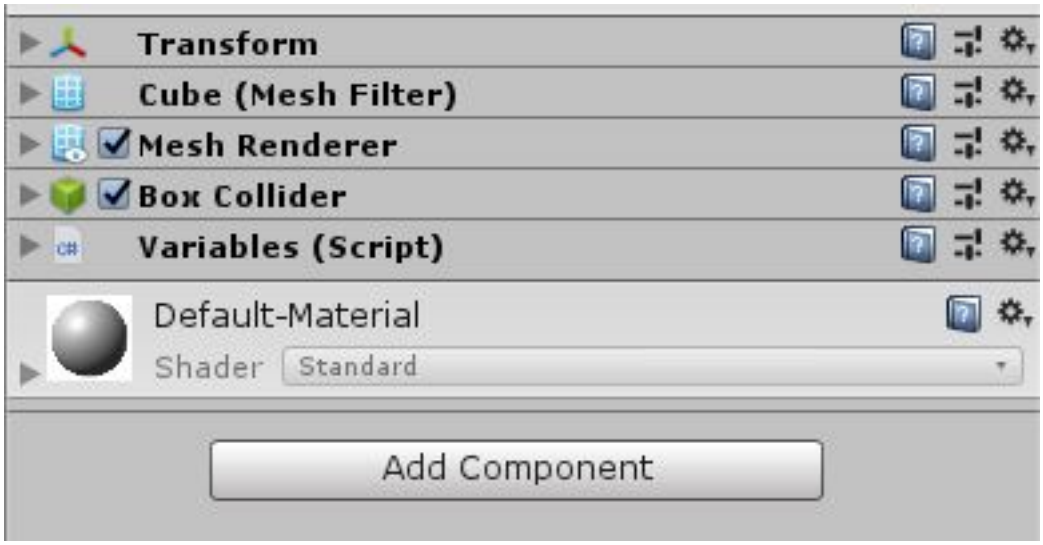
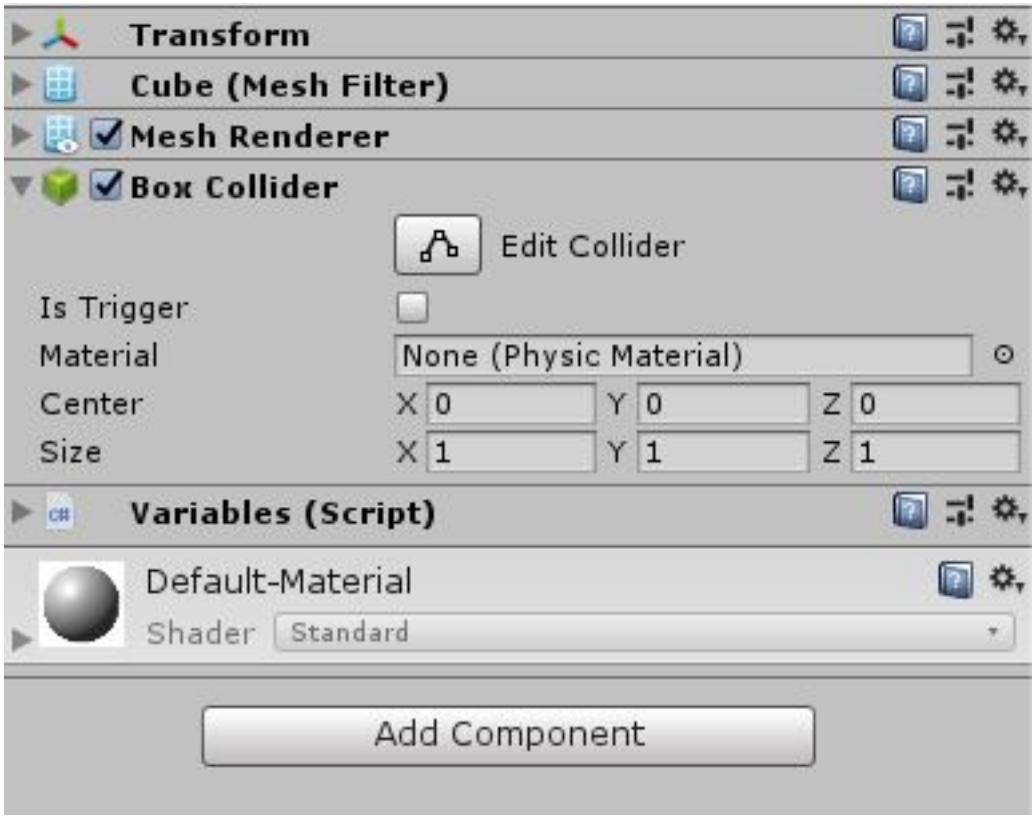
C# and Unity

Variables and the Inspector

When creating a script, you are essentially creating your own new type of component that can be attached to Game Objects just like any other component.



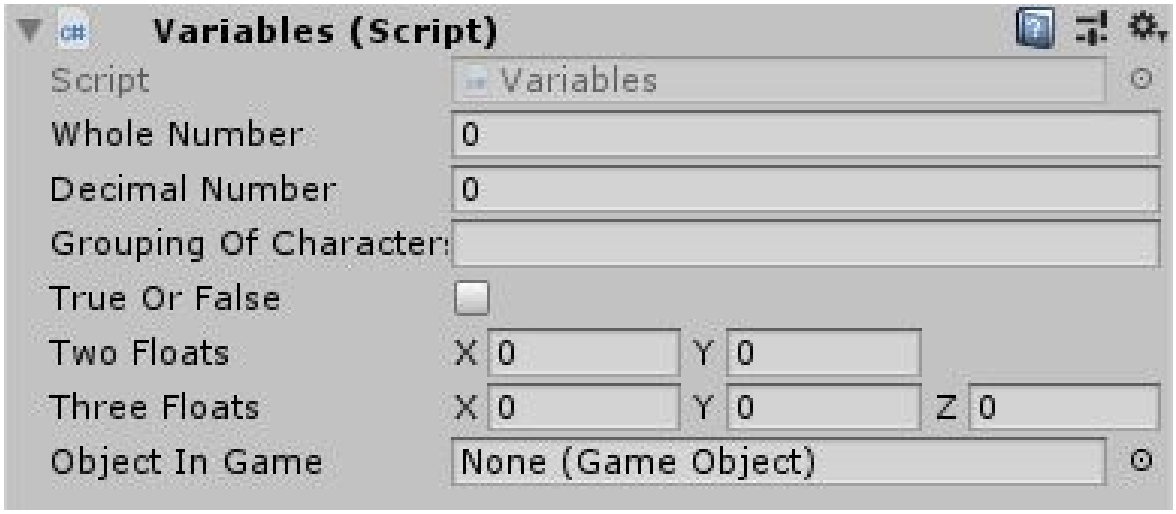
Just like other Components often have properties that are editable in the inspector, you can allow values in your script to be edited from the Inspector too.



Variable names are written using camel casing.
Camel Case is where the first word is lowercase and each new word is not separated by a space and the first letter of each new word is
capitalized.

```
public int wholeNumber;
public float decimalNumber;
public string groupingOfCharacters;
public bool trueOrFalse;
public Vector2 twoFloats;
public Vector3 threeFloats;
public GameObject objectInGame;
```

This code creates editable fields in the Inspector that are labelled with the variable name. In C#, you must declare a variable as public to see it in the Inspector.



Unity creates the Inspector label by introducing a space wherever a capital letter occurs in the variable name. However, this is purely for display purposes and you should always use the variable name within your code.

Unity will actually let you change the value of a script's variables while the game is running. This is very useful for seeing the effects of changes directly without having to stop and restart. When gameplay ends, the values of the variables will be reset to whatever they were before you pressed Play. This ensures that you are free to tweak your object's settings without fear of doing any permanent damage.

Variables, Operators, Brackets and Modifiers

Core Variables with in c#

int Integer – positive or negative whole number **-53, 0, 10, 401568**
string Collection of characters **aBThZx643#@$ (_-':***
float Positive or negative decimal number **-1.254f, 0.0f, 7598.255f**
bool Boolean – true or false **true, false**

Unity Variables and their make up

GameObject An Object with in the Game Contains Transform and object name
Transform Objects position, rotation and scale in the worldspace made of 3 Vector3's
Vector2 2 floats normally used as X and Y Made up of 2 floats - **vector2(0.0f,0.0f)**
Vector3 3 floats normally used as X, Y and Z Made up of 3 floats - **vector3(0.0f,0.0f, 0.0f)**

Commenting, Headers, Regions and keeping code neat

// commenting out a single line

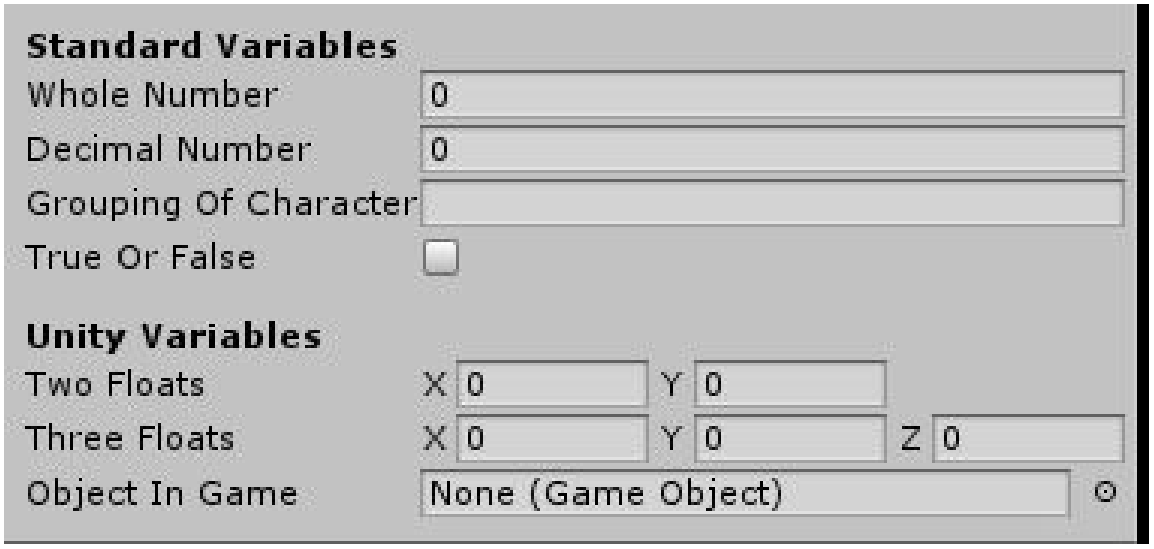
```
//Single line comment
```

/* */ commenting paragraph or multi line comment

```
/*
 Multi
 line
 comment
 */
```

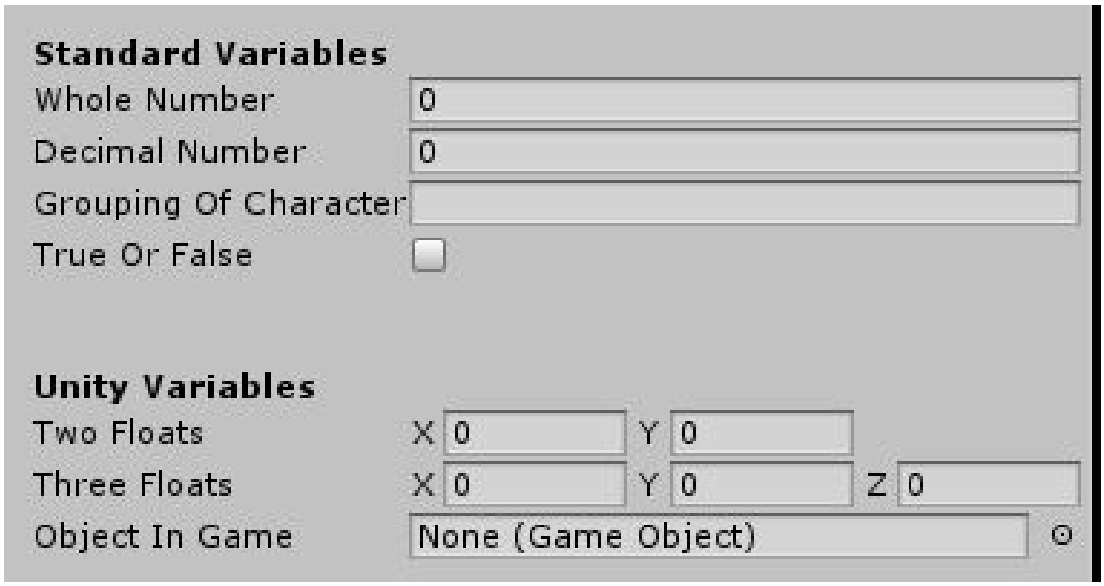
[Header("Header Name")] Allows public variables to have nice labels in the inspector

```
[Header("Standard Variables")]
public int wholeNumber;
public float decimalNumber;
public string groupingOfCharacters;
public bool trueOrFalse;
[Header("Unity Variables")]
public Vector2 twoFloats;
public Vector3 threeFloats;
public GameObject objectInGame;
```

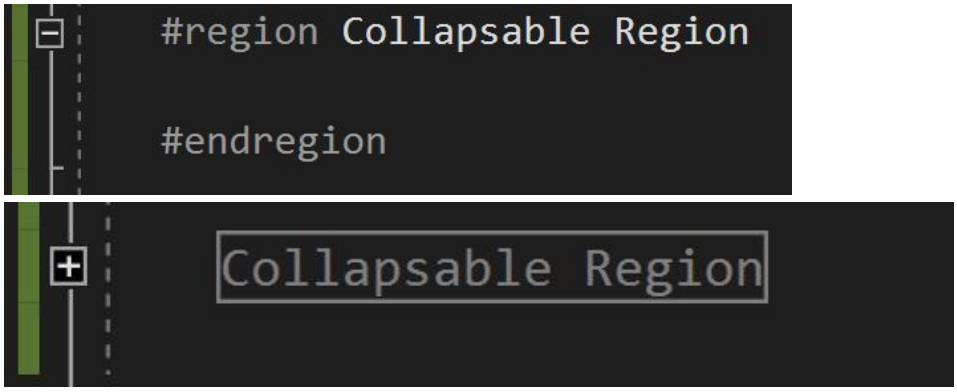


[Space(Integer)] Clears an empty space within the inspector

```
[Header("Standard Variables")]
public int wholeNumber;
public float decimalNumber;
public string groupingOfCharacters;
public bool trueOrFalse;
[Space(20)]
[Header("Unity Variables")]
public Vector2 twoFloats;
public Vector3 threeFloats;
public GameObject objectInGame;
```



#region Region Name Allows you to make collapsible areas to organise code
#endregion



Modifying Operators

+ Plus, Add, Adds two or more numbers 1+2 = 3
- Take, Minus, Subtract Takes two or more numbers 1-2 = -1
*****Multiply, Times Multiplies two numbers 3*3 = 9
/ Divide Divides two numbers 12/3 = 4
= Equals Sets the answer to a math question
++ Plus Plus Increase by 1
--Take Take Decrease by 1
+=Plus Equals health += healthPotion (50 += 10...now has the value of 60)
-=Take Equals health -= damage (60 -= 20...now has the value of 40)

Conditional Operators

== Is equal to or Is asking a question: **is value A the same as Value B**
||Or: **if one situation or the other is true**
&& And: i**f both situations are true**
!Not: **if the situation is false**
!= Not Equal to or Is Not: i**f Value A isn't the same as Value B**
<Less Than: **if Value A is Smaller than Value B**
> Greater Than: **if Value A is Bigger than Value B**
<= Less Than or Equal to: **if Value A is Smaller or the same as Value B**
>= Greater Than or Equal to: **if Value A is Bigger or the same as Value B**

Brackets and Braces

() Parentheses, are typically used for method or delegate invocation or in cast expressions. You also use parentheses to specify the order in which to evaluate operations in an expression.
[]Square brackets, are typically used for array, indexer, or pointer element access.
{} Curly Braces, hold and group code.
<> Calls Generic Method, Script or Component

Access Modifiers

publicThere are no restrictions on accessing public members.
privatePrivate members are accessible only within the body of the class or the struct in which they are declared.

new	Explicitly hides a member inherited from a base class.
static	Declares a member that belongs to the type itself instead of to a specific object.

Event Functions

Initialization Events

It is often useful to be able to call initialization code in advance of any updates that occur during gameplay.

Awake - runs even if the script is disables on the first frame that the game object is active in the scene

Start - runs on the first frame that script is enabled and the game object is active in the scene

Update Events

A game is rather like an animation where the animation frames are generated on the fly. A key concept in games programming is that of making changes to position, state and behaviour of objects in the game just before each frame is rendered.

Update - runs roughly every 0.0160 - 0.0555, would use this for inputs

LateUpdate - runs on the end half of Update, use to Clamp outcomes

FixedUpdate - runs on a fixed time 0.02, use for physics its constant

GUI Events

Unity has a system for rendering GUI controls over the main action in the scene and responding to clicks on these controls.

OnGUI - runs alongside Update, Renders GUI elements and events

Physics Events

The physics engine will report collisions against an object by calling event functions on that object’s script.

OnCollisionEnter - runs on moment of impact

OnCollisionStay - runs while still impacting

OnCollisionExit - runs when impact stops

OnTriggerEnter - runs when object enters zone

OnTriggerStay - runs when object stays in zone

OnTriggerExit - runs when object leaves zone

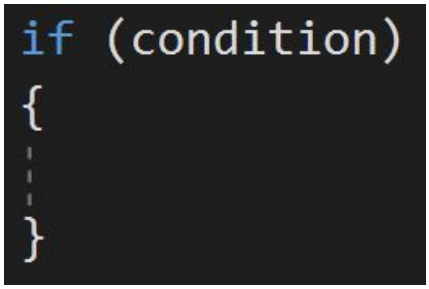
If Statements

An if statement identifies which statement to run based on whether the condition of the statement being asked it met.
Meaning the of the value the condition being asked has to be true, this is a Boolean expression.

An if statement in C# can take a few forms, as seen below.

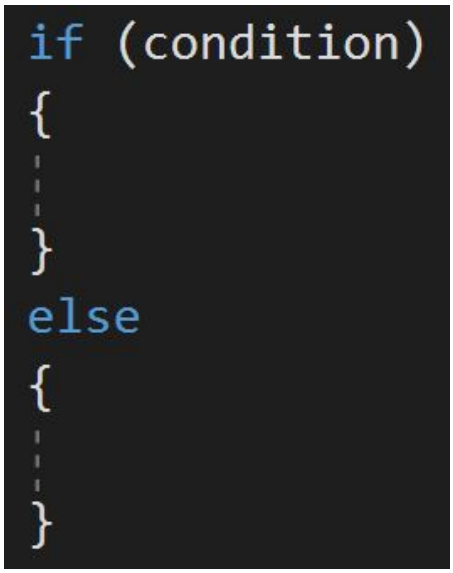
if condition is true run code written in if brackets.

nothing else happens if the if statement is not met.



if condition is true run code written in if brackets.

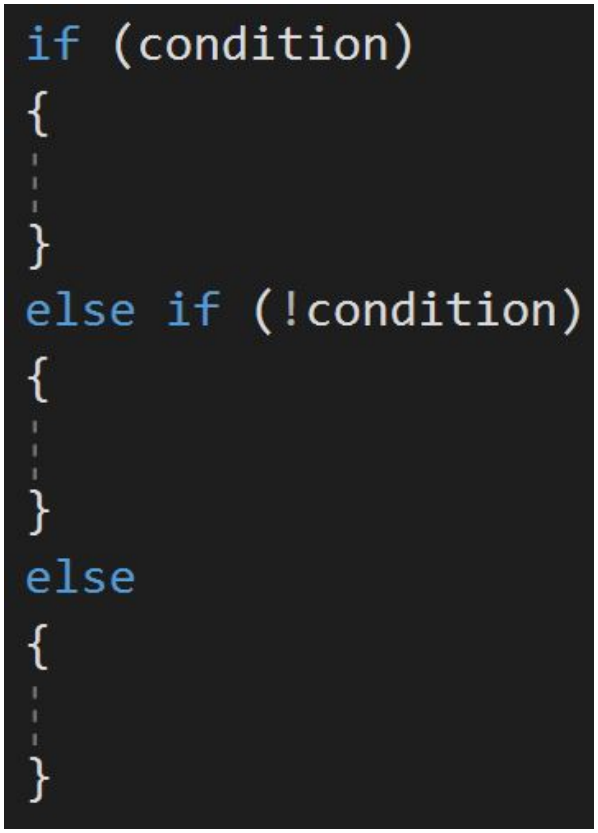
else for any other condition run code in else brackets.



if condition is true run code written in if brackets.

else if second condition is met run code in if else brackets.

else for any other condition run code in else brackets.



you can have as many else if statements as you like as long as all conditions are different.

Read more here.

Controlling GameObjects using components

In the Unity Editor, you make changes to Component properties using the Inspector. So, for example, changes to the position values of the Transform Component will result in a change to the GameObject's position. Similarly, you can change the color of a Renderer's material or the mass of a Rigidbody with a corresponding effect on the appearance or behavior of the GameObject. For the most part, scripting is also about modifying Component properties to manipulate GameObjects. The difference, though, is that a script can vary a property's value gradually over time or in response to input from the user. By changing, creating and destroying objects at the right time, any kind of gameplay can be implemented.

Accessing components

The simplest and most common case is where a script needs access to other Components attached to the same GameObject. As mentioned in the Introduction section, a Component is actually an instance of a class so the first step is to get a reference to the Component instance you want to work with. This is done with the GetComponent function. Typically, you want to assign the Component object to a variable, which is done in C# using the following syntax:

```
public Rigidbody rb;

void Start ()
{
    rb = GetComponent<Rigidbody>();
}
```

Once you have a reference to a Component instance, you can set the values of its properties much as you would in the Inspector. If you attempt to retrieve a Component that hasn't actually been added to the GameObject then GetComponent will return null; you will get a null reference error at runtime if you try to change any values on a null object.

Accessing other objects

Sometimes you need to access a Component on other GameObjects.
The most straightforward way to find a related GameObject is to add a public GameObject variable to the script:
You can create a reference and drag the GameObject into the slot in the inspector.

Referencing a GameObject

```
public GameObject otherObject;
```

Finding the GameObject in the scene by name

```
public GameObject otherObject;

void Start ()
{
    otherObject = GameObject.Find("Other Objects Name");
}
```

Finding the GameObject in the scene by tag

```
public GameObject otherObject;

void Start ()
{
    otherObject = GameObject.FindGameObjectWithTag("Other Objects Tag");
}
```

Finding child GameObjects

```
public Transform[] waypoints;

void Start()
{
    waypoints = new Transform[transform.childCount];
    int i = 0;

    foreach (Transform t in transform)
    {
        waypoints[i++] = t;
    }
}
```

Sometimes, a game Scene makes use of a number of GameObjects of the same type, such as enemies, waypoints and obstacles. These may need to be tracked by a particular script that supervises or reacts to them (for example, all waypoints might need to be available to a pathfinding script). Using variables to link these GameObjects is a possibility but it makes the design process tedious if each new waypoint has to be dragged to a variable on a script. Likewise, if a waypoint is deleted, then it is a nuisance to have to remove the variable reference to the missing GameObject. In cases like this, it is often better to manage a set of GameObjects by making them all children of one parent GameObject. The child GameObjects can be retrieved using the parent's Transform component (because all GameObjects implicitly have a Transform):