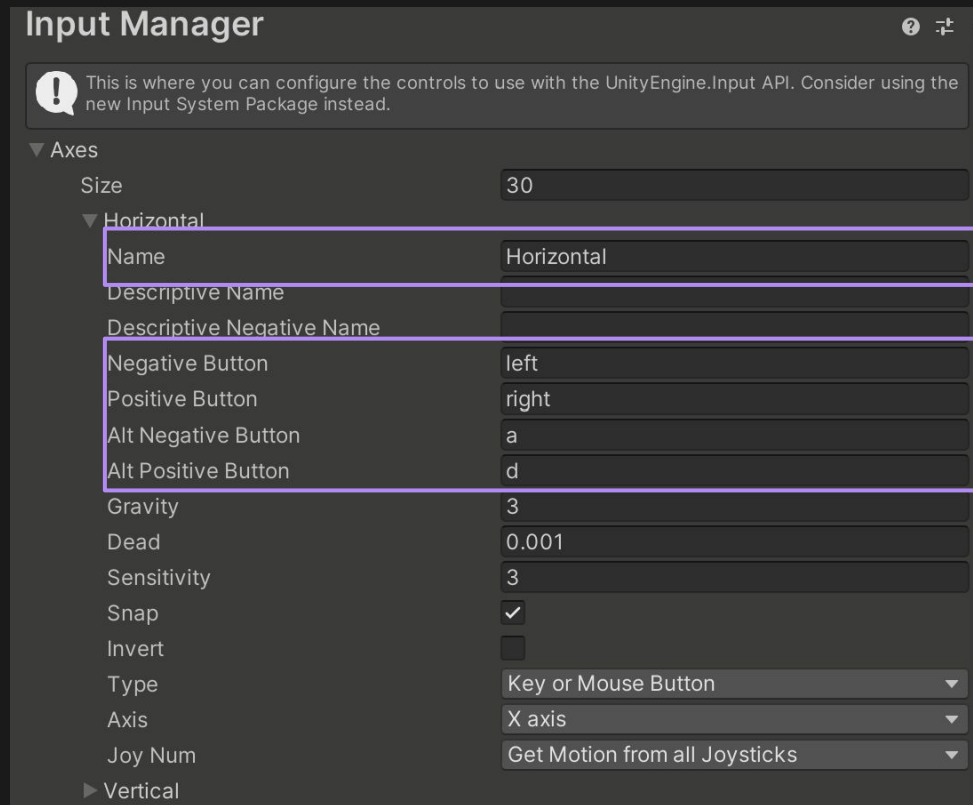
Input Manager

We will expand the **Horizontal** axis in the **Input Manager** to examine its settings.

- **Name** is crucial because it is the identifier we will use in our script to communicate with this input.
- **Buttons** define which keys correspond to this input. The **Negative Buttons** are set to **A** and **Left Arrow**, while the **Positive Buttons** are set to **D** and **Right Arrow**.

This setup means:

- If the player presses **A** or **Left Arrow**, the input will return **-1**.
- If the player presses **D** or **Right Arrow**, the input will return **1**.
- If no key is pressed, the input will return **0**



Get Axis

By using

`Input.GetAxis("Horizontal")`, we can retrieve the values **-1**, **0**, or **1** based on the player's key inputs. This allows us to detect movement direction dynamically.

As we press the assigned buttons, we can observe the input results updating accordingly.

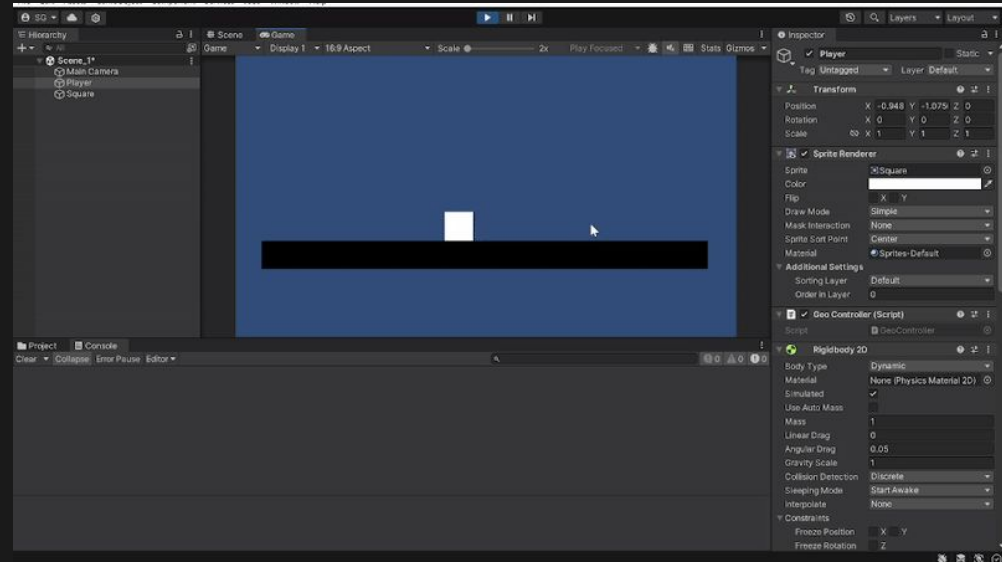


```
// Update is called once per frame
Unity Message | 0 references
void Update()
{
    float xInput = Input.GetAxis("Horizontal");
    Debug.Log(xInput);
}
```

Challenge: Move Player Using Input Manager

Now, let's put your skills to the test!

Apply the `xInput` variable to move the player using rigidbody make sure to keep the y velocity as is.



Answer

```
// Update is called once per frame
```

```
📦 Unity Message | 0 references
```

```
void Update()
```

```
{
```

```
    float xInput = Input.GetAxis("Horizontal");
```

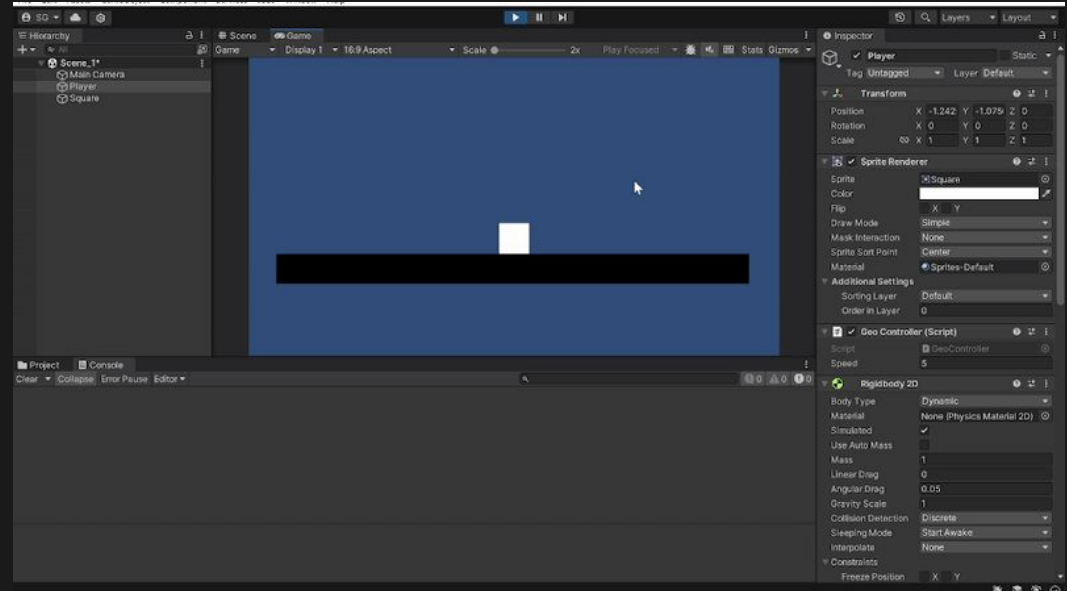
```
    rb.velocity = new Vector2 (xInput, rb.velocity.y);
```

```
}
```

Challenge: Speed Up Player


Now, let's put your skills to the test!


Our player is very slow, create a global public int variable that modify the xInput speed without affect the y speed.



Answer

```
public class GeoController : MonoBehaviour
{
    private Rigidbody2D rb;
    public int speed = 5;

    // Start is called before the first frame update
     Unity Message | 0 references
    void Start()
    {
        rb = GetComponent<Rigidbody2D>();
    }

    // Update is called once per frame
     Unity Message | 0 references
    void Update()
    {
        float xInput = Input.GetAxis("Horizontal");
        rb.velocity = new Vector2 (xInput * speed, rb.velocity.y);
    }
}
```

Detecting Collision

Methods - Functions - Anatomy

Just like variables, functions can be **private** or **public**. An example of a public function is **Log()**.

After defining the access level, we have the **return type**, which specifies what the function will give back after execution. So far, all the functions we've worked with use **void**, meaning they do not return anything. However, if a function is defined with **int**, it must return an integer, as shown in the **GetNumberFour** function.

Lastly, functions can take **parameters**, which are pieces of information passed into them. In **PrintGivenNumber**, the function requires an input value to execute properly. This is exactly how **Log()** works, when we call **Log("Hello World")**, it takes the string input and prints it.

```
// Start is called before the first frame update
@ Unity Message | 0 references
public void Start()
{
    ...
}

0 references
private void PrintGivenNumber(int number)
{
    Debug.Log(number);
}

0 references
public int GetNumberFour()
{
    return 2 + 2;
}
```

OnTriggerEnter2D, OnTriggerStay2D, OnTriggerExit2D

Unity comes with several built-in functions that handle specific interactions, some of which are **OnTriggerEnter2D**, **OnTriggerStay2D**, and **OnTriggerExit2D**. These functions detect when an object **enters**, **stays inside**, or **exits** a 2D collider, allowing us to execute code based on those events.

It's important to use the **2D** versions of these functions (e.g., **OnTriggerEnter2D** instead of **OnTriggerEnter**) because Unity also has 3D versions that will not detect collisions with 2D colliders.

Unity Message | 0 references

```
private void OnTriggerEnter2D(Collider2D collision)
{
    ...
}
```

Unity Message | 0 references

```
private void OnTriggerStay2D(Collider2D collision)
{
    ...
}
```

Unity Message | 0 references

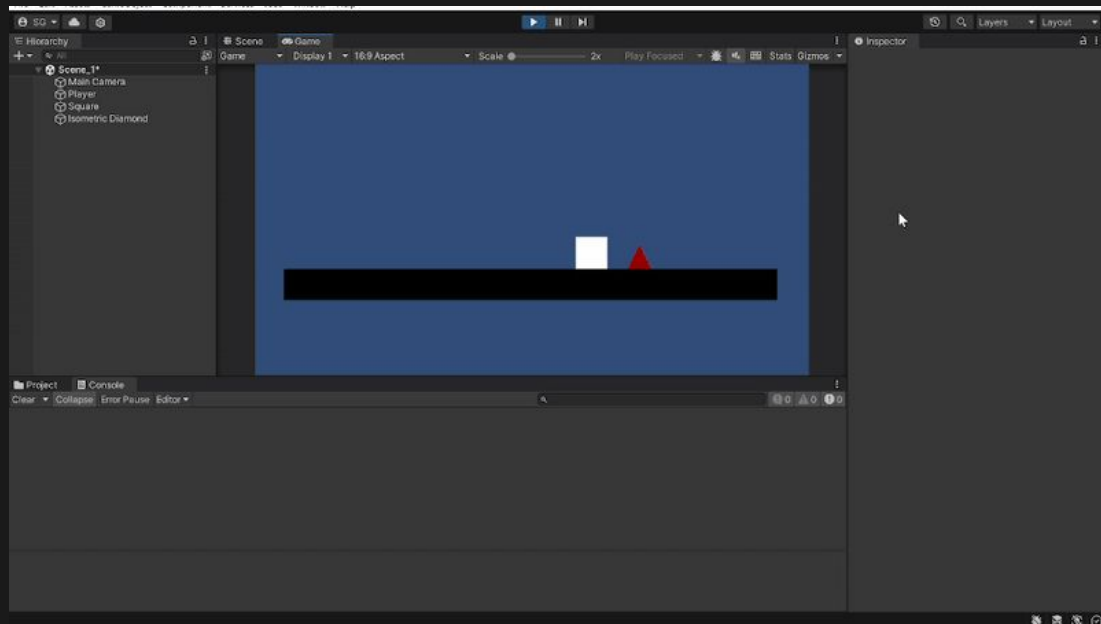
```
private void OnTriggerExit2D(Collider2D collision)
{
    ...
}
```

Challenge: Collision

Now, let's put your skills to the test!

Create a Red Isometric Diamond and give it a Polygon Collider 2D make sure it's triggered.

Then inside our GeoController create a OnTriggerEnter2D and print "Hit" when the player touches it.



Answer

Unity Message | 0 references

```
private void OnTriggerEnter2D(Collider2D collision)
{
    Debug.Log("Hit");
}
```

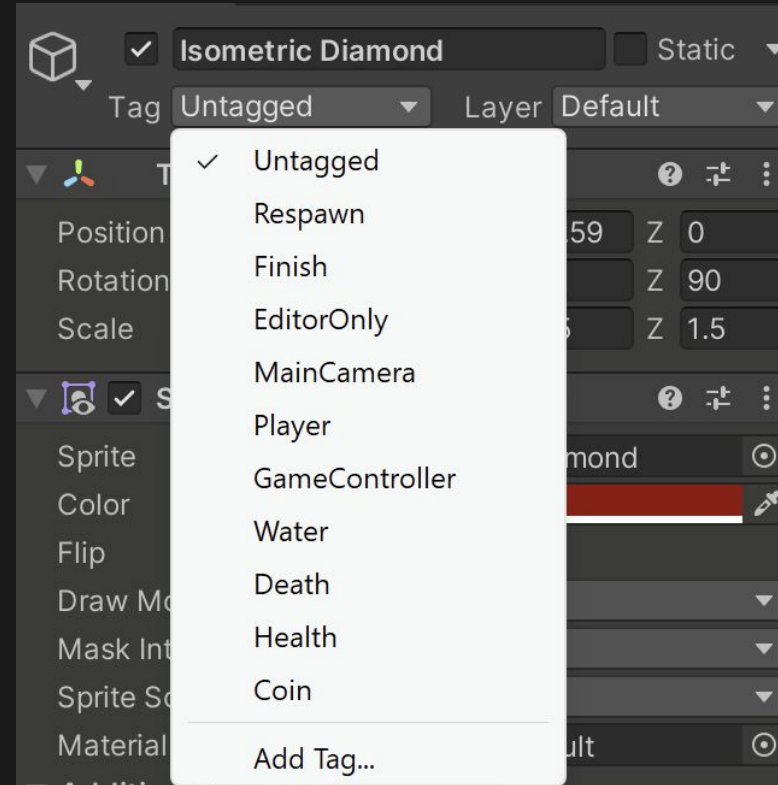
Tags

Now, we don't want the player to trigger a collision response with just any **triggerable** object, as there will be many different objects they can interact with.

To ensure that only specific objects, like hazards, cause a response, we use the **Tag system**.

Each **GameObject** in Unity can have a **Tag** assigned to it, which can be set in the **Inspector** under the object's **header**.

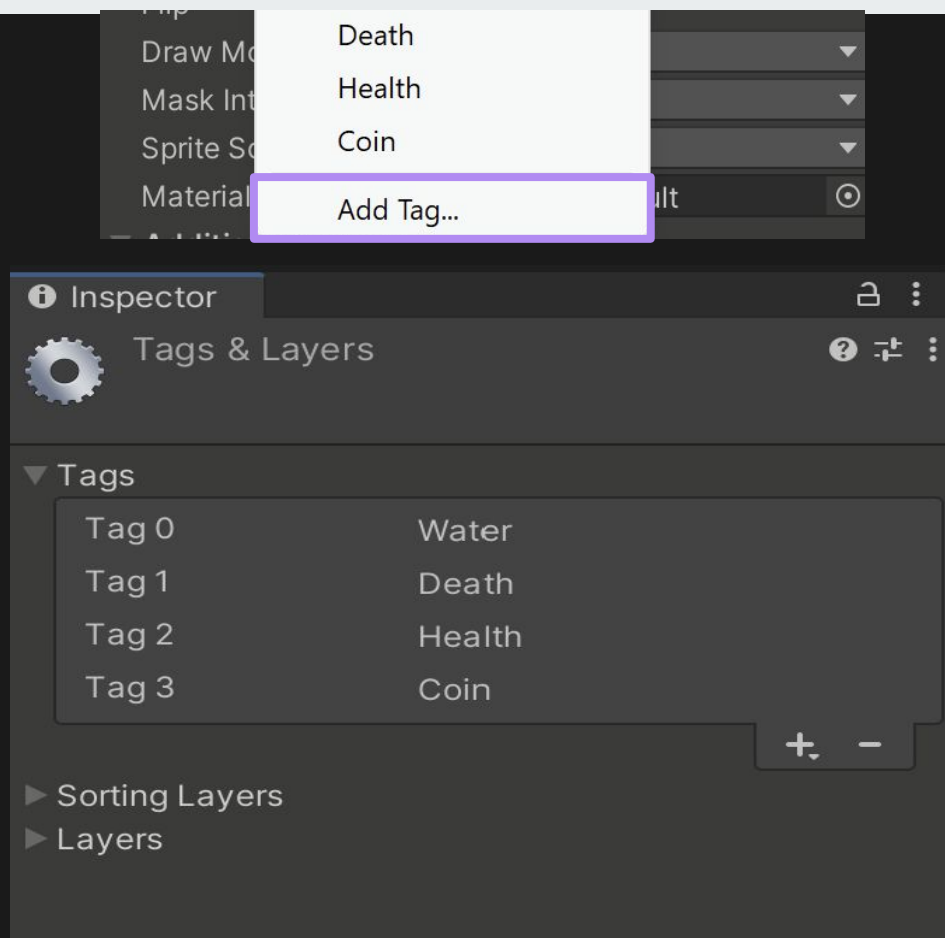
If you click on the **Tag** dropdown, you'll see several **pre-made tags**. For instance, we can use the **"Death"** tag to identify dangerous objects like spikes and enemies, ensuring only these objects trigger a response when the player collides with them.



Tag Inspector

If you want to create your own **Tags**, you can click on **"Add Tag"**, which will take you to the **Tags and Layers** window.

Here, you can add or remove custom tags that will be saved to this **project**. This allows you to categorize objects in a way that suits your game's needs, making it easier to identify and interact with them through scripts.



Collider2D collider

To retrieve the **Tag**, we can communicate with the **Collider** that was triggered during the event.

Since the **Collider** is linked to the **GameObject** it belongs to, we can simply access the **tag** directly from it. This allows us to check what kind of object we collided with and execute the appropriate response in our script.

```
Unity Message | 0 references  
private void OnTriggerEnter2D(Collider2D collision)  
{  
    Debug.Log(collision.tag);  
}
```



[16:17:28] Death
UnityEngine.Debug:Log (object)

Switch Statement

Now, we need a way to quickly test for different **tags**. While we could use **if statements**, a more efficient approach is to use a **switch statement**.

A **switch statement** takes one input and checks it against multiple **cases**. For example, if the tag is "Lose", the program will skip the "Win" case and execute the "Lose" case directly.

```
Only message 10 references
private void OnTriggerEnter2D(Collider2D collision)
{
    switch (collision.tag)
    {
        case "Win":
        {
            break;
        }
        case "Lose":
        {
            break;
        }
    }
}
```

Break

The **break** statement is used to indicate that a case is complete. Without it, execution would continue into the next case, even if it doesn't match the input.

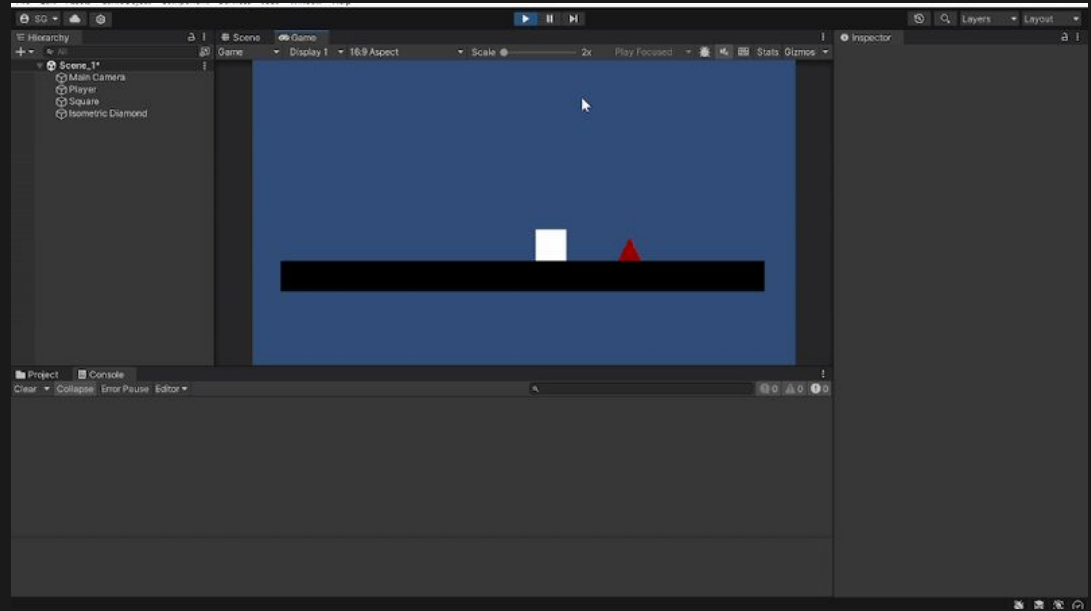
As shown here, multiple cases can roll into each other if no **break** is used. This means that unless specified, the code from previous cases might still execute in the following cases. To prevent this, always use **break** unless you intentionally want cases to share execution.

```
switch (collision.tag)
{
    case "Win":
    case "Winner":
    case "BestWinner":
        {
            break;
        }
    case "Lose":
        {
            break;
        }
}
```

Challenge: Player Has Died

Now, let's put your skills to the test!

Create a switch case for Death and when player has entered it print "Player Has Died"



Answer

Unity Message | 0 references

```
private void OnTriggerEnter2D(Collider2D collision)
{
    switch (collision.tag)
    {
        case "Death":
        {
            Debug.Log("Player Has Died");
            break;
        }
    }
}
```

Scene Loading

Scene Management and Reload

Now, we need to implement a consequence for the player touching a "Death" object by **resetting the level**.

To achieve this, we must include a specific library at the top of our script:

```
using UnityEngine.SceneManagement;
```

This library grants us access to scene-loading functionality.

To reload the level, we first obtain the name of the current scene using **SceneManager**. We do this by accessing the **Scene** object and retrieving its **name** variable.

After retrieving the scene name, we then use **SceneManager** to reload it, effectively resetting the level for the player.

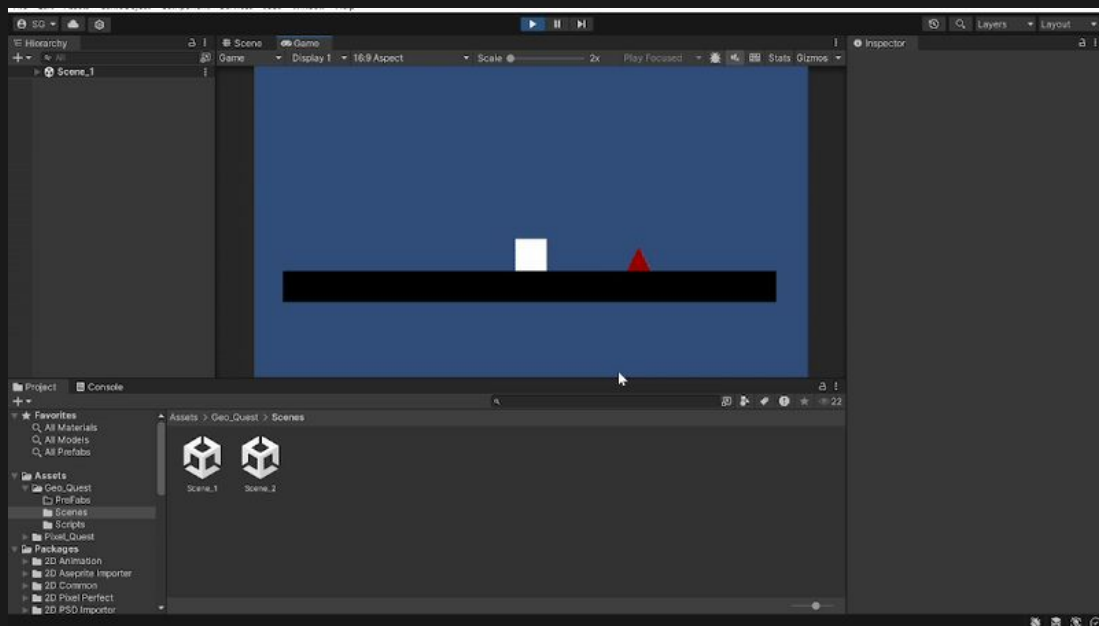
```
using UnityEngine.SceneManagement;
```

```
case "Death":  
    {  
        string thisLevel = SceneManager.GetActiveScene().name;  
        SceneManager.LoadScene(thisLevel);  
        break;  
    }  
}
```

Challenge Reload

Now, let's put your skills to the test!

Implement reload functionality we just covered.



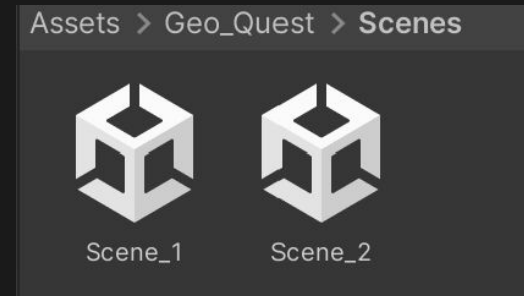
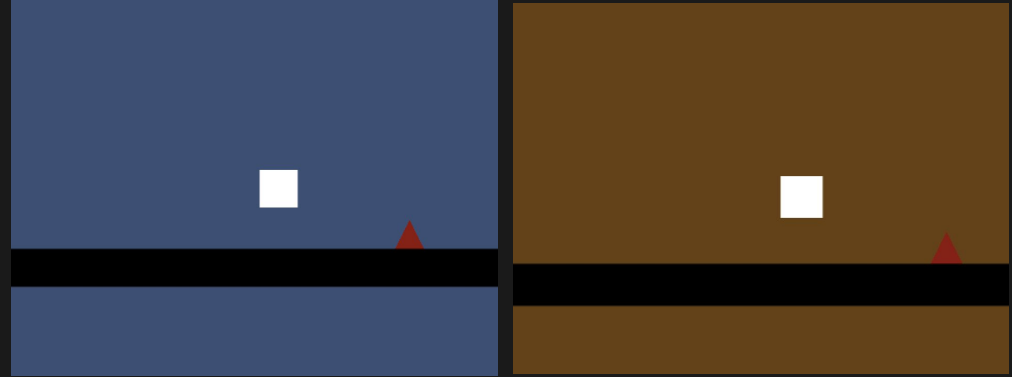
Challenge Create a Second Level

Now, let's put your skills to the test!

Save your current Scene.

Create a copy of the scene we've been working out of, (**Ctrl+C**, **Ctrl+V**)

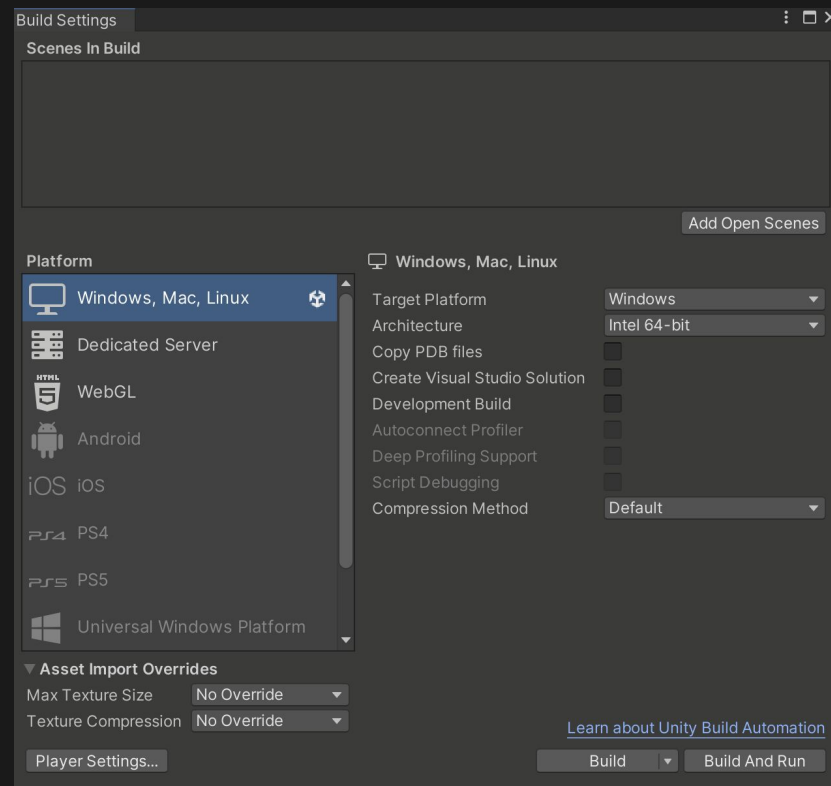
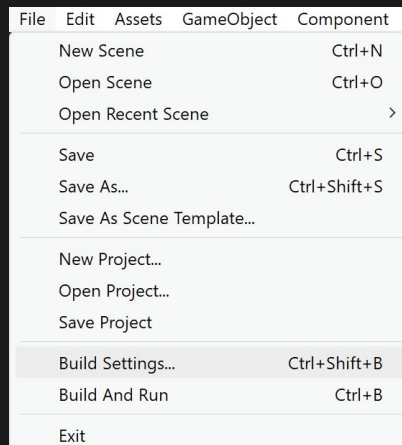
Change the name to Scene_2 and change the background color.



Build Settings

Before we can transition between scenes using code, we need to add them to the **Build Settings**. In Unity, go to **Edit** → **Build Settings** and locate the **Scenes In Build** section.

This section lists all scenes that will be included when the game is built and executed.

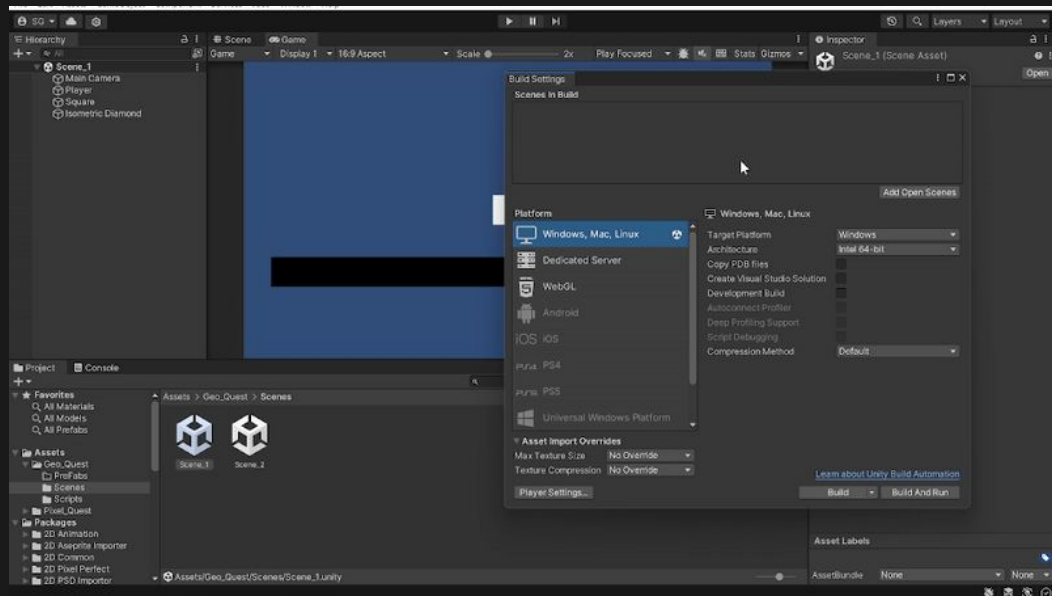


Adding Scene to Build Settings

To add a scene to the **Build Settings**, you have two options. You can either click the "**Add Open Scenes**" button to add the currently open scene or manually **drag** the scenes from your **Assets** → **Scenes** folder directly into the list.

One important detail to keep in mind is that the scene at the **top of the list** is the one that the game will load first when it starts.

If needed, you can rearrange the order by dragging scenes up or down to ensure the correct starting scene is selected.



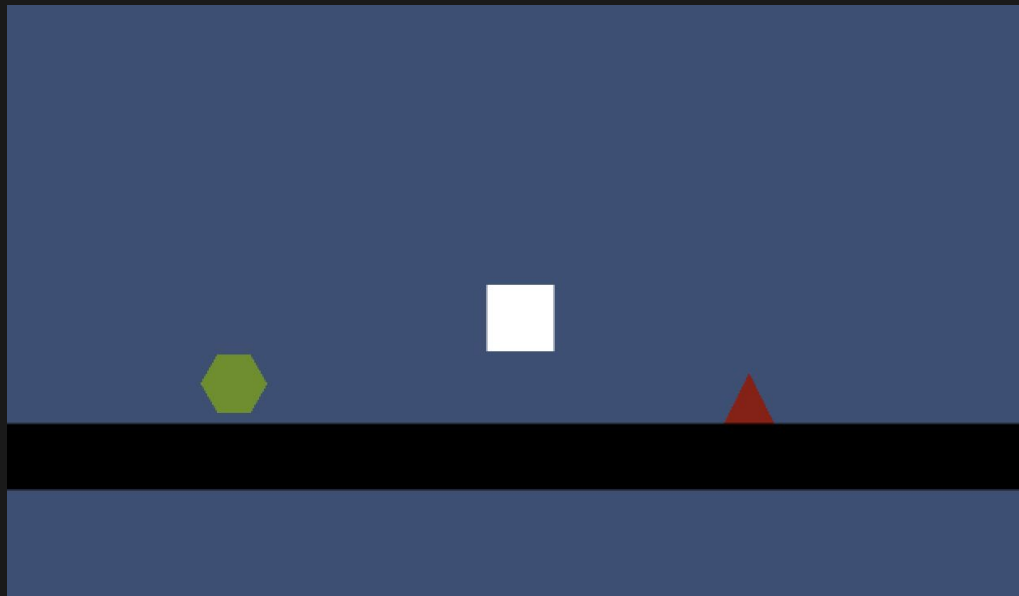
Challenge Win Object

Now, let's put your skills to the test!

Create a Hexagon Game Object

Add a Box Collider 2D that's set to is Triggered.

Tagged the Game Object with "Finish" tag.



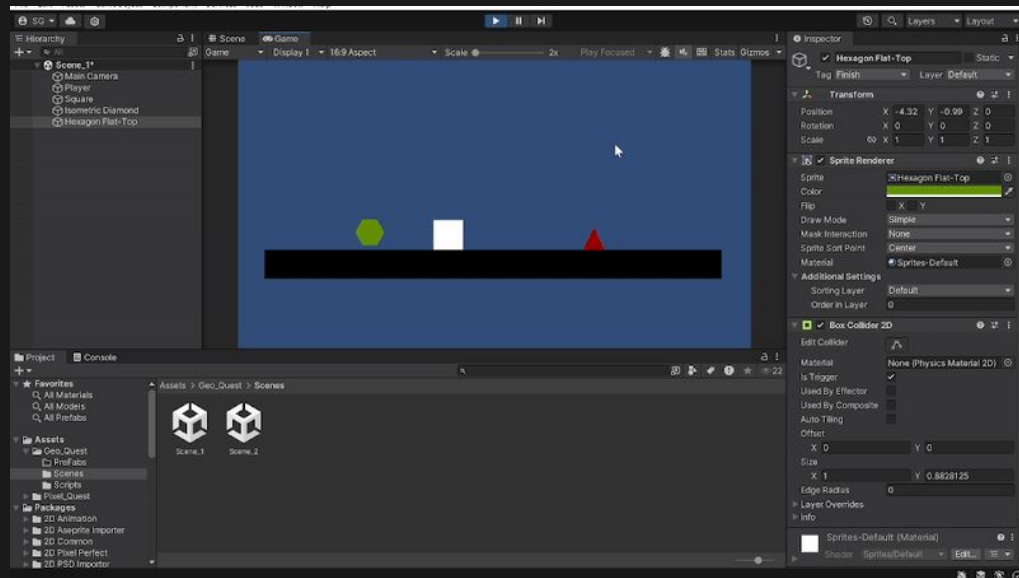
Challenge Win Object Code

Now, let's put your skills to the test!

Create a public global string variable that holds the name of the next level.

Create a new case in the switch statement for “Finish” tag.

In that case use SceneManager loads us into that level.



Answer

```
Unity Script (2 asset references) 1 0 references
public class GeoController : MonoBehaviour
{
    private Rigidbody2D rb;
    public int speed = 5;
    public string nextLevel = "Scene_2";

    // Start is called before the first frame update
    @ Unity Message | 0 references
    void Start()
    {
        rb = GetComponent<Rigidbody2D>();
    }

    // Update is called once per frame
    @ Unity Message | 0 references
    void Update()
    {
        float xInput = Input.GetAxis("Horizontal");
        rb.velocity = new Vector2 (xInput * speed, rb.velocity.y);
    }

    @ Unity Message | 0 references
    private void OnTriggerEnter2D(Collider2D collision)
    {
        switch (collision.tag)
        {
            case "Death":
            {
                string thisLevel = SceneManager.GetActiveScene().name;
                SceneManager.LoadScene(thisLevel);
                break;
            }
            case "Finish":
            {
                SceneManager.LoadScene(nextLevel);
                break;
            }
        }
    }
}
```

Challenge PreFabs

Now, let's put your skills to the test!

Create PreFabs of the Win and Spike Game Objects so we can use it in the creation of Geo Quest.

