



C# Fundamentals

Today's Agenda



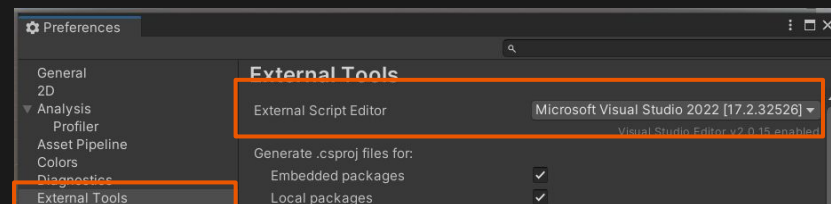
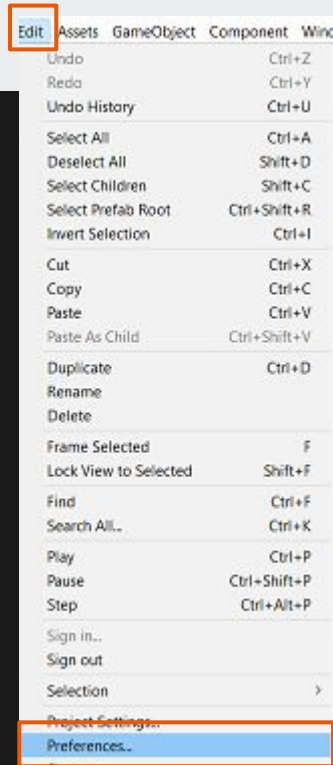
We're going to be going over Chapter 7: Scripting 1 and Chapter 8: Scripting 2

- We're going to learn how to declare and assign variables
- How to use boolean statements to determine actions
- Use loops and arrays to go through data
- Create your own functions

Unity Visual Studio Connected

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

Go to Edit in the File Menu drop down and check your Preferences, in there check External Tools and click on the External Script Editor drop down, if Visual Studio isn't selected check it and now all Scripts will open through Visual Studio.



Create Script

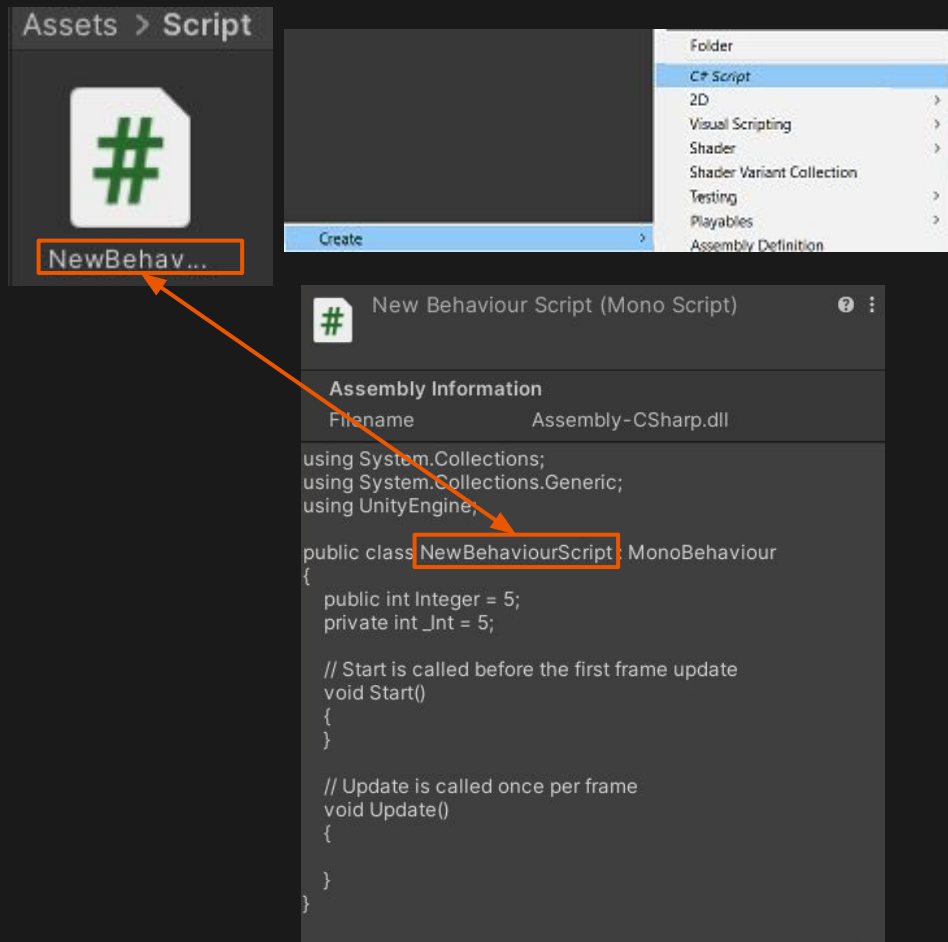
Now that we are sure that the Script will open up correctly we can create a Script Game Asset.

Go to the Project View and create the **C# Script**.

When creating the Script it's important to **name** it correctly the first time otherwise you might have issues as the name of the script has to match the name of the class inside the script.

If you need to change the name make sure you update both the Game Asset Name and the Class name in the file afterwards.

When selecting the Game Asset the Inspector will show you the preview of the Script inside it.



Anatomy of a Basic Script

The Using section of the program connect different **Libraries** to the Script you are programming in.

Libraries are just big collections of **Methods** or **Functions** that you can import, such as the Start and Update Methods shown in the body of the class.

As you see two of them are Grayed out and one is White, the Grayed out Libraries mean that they aren't currently being used as opposed to the **UnityEngine** one which is.

Now the **public class NewBehaviorScript** section defines our class. A class is a definition of component and anything in its **body**, anything between the { }, is part of its properties.

```
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
14      // Update is called once per frame
15      void Update()
16      {
17
18  }
```

Anatomy of a Basic Script

The `: MonoBehaviour` is an extension of the class meaning there is a Class called `MonoBehaviour` and it has its own properties that our new Script can use as if it was written inside its body. One of the main one being the ability to turn this Script into a Component.

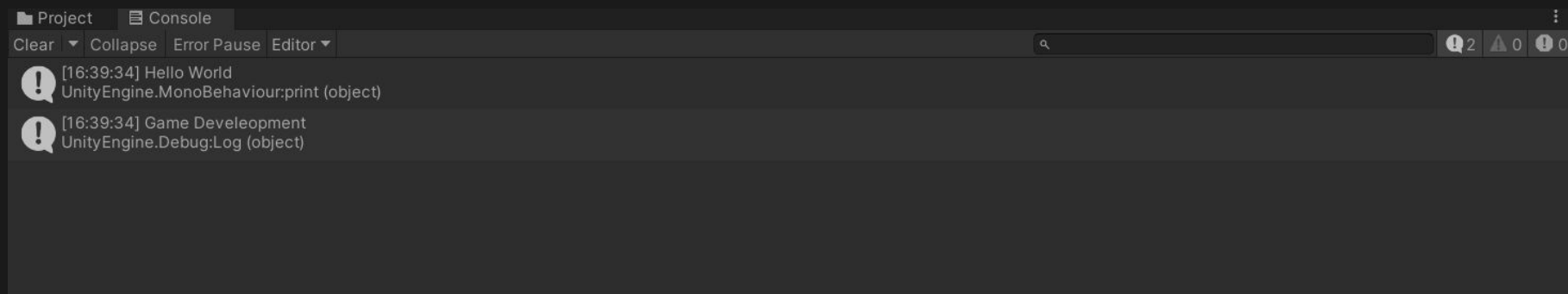
Now within the body of our class you will always start with two Methods `Start` and `Update`.

`Start` executes only once when the game goes into Play mode.

`Update` executes continuously every frame, every time Unity redraws everything on the screen. Frame is defined by your computer's performance, later we'll go over how to unify it.

```
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  }
```

Printing Out into Console



“Hello World”

You can print things to the Console View using either the `print()` Method or the `Debug.Log()` Method.

Notice that each of the line ends with a `;`

That indicates that the statement is completed, some statements may be very long and take up few lines so we need this end of line statement so that the computer will be able to separate all of the actions.

```
// Start is called before the first frame update
Unity Message | 0 references
void Start()
{
    print("Hello World");
    Debug.Log("Game Deveelopment");
}
```

Challenge 1: Hello World

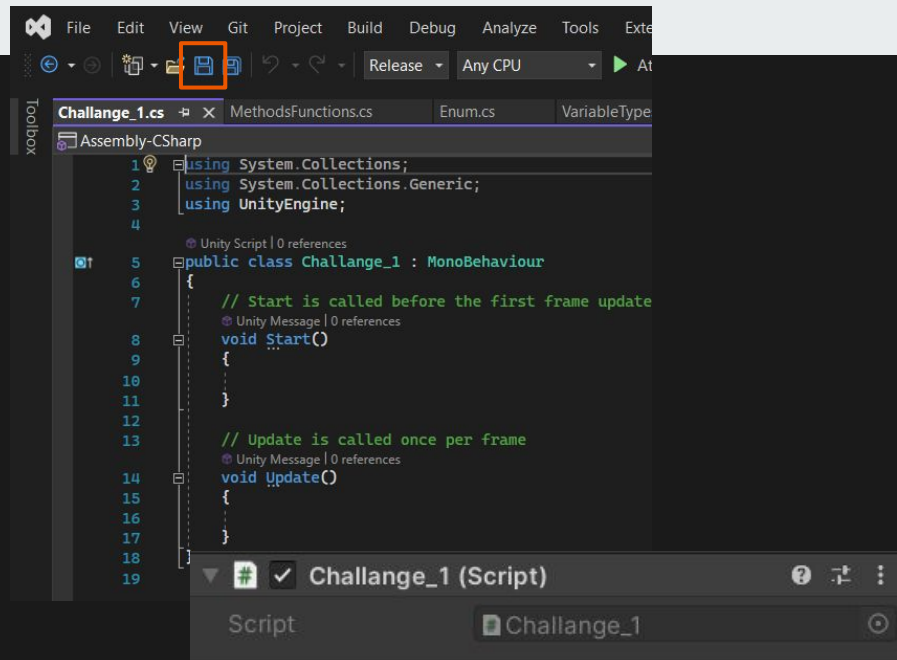
In the Start method's body use the `print()` function to print out the phrase "Hello World".

In the Update method's body use the `Debug.Log()` function to print out the phrase "Game Dev".


Make sure that once you've done the changes to the Script in Visual Studio you click the **Save** button otherwise those changes will not be reflected in the Script in Unity.

Finally you will have to **attach** the Script as a Component to a Game Object to see its effect take place. You can either do that by adding it from the Component List or just Drag and Dropping the Script to the component list.

Once you've done all of that you can click **Play** and see the results in the Console View.



Variables Types



There are many variables that you will be able to use as you create Scripts but these are the most common ones.

Int only allows you to use integers, they're great for keep count of items.

Float and **Double** are number that can use decimal places great for defining position, rotation, scale and so on. We'll mostly be using **Floats**.

Booleans come in only two choices, true and false and all us to keep track of actions.

Char are characters single letters that are used to make up **Strings**.

Strings are a collection of chars that can be used to display text to the user.

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

You cannot directly change one variable type to another so doing

Integer = String will result in an error, however there are internal Methods that allow you to convert Data Types.

And some data type can interact with each other as we will see.

Variable Scope



Scope of the variable is something very important to understand. When you place a variable in the body of a class it become a **Global** variable that any other function could access and modify.

Meanwhile if you create a variable inside a Method such as the start Method that variable is **Local** only to Start, other Methods such as the Update Method won't have access to it.

```
int variableOne = 4;

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

Declaration and Assignment



Declaration is the creation of a variable, while the **Assignment** is giving it a value.

In this case variableOne is just declared while variableTwo is declared and has the value of 5 assigned to it.

C# is a good language that will auto assign default variables to anything that is declared. In this case variableOne has a hidden = 0 at the end of it. However if you try to use an unassigned variable you will run into a error.

If you ever encounter something that's underlined in red that means the code will not execute, in most cases Visual Studio will tell you what the problem is if you hover your mouse over the issue.

```
int variableOne;
int variableTwo = 5;

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

print(variableOne);
```

Each declaration of variables requires two things, the **Data Type** in this case int and the name for the variable. You can't have two variables with the same name.

Once a variable has been assigned using the = will **overwrite** the value and give it that new value.

Public Vs Private Variables

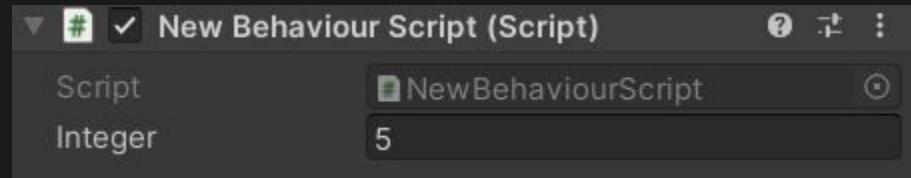


Similarly to Scope Private and Public variable tells the variable if they can be used only inside this Class or if they can be modified by other classes directly.

We've encountered few public variables in previous scripts that allowed you to change the speed at which the character was moving. Those you can directly influence through the Inspector View

Private Variables on the other hand are not visible in the Inspector View and will only be changed by the internal workings of the Script.

```
public int Integer = 5;  
private int _Int = 5;
```



Arithmetic Operators



Your programs will have to do math, whether it's moving the player in a direction or adding or subtracting health or collectibles you will have to perform Arithmetic operations.

You are familiar with four of them, **addition**, **subtraction**, **divisions** and **multiplication**.

The only one that might be new is **Modulo** or **%**. Think of it as a fraction asking how much is left.

If you have $5/2$ the equivalent is $2 \frac{1}{2}$, Module care about the number left on top at the end, 1.

```
// 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 number fits into the first and what is the remainder.
    //Here the result is 0 as 5 fits into 5, one time without leaving 0 as remainder. Think of it as 5/5 which is 1 0/5.
    variableOne = variableTwo % 5;
    print(variableOne);

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

Assignment Operators



Sometimes you want to quickly change the value of the variables you are currently working with.

Rather than saying `variable = variable + 2` you can condense the action by just performing the action on the **assignment**.

Thus `variable = variable + 2` turns into `variable += 2`.

In addition you will want to move **incrementally** by one or up or down these are even further simplified by performing `variable++` or `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 +=, -=, *= and /=
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);
```

Equality Operators

Equality Operators check if one of the two sides are and return you a value that's either true or false.

Since `=` is used as an assignment operator we use `==` to ask if two side are equal to each other.

Similarly we use `!=` to check if the two side are not equal to each other.

You've used the greater, less, greater or equal, and less or equal values in math classes.

The main idea to keep in mind is that `3 < 3` is false as computer since is very direct and `s 3` is not smaller then 3 it's false.

```
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



Logic operators allow you to string together several logic statements. Using `&&` and `||` connects two or more logic statements.

`&&` or And asks if both of the statement are true, if both are then the result is truth if any of the statements in the And is false the whole statement is false.

`||` or Or asks if any of the statements presented is true, if any of the statements present is true then the whole statement is true.

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


If Statement

If statements allow you to create paths for the code to flow through based on state of your variables.

If the If Statement is true the code inside its body will execute, otherwise it will be skipped and the next line will execute.

If Statement can be extended by adding an Else clause which will execute if the statement is false.

If statement can also have multiple secondary using the 'Else if' statements that provide other routes. Whatever the first statement that is true will execute.

Such as when we have `variable <= 6` and `< 6` the first one will execute and the rest will be skipped.

```
int variable = 0;
//Asks if a statement true, if is execute code inside the braces otherwise moves on
if (variable == 0)
{
    print("Statment One: If");
}
```

```
//Asks if a statement true, if is execute code inside the brace otherwise moves on
//Else execute the code in else braces
if (variable == 6)
{
    print("Statment Two: If");
}
else
{
    print("Statment Two: Else");
}
```

```
//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);
```

Ternary Operator is a special case of if statement that allows you to assign value based on the result of a statement.

So first you have the statement.

The Question mark tells us that the statement is over then the first value is the case where it's true

The second value is the case where it's false.

Switch Statement

Switch statement is very similar to a If Statement with main difference being that you are comparing a variable to different cases it could become.

It starts off placing in the variable you are using the compare to different states.

Inside the body of the switch statement you will create cases that the variable could equal to.

Each case will have break statement in it to jump out of the sequence, otherwise the code will just move on to the next state.

Switch statement have default cases which work like else in if statements.

```
//Switch Statement allows you to test the variable against a bunch of different cases
int variable = 0;
switch (variable)
{
    case 1:
    {
        print("Case 1");
        //Break Tells us to leave the Switch Statment,
        //otherwise we will continue to look at all the other statements
        break;
    }
    case 2:
    {
        print("Case 2");
        break;
    }
    case 3:
    {
        print("Case 3");
        break;
    }
    default:
    {
        print("Case Deafult");
        break;
    }
}
```

Enum

Enum is a way to create a Data Type that correlate Naming to integer variables.

This is very useful for switching between states in game.

You first create Enum and state its name and within the body you define the states that the enum could be.

Once you create an Enum you can create a variable of that enum that can be used as any other type of variable.

```
//Enum allows you to create named constants that are easier  
//for the programmer to keep track of states in a more undesirable  
//way than using integer to define states
```

5 references

```
enum State  
{  
    Load,  
    Playing,  
    Die  
}
```

```
//Enum create a new variable type that you can now access  
State currentState = State.Load;
```

```
// Start is called before the first frame update
```

Unity Message | 0 references

```
void Start()  
{  
    switch (currentState)  
    {  
        case State.Load:  
        {  
            //Do Something  
            break;  
        }  
        case State.Playing:  
        {  
            //Do Something  
            break;  
        }  
        case State.Die:  
        {  
            //Do Something  
            break;  
        }  
    }  
}
```

While Loop

```
int counter = 0;
//While loop executes until the statement is true,
//It is possible to make a statement that will not be
//fulfilled and create an infinite loop that will never finish executing
while (counter < 100)
{
    counter++;
    print(counter);
}
```

While Loop allows you to perform same action as long as the statement in the while loop is true.

In this case we are performing the action until the counter hit 100, the code does not do anything other than the while loop is working.

It's very easy to create an infinite loop that will run forever if the statement is never turned false, Update Method works a infinite while loop.

For Loop



The for loop does a very similar job to the while loop but with a clear end.

It works by declaring and assigning a variable that you will go through. Usually we use *i* and *j*, this variable will be localized to the for loop and once it's done the variable will no longer be accessible, such that you can declare another for loop with variable *i* without any problem.


Once you declare the variable you will be working with you then set the condition under which the for loop will continue. In first instance as long as *i* is less than 4 we execute the loop.

Lastly you set the increment step, in the first case it's by 1. So we will have 4 prints out of 0, 1, 3, 6.

```
int variable = 0;
//For loop allows you to perform something x amount of times 4
//you create a variable i that's only in the scope of the loop
//Once you exit the loop i no longer exists and can be used in a new
//loop
for (int i = 0; i < 4; i++)
{
    variable += i;
    print(variable);
}

//You can use any start, end and increments for the for loop
for (int i = 8; i > 0; i -= 2)
{
    variable -= i;
    print(variable);
}
```

Array



```
//Array holds a preset number of variables declared by the [4]
//with the variables specified in the {0, 1, 2, 3}
int[] array = new int[4] { 0, 1, 2, 3 };

//The positions in an array start at 0 and increment by 1
print(array[0]);

//Array has a length component that can be assessed by the .
print(array.Length);
```

Array is a container that can hold data types. You declare it by declaring a regular Data Type followed by [].

You need to use the special new operator, this asks the computer to allocate the space necessary for the container followed by the type of data and number of spaces you want to allocate.

Arrays are static conditions that can not add or remove any spaces in the container you've created.

You can declare the values of the array using {} or you can leave them unassigned without using the {}.

You can access each value using nameOfArray[x] where x is called the index and the first index is denoted by starting at 0.

Accessing Array



Best way to access all of the element in the array is using a loop.


Using the length -1 allows you to go through all of the variables. Remember array's index starts at 0 so if the length is 4 you want to go through indexes 0,1,2, and 3.

You could also use a special variation of the for loop called foreach in which you declare a variable of the same data type that is stored in the array and point to the array.

The foreach loop will go through every variable that exists in it and perform an action.

```
//For Loop is great for accessing each variable in the array,  
//the end size is array.Length - 1 as we start at 0 and end at 3.  
for (int i = 0; i < array.Length - 1; i++)  
{  
    print(array[i]);  
}  
  
//Foreach loop allows you to go through every variabe in the array  
foreach(int variable in array)  
{  
    print(variable);  
}
```


List



List differs from Array in a way that it allows you to Dynamically add and remove items to the container.

List uses some different wording such as Length being changed to Count.

Using the function .Add adds a item to the end of the list.

Using the function Insert(0,1) indicates that you want to place a variable of value 1 at the index of 0, the list will automatically shift all of the other values +1 index.

You can also use the Remove function to find the first occurrence of the value 52 and remove it from the list.

Likely you can also choose to use RemoveAt function to remove a specific index.

Game Object use lists to keep track of all of the children it has.

```
//Lists work very similarly to array with the exception that they have
//the power to be expanded or shrunk in their size
List<int> list = new List<int>();

//Tells you the Length of the list
print("Size " + list.Count);

//Extends the size and adds a the value at the end of the list
list.Add(52);
print("Index 0 " + list[0] + " Size " + list.Count);

//Extends the size and adds the value at the provided index of the list
list.Insert(0, 1);
for (int i = 0; i < list.Count; i++) { print("Index " + i + " " + list[i]); }
print("Size " + list.Count);

//Removes the first index that has the same value as pass in value
list.Remove(52);
print("Index 0 " + list[0] + " " + list.Count);

//Removes the value at the provided index
list.RemoveAt(0);
print("Size " + list.Count);
```

Array Vs List



Using an array versus list depends on what it is you are trying to achieve.

Let's say you are putting together a inventory for your character, you want to have a limited slots, kind of like the hotbar in minecraft. Each space in the bar is a index in the array and instead of storing integers it can be storing game objects.

On the other hand you want to use a list when you have something like Quest System, where your quest log is empty at the very start of the game and as you interact with people new Quests are added or removed based on your completion of them.



Methods - Functions



If you were only to code inside of Start and Update it would be impossibly long and you'd be repeating same actions ever few line.

To avoid that you can create your own Methods that will execute the same way every time you call on them.

In this example our function does nothing but prints out the Hello World but imagine one that prints out an array or list.

```
// Start is called before the first frame update
```

```
Unity Message | 0 references
```

```
void Start()
```

```
{  
:  
}
```

```
PrintOutStatment();
```

```
1 reference
```

```
void PrintOutStatment()
```

```
{  
:  
}
```

```
print("Hello World");
```

Returned and Passed in Values

Each method is built out of three elements.

The Return value, in our print statement it's void meaning we give nothing back but in the SquareValue method we return an integer, we do that by using the return operator.

Each method has to have a name that we can use to call it.

Lastly each method take in parameters that are indicated in the (), this is what is required for you to pass in for the function to work. Think about it like the print function requiring a String.

```
// Start is called before the first frame update
```

```
Unity Message | 0 references
```

```
void Start()
{
    PrintOutStatment();
}
```

```
1 reference
```

```
void PrintOutStatment()
{
    print("Hello World");
}
```

```
// Start is called before the first frame update
```


```
Unity Message | 0 references
```

```
void Start()
{
    int variable = SquareValue(2);
    print(variable);
}
```

```
1 reference
```

```
int SquareValue(int value)
{
    return value * value;
}
```

Default Parameters




Depending on what you're doing you might want a Method to act a certain way most time and differently sometime.

Default Parameters allow you to give a standard way of action of method that you could overwrite if you choose to apssi n a parameter.

```
// Start is called before the first frame update
@ Unity Message | 0 references
void Start()
{
    DeafulParametersPrint();
    DeafulParametersPrint("New Text");
}
```

```
/*
 * Purpose: Prints out the Hello Wolrd Statment
 * Parameters: N/A
 * Return: N/A
 */
0 references
void DeafulParametersPrint(string text = "Hello Wolrd")
{
    print(text);
}
```

Overloading Methods



Unlike variables you can have two of the same function as long as they take in different parameters.

This is called overloading and it's very useful when you want to perform the same action with different data types.

Imagine you have the SquareValue function. You wouldn't want to make SquareValueInt and SquareValueFloat, you rather just have SquareValue and pass in whatever data you need.

```
// Start is called before the first frame update
// Unity Message | 0 references
void Start()
{
    PrintOutStatment();
    PrintOutStatment("Game Development");
}

1 reference
void PrintOutStatment()
{
    print("Hello World");
}

1 reference
void PrintOutStatment(string statment)
{
    print(statment);
}
```

Commenting



Just like keeping your Project and Hierarchy Views organized so it is same for the code.

Whether you're working in a group or by yourself over time you will forget what the code does and good documentation will help you or your partner catch up to speed.

This is my personal style of organizing but you can develop whatever way of documenting works for you.

```
//=====
// 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");
}
```

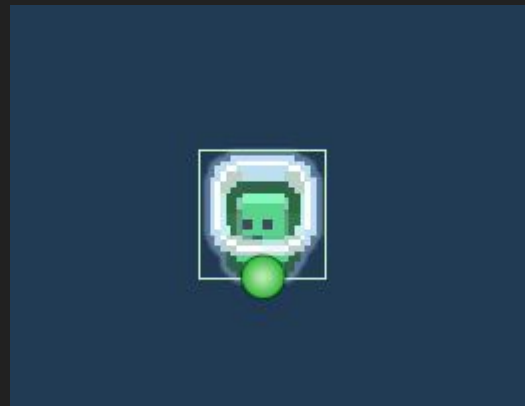
Player

Player

We will now create a player. The player is composed of five game objects:

1. Parent
2. Sprite
3. Main Camera
4. Ground Check
5. Particle Effect

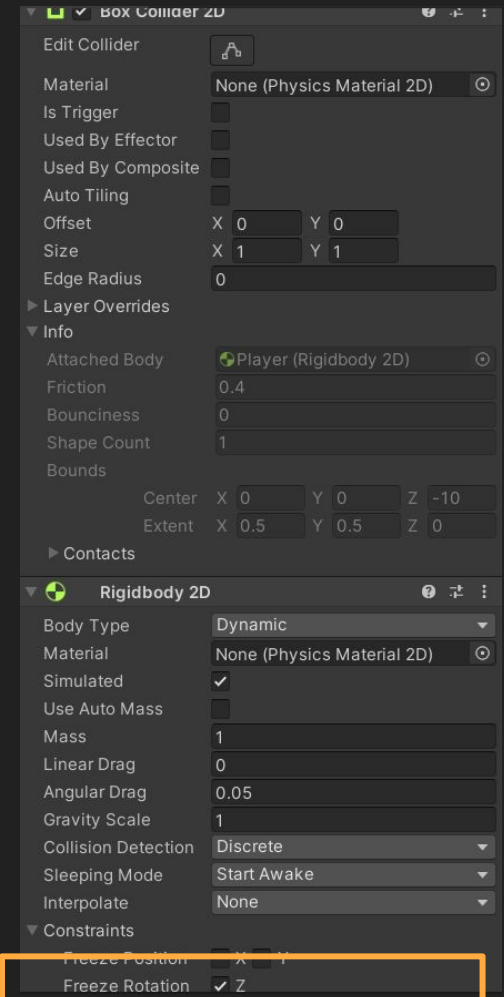
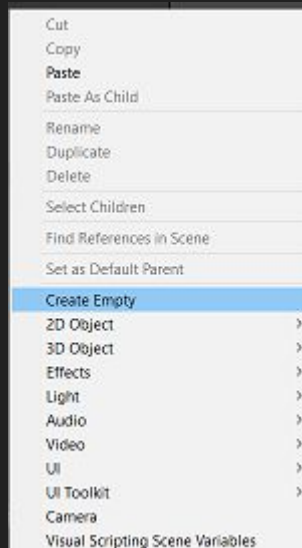
In addition to those, we will create an empty game object called RespawnPoint that the player will be sent back to if they die.



Player Object

Let's start by creating an empty game object and call it "Player." We're going to connect a Box Collider and Rigidbody.

Make sure the Rigidbody has Frozen Z rotation.

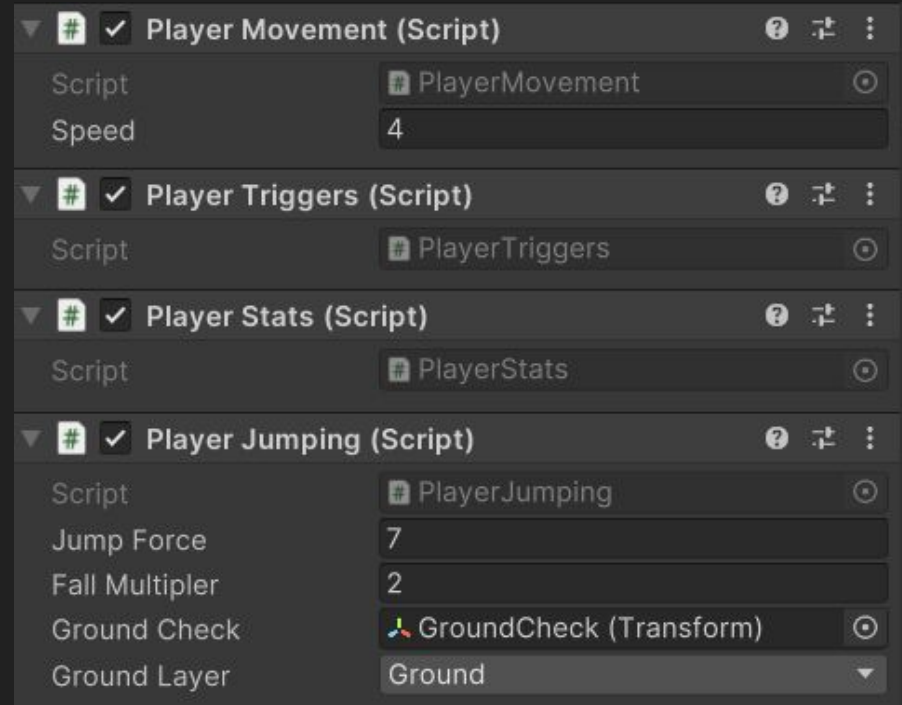


Scripts

We're going to add several scripts to the "Player" game object.

We're going to add "Player Movement," "Player Trigger," "Player Stats," and "Player Jumping."

We'll fill out these scripts as we continue making the level.

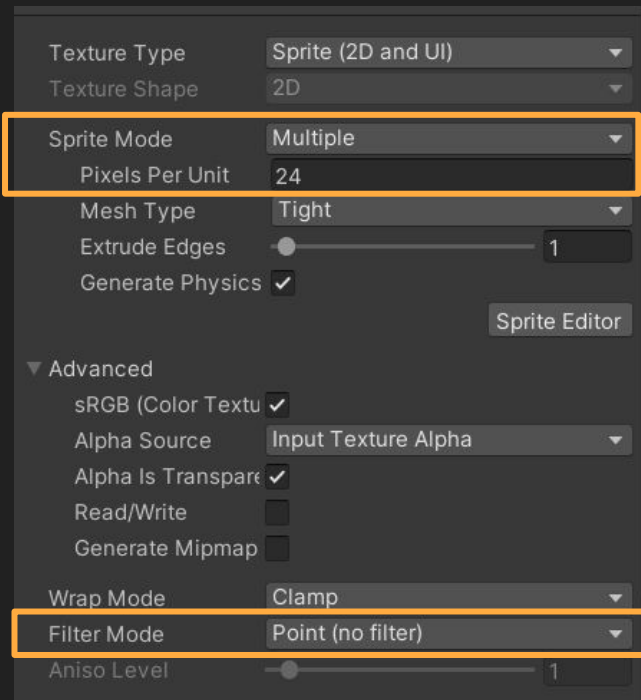


Tilemap

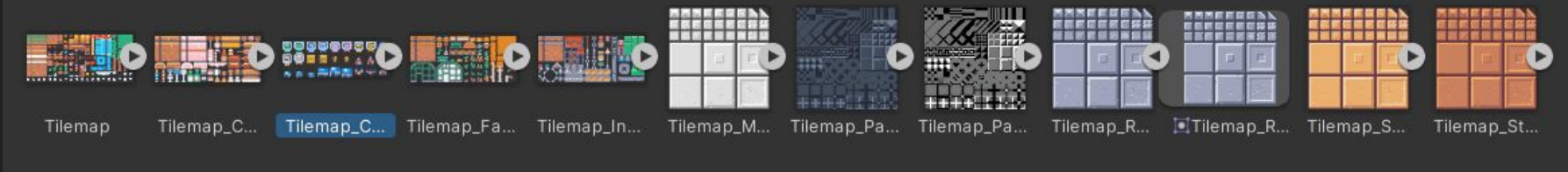
Before we can give the player a sprite, we have to break down the spritesheet.

Like before, go to the Tilemap folder, look at Tilemap_Character, and set Sprite Mode to Multiple, Pixels Per Unit to 24, and Filter Mode to Point.

After it's all set up, open up Sprite Editor.



Assets > Sprites > Tilemaps



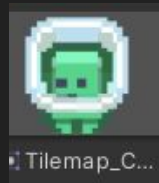
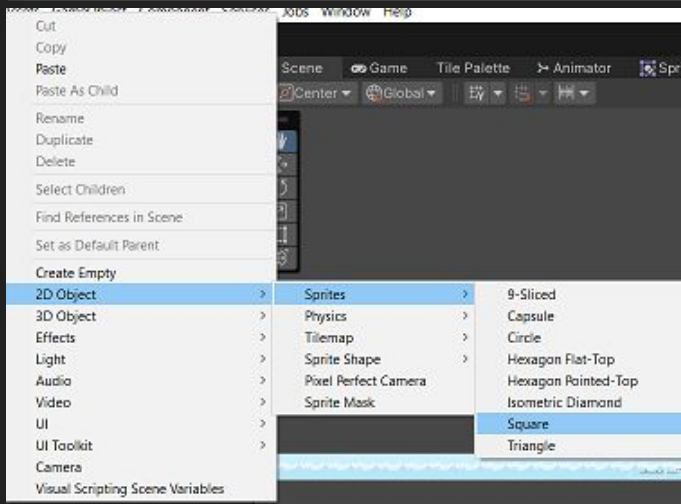
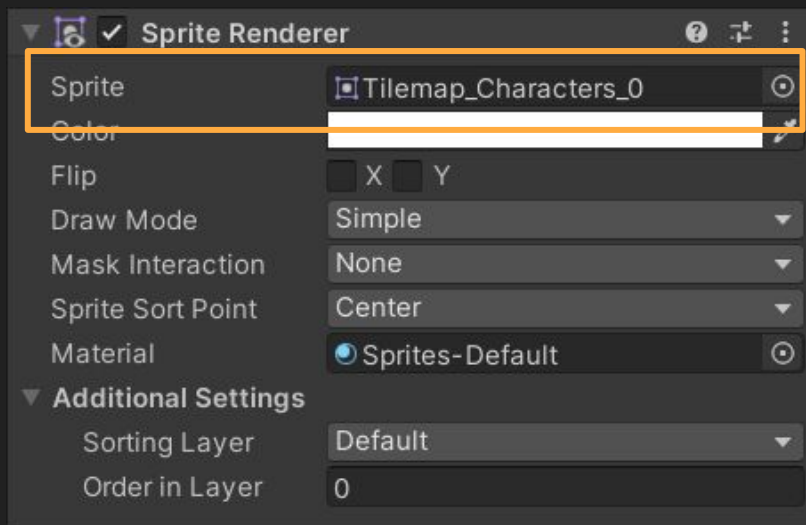
Sprite Editor

We're going to slice it, go to Grid By Cell Size, with the size being 24x24, slice it, and remember to apply the changes.



Sprite

Next, create a child that has a Sprite Renderer component. Set the Sprite in the Sprite Renderer to be the green guy.

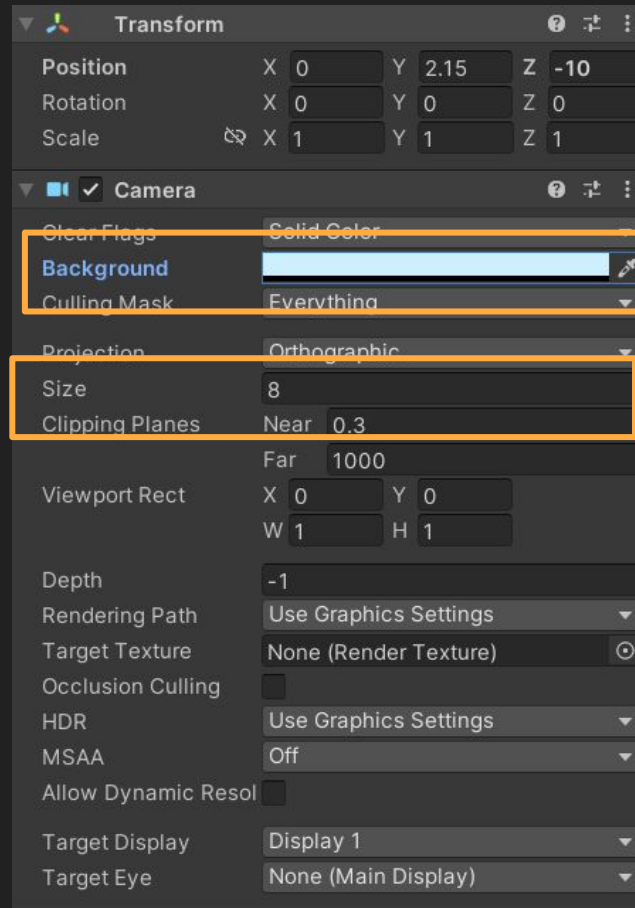
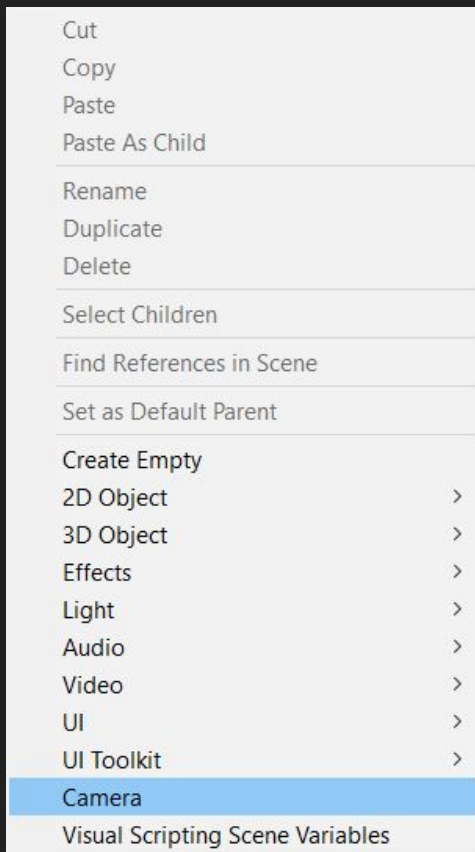


Camera

Add a camera to the Player by creating a camera object. Inside the camera, you want to set the Y position a little above so the player is a bit lower on the screen.

Ensure that the Size is large so the player has a good view of the level we're creating.

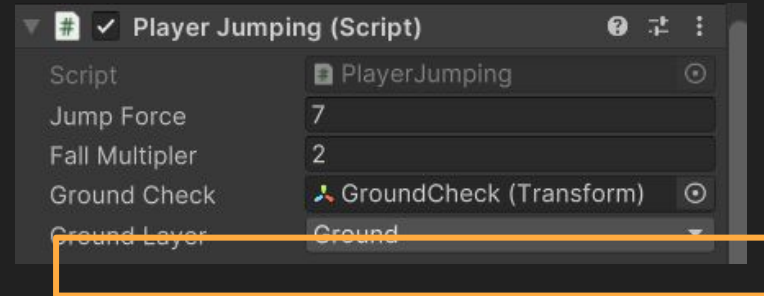
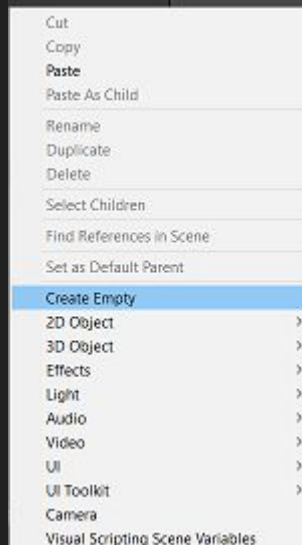
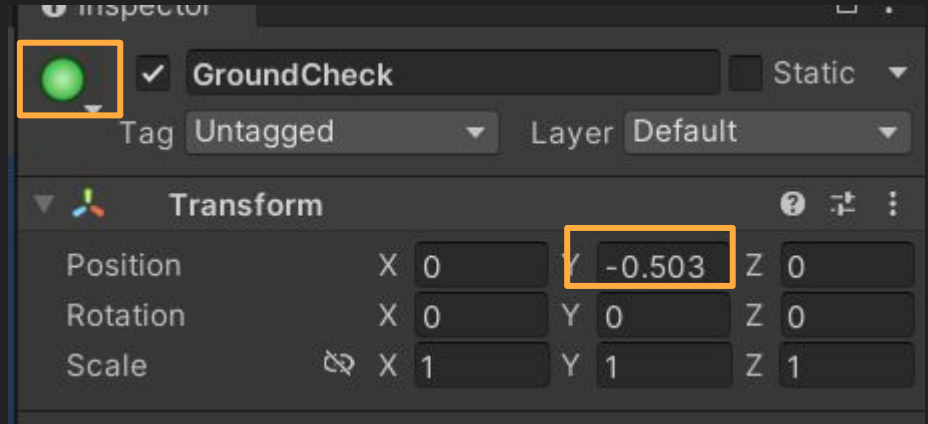
Also, make sure the background color is the same as the background image, so if the player jumps past the background top, it looks like the same color.



Ground Check

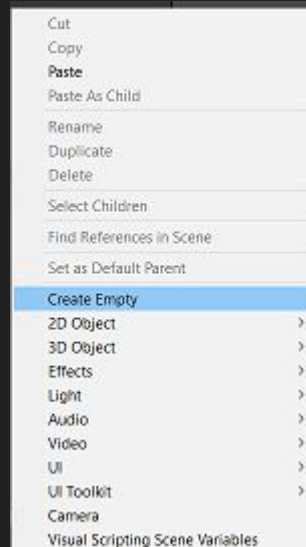
Create an object called "GroundCheck" and add an icon to it so you can see where it is.

Move it to the feet of the player, and this will act as a jumping collision box created through code. Make sure you connect this object to the script in the player and set the ground layer to "Ground."



Respawn Point

Create an empty Game Object, call it "RespawnPoint," and set the icon to be a red circle. This will allow the code to respawn the player at that position if they touch something bad.



Level Objects

Level Objects

There are several objects that we will create to make the levels more interesting:

1. Coins
2. Heart Pickups
3. Power-Up Items
4. Checkpoints
5. Bounce Pads
6. End Level Goals

In addition to that, you will be provided with two enemies to populate your game.

Assets > Prefabs



BouncePad



CheckPoint



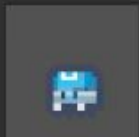
Coin



EndLevel



EnemyPat...



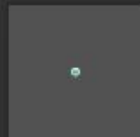
EnemyPat...



Fade



Heart



Player



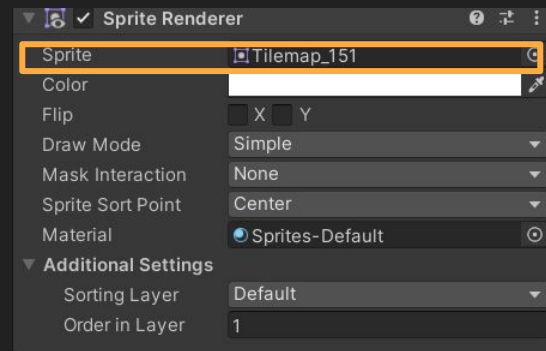
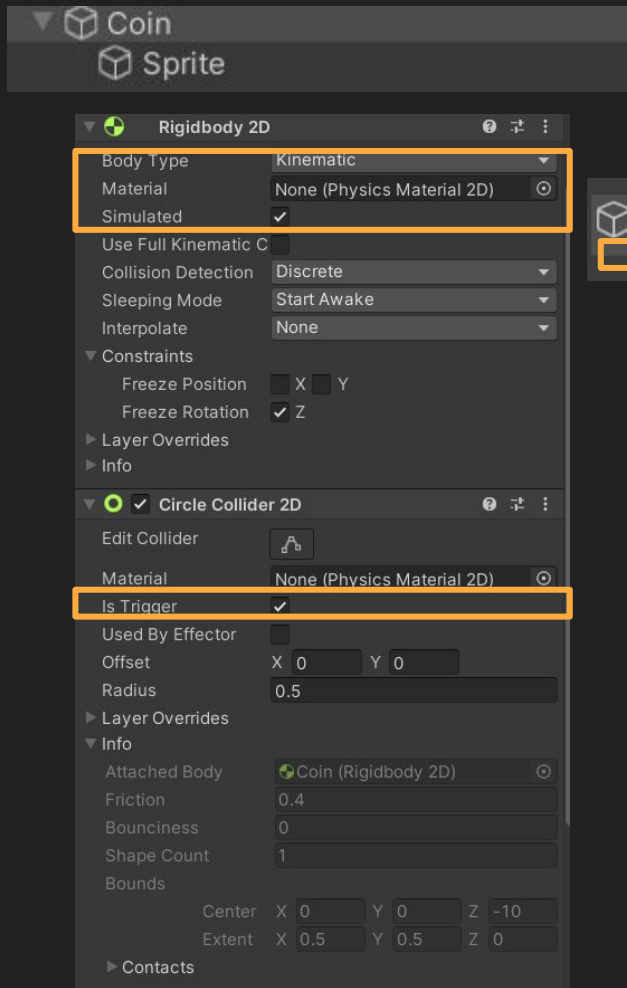
PowerUp

Coin

Create an empty game object, and then create a sprite object that's a child of that.

The parent should have a Circle Collider that's a trigger, a Rigidbody whose mode is set to Kinematic, and Z rotation is frozen. Set the tag to be "Coin" so that the code will react to it.

And set the Sprite in the Sprite Renderer to be that of the coin.

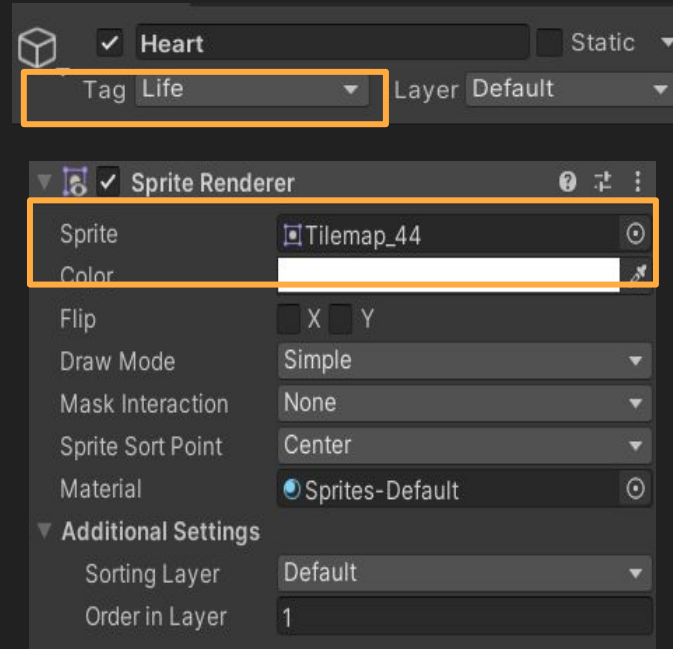


Heart

To create the heart, copy the Coin game object, and we'll change a few things.

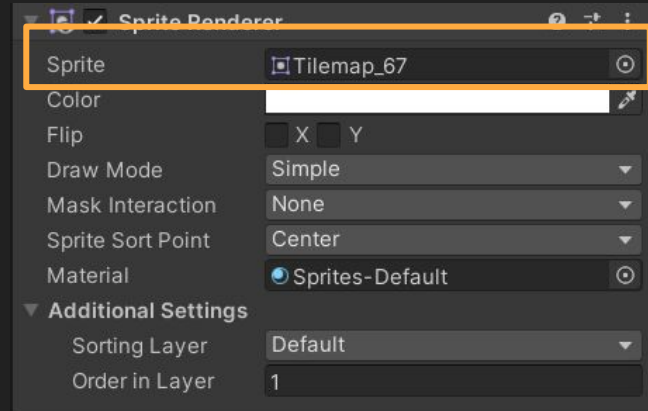
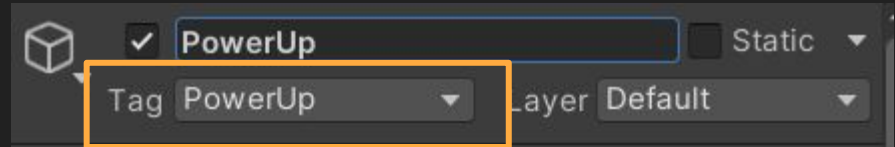
Change the name to "Heart," the tag to "Life," and the Sprite in the Sprite Renderer to the heart tile.

All the other components should carry over from the coin, and you don't have to set them up again.



Power Up

For the Power Up, we'll do the same thing. Copy the Heart, and change the Name to "Power Up," the tag to "Power up," and the sprite to the diamond sprite.

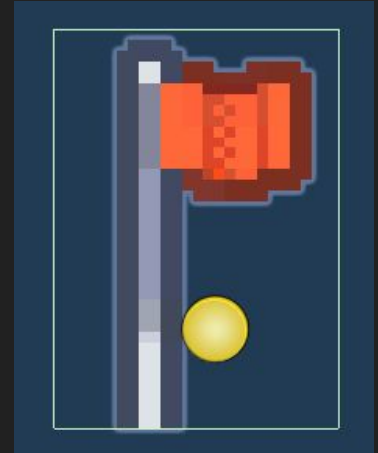
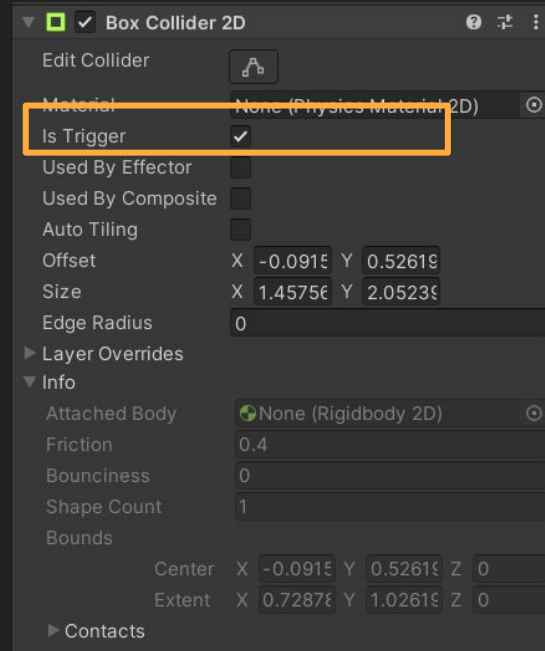
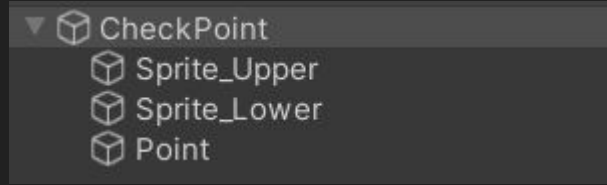
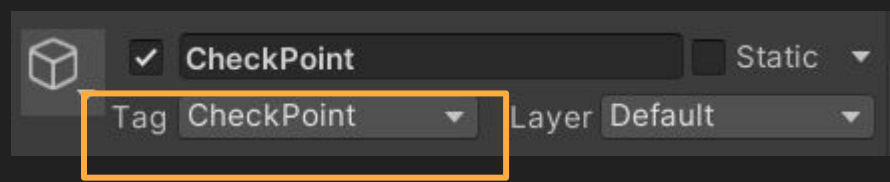


Checkpoint

The checkpoint will be made up of a few more objects than normal.

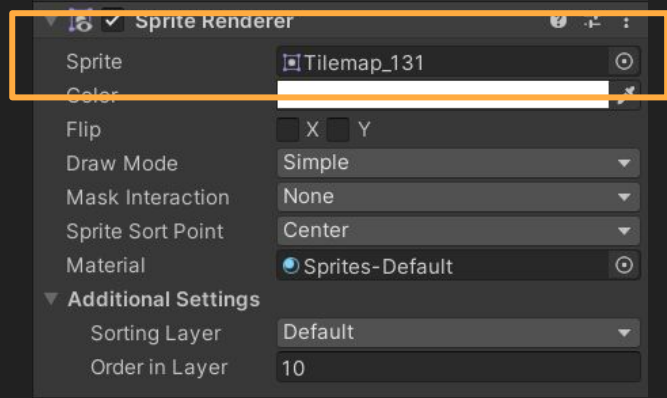
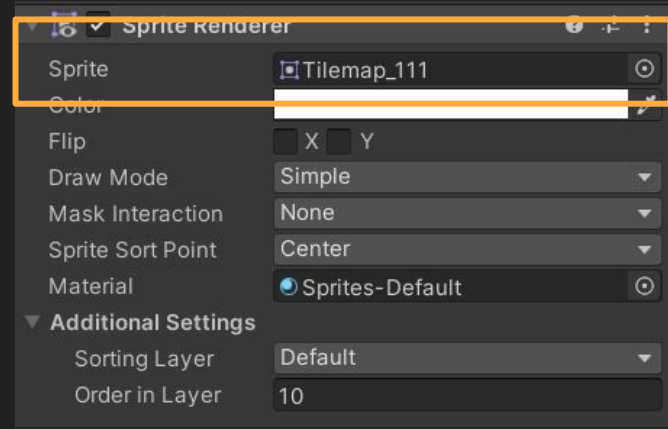
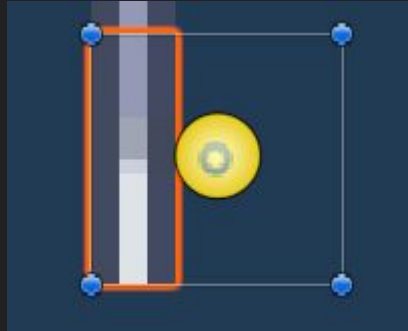
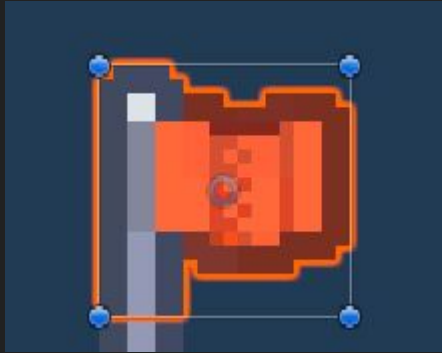
The parent will have a collider that will act as a trigger. Then it will have two sprites, an upper one with the flagpole and a lower one with the lower part of the flag.

Lastly, it will have a point that will be used by the respawn.



Sprites

You will have two children objects, both with Sprite Renderers, and each one showing half of the flagpole.

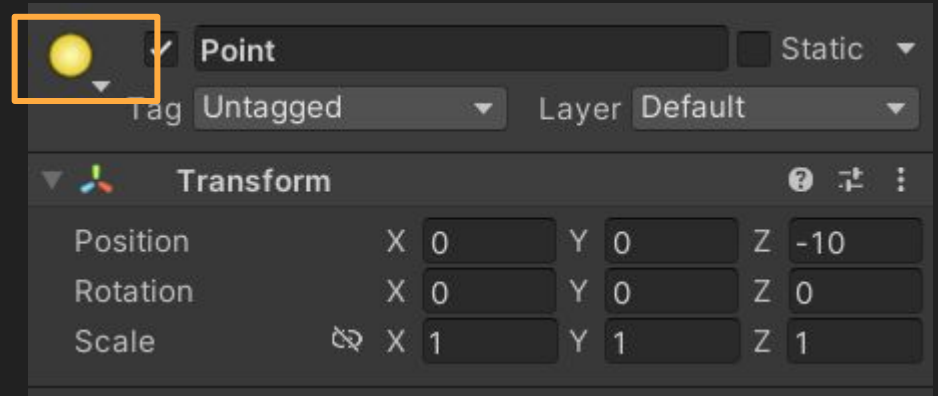


Point

Lastly, create a point, give it a yellow circle icon.

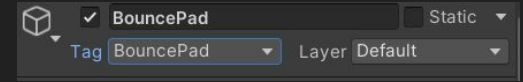
This will serve as a respawn point update; when the player triggers the checkpoint, this will be where the player respawns upon death.

Make sure to place the point near the bottom of the flagpole.



Bounce Pad

The bounce pad will be the simplest object. The parent will have a Box Collider that is a trigger and uses an effector. The effector we use is the Area Effector 2D to bounce the player 90 degrees in the sky. Its child will have a sprite of a spring.

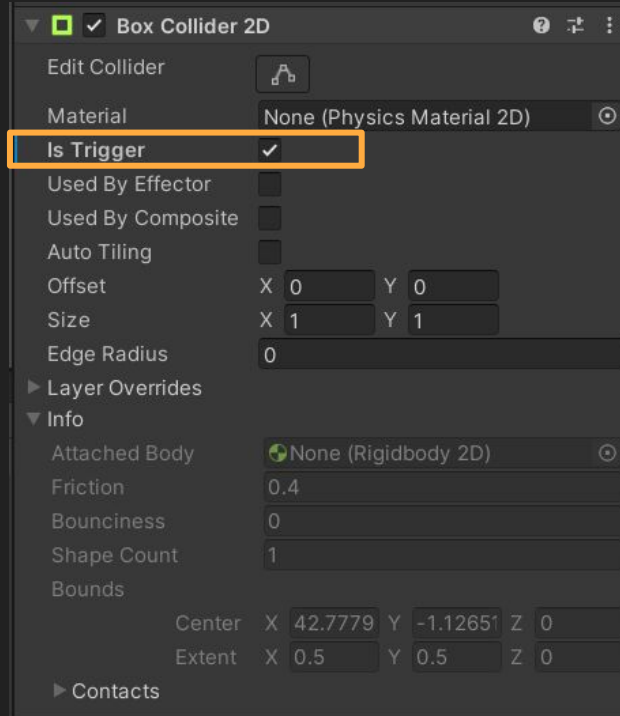
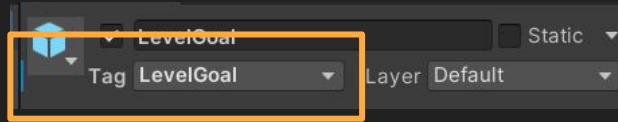
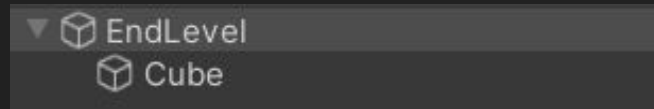


End Goal

For the last object, we're going to make it special.

We're going to place a 3D object in our 2D game.

The parent of the object should have a 2D box collider that's a trigger. And it has the LevelGoal tag attached to it.

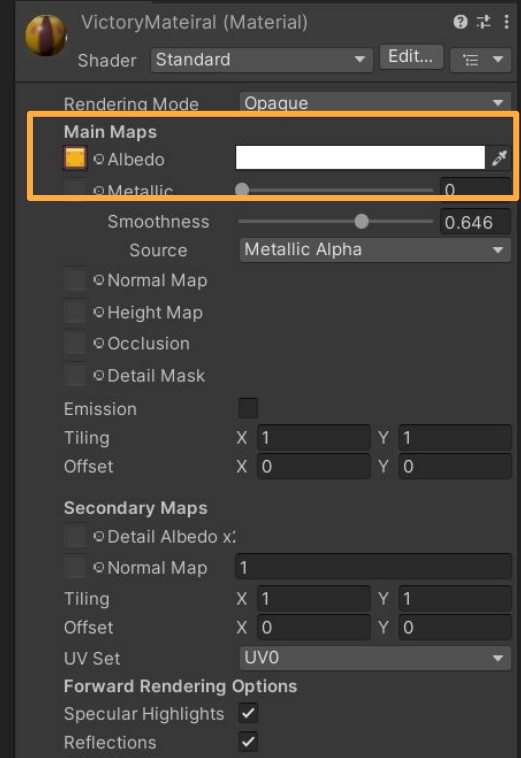
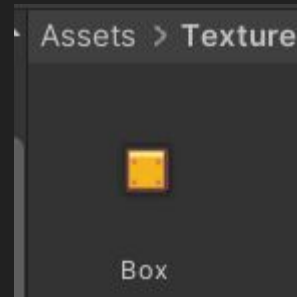


End Goal Texture & Material

In your Materials folder, you will find a material called "Victory Material."

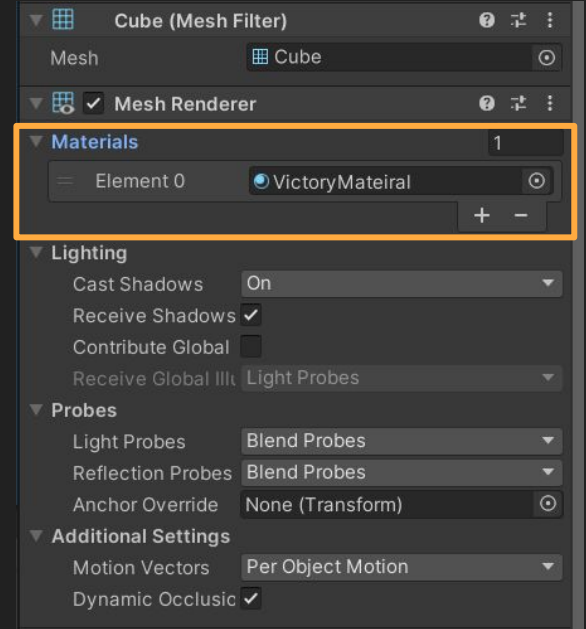
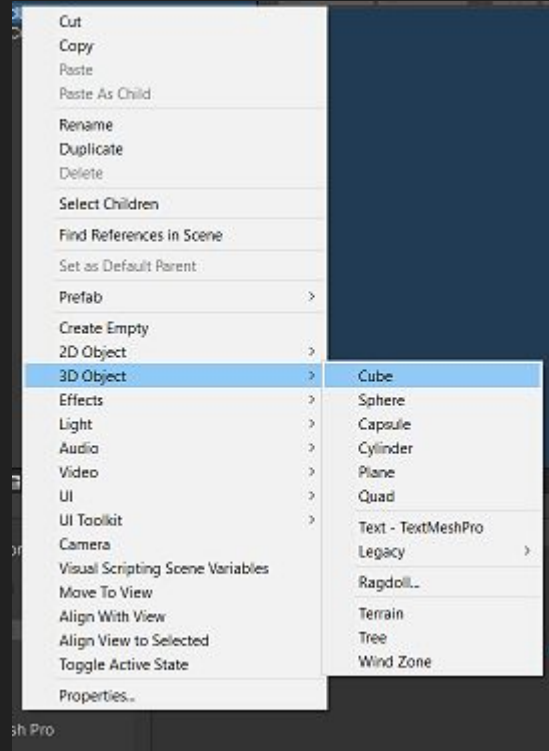
We're going to attach a Box Texture to it so that it looks like the box from our art.

Drag the texture into the albedo of the material.



Cube

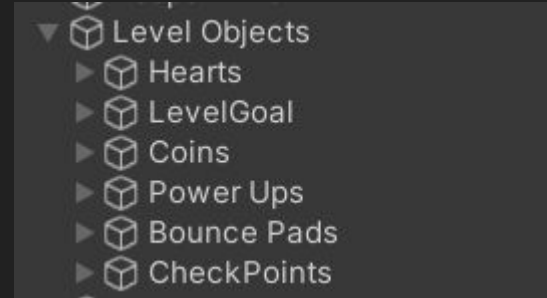
Create a cube as the object, and in the Mesh Renderer, attach the "Victory Material."



PreFabs

Once you've created all of your objects, make them into Prefabs by dragging them into the Prefab Folder from the hierarchy.

Also, make sure to create parent game objects for each of the items you've created. This will help you keep your level organized.



Assets > PreFabs



BouncePad



CheckPoint



Coin



EndLevel



EnemyPat...



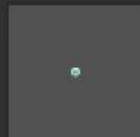
EnemyPat...



Fade



Heart



Player

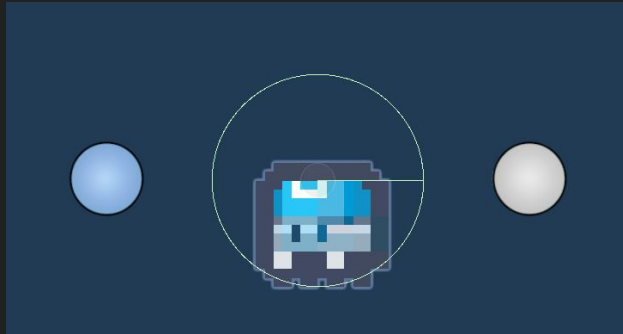
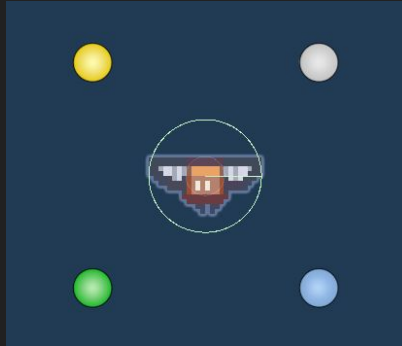


PowerUp

Enemies

In your Prefabs folder, you will find two enemies that are pre-made for you.

They are made up of two components: a Patrol Points component, which dictates their travel path, and the Enemy itself, which will kill the player if they are touched.

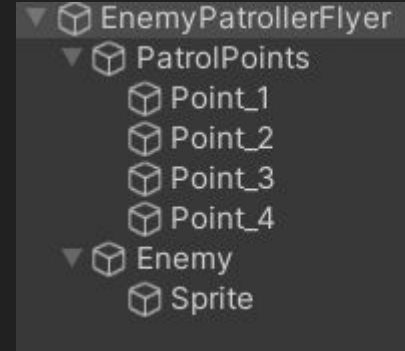


Enemies

The enemies have a circle collider that is triggered and with a script that will reference the patrol points.

You will notice that the points are separated from the enemy.

If they were under the enemy, the points would move when the enemy moves, meaning the enemy would never reach them, so we separate them to not affect their position during the enemy's movement.



Enemy Script

The script is made up of the points that the enemy will move between and the time it will wait at each location after getting there.

We also pre-set it to appear at the first point when the game starts.

```
Unity Script (2 asset references) | 0 references
public class EnemyPatroller : MonoBehaviour
{
    //===== Movement
    public List<Transform> _patrolPoints = new List<Transform>(); //Holds all the positions that the enemy will travel
    private const float Speed = 2f; //Speed of the enemy
    private int _currentPointIndex; //Which point are they at right now

    //===== Pause
    private float _waitTime = 0.5f; //Timer
    private const float StartWaitTime = 0.5f; //What timer resets to

    // Start is called before the first frame update
    Unity Message | 0 references
    private void Start()
    {
        //Sets the position and rotation to point 1 or 2 based on isStartingOnPointOne state
        transform.position = _patrolPoints[0].position;
        transform.rotation = _patrolPoints[0].rotation;
    }
}
```

Enemy Script

The enemy will move along the path to the current location.

When they reach it, they will set the next location as their destination. If the next current location they're heading to is the final one, they will look back to the first location and repeat the process.

```
//Updates the enemy to move between different points provided in the array
// Unity Message | 0 references
private void Update()
{
    //Moves towards the current objective
    transform.position = Vector2.MoveTowards(transform.position, _patrolPoints[_currentPointIndex].position,
        Speed * Time.deltaTime);

    //Checks if the enemy has reached the point it's moving towards
    if (transform.position != _patrolPoints[_currentPointIndex].position) return;

    //Checks if it has stayed on the point for long enough
    if (_waitTime <= 0)
    {
        //If its last point in array go to first point
        if (_currentPointIndex == _patrolPoints.Count - 1)
        {
            _currentPointIndex = 0;
        }
        //Go to next point in array
        else
        {
            _currentPointIndex++;
        }

        //Update the rotation of the sprite
        transform.rotation = _patrolPoints[_currentPointIndex].transform.rotation;
        //Reset the timer
        _waitTime = StartWaitTime;
    }
    //Else count down till it leave the point
    else
    {
        _waitTime -= Time.deltaTime;
    }
}
```