Dice Game

Here we are going to write our first program.

Structure

To begin we need to know where to actually write our code.
In the Editor duh!
Not exactly we can't just put it anywhere in this file. For this we need to use the "Start()" function.

Let's just talk a bit more about those functions.
The "Start" and "Update" functions are inside the class `TestScript` . How do we know this? The `{}` !
You can more or less think of `{}` or curly braces as a container. When we make our `class` we "open" a curly brace, put what we want in the class inside the brace, then "close" the brace.
So if you look at the code below you'll notice after we "declare" our `class` we have a `{` then at the bottom of the code a `}` . So everything between these two brackets is inside the `class TestScript` .

```
public class TestScript : MonoBehaviour {

        // Use this for initialization
        void Start() {

        }

        // Update is called once per frame
        void Update() {

        }
}
```

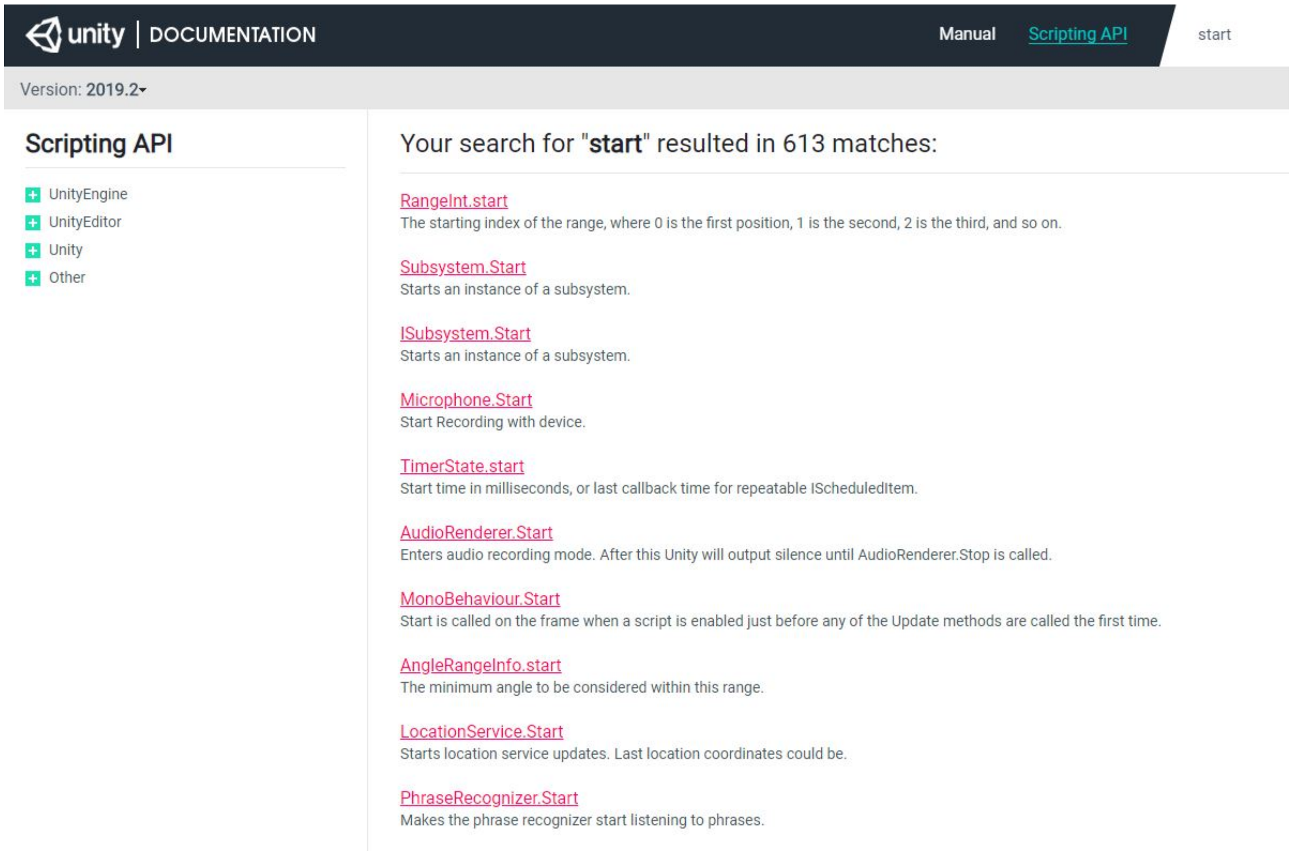
This is what is contained within the class `TestScript`. Using this same logic we can apply it to the `Start()` function. Look for the curly braces just after it.

```
void Start() {

}
```

Using The Scripting API

Now why do we want to use the `Start` function?
We're going to try out using the Unity Scripting API for this.
Say we were perusing through this code, and we stumbled across this function and didn't know what it does. Well, we'd go to the Scripting API and search "Start", the name of the function, up. When you do, you should be greeted by this :



Your search for "**start**" resulted in 613 matches:

RangeInt.start
The starting index of the range, where 0 is the first position, 1 is the second, 2 is the third, and so on.

Subsystem.Start
Starts an instance of a subsystem.

ISubsystem.Start
Starts an instance of a subsystem.

Microphone.Start
Start Recording with device.

TimerState.start
Start time in milliseconds, or last callback time for repeatable IScheduledItem.

AudioRenderer.Start
Enters audio recording mode. After this Unity will output silence until AudioRenderer.Stop is called.

MonoBehaviour.Start
Start is called on the frame when a script is enabled just before any of the Update methods are called the first time.

AngleRangeInfo.start
The minimum angle to be considered within this range.

LocationService.Start
Starts location service updates. Last location coordinates could be.

PhraseRecognizer.Start
Makes the phrase recognizer start listening to phrases.

This is a list of all the `Start` functions in Unity. Now we need to find the one we are actually using in our script.
In this case we should look at the class we are using. In the last lesson we mentioned how the `TestScript` class we made is "inheriting" from the `MonoBehaviour` class.
With this logic we can say that the `Start` function is part of the `MonoBehaviour`. When we look at our search result, you can see that the seventh one down is "Monobehaviour.Start". Click on that.

On this page, Unity tells us everything about `MonoBehaviour.Start` that we need to know.

> Start is called on the frame when a script is enabled just before any of the Update methods are called the first time. Like the Awake function, Start is called exactly once in the lifetime of the script. However, Awake is called when the script object is initialized, regardless of whether or not the script is enabled. Start may not be called on the same frame as Awake if the script is not enabled at initialization time.

This might sound a little confusing as it is written by and for more experienced developers. That is ok though, you'll begin to understand it better as we go along.
For now we will paraphrase.

> The Start function is run as soon as the script becomes active in your game. It's only ever called once.

When we press play in our game if our script is loaded onto an object (we'll get to that later) the `Start` function on that script (if there is one) is run.

Since we want our code to run by itself we will write it inside this function we can type it on the line after `void Start() {` but before the `}` .
Let's type this in `Debug.Log("Hello World");` . For now just write this exactly, and we'll customize it later.
The function should look like this :

```
void Start() {
        Debug.Log("Hello World");
}
```

Syntax

Before we continue we should talk about the spacing being used. You'll notice that the `Start` and `Update` functions are tabbed in and now the `Debug.Log` is tabbed in again. This is to ensure our code is "readable" so when you have to come back to it or someone else wants to look at the code it is very easy to see where everything is and what it does. The main idea with tabbing in our code is that we want to be able to see what is inside what. `Debug.Log` is inside our `Start` function. We know this because of the curly braces. So we make `Debug.Log` indented a little further than start.
Most editors (like the one Unity will have opened for you) automatically indent your code and help maintain a standard "Syntax". Syntax is our fancy word for the formatting of code.

There are tons of rules you can follow for writing syntactically nice code. Certain people prefer certain rule sets over others. This is mainly down to personal choice but it is **extremely** encouraged to follow a rule set, otherwise you will not have a good time. There are too many rules to go over here, so we will give you this resource to read up on. Not everything might make sense now but as we move on continue to refer back to it.

There are three points of syntax we will touch on specifically.

White Space

White space is spaces " " just like that. C# is a white space insensitive language.
Meaning :

```
        void     Start                    ( )  {
Debug  .    Log     ("Hello World"    )      ;
                         }
```

That will run just as well as what we had above.
Just remember that if you ever code like this and someone tries to use your code they might try to snuff you out.
Just because you can does **not** mean you should.

*Note that inside of quotes `""` **is** white space sensitive. You'll find out why in the next lesson.*

Case Sensitivity

Case or capitalization is important in C#. `Debug.Log()` is **not** the same as `debug.log()` or `deBUG.loG()`.
Almost every pre-written function in Unity is written in "UpperCamelCase".
UpperCamelCase is a type of CamelCase. CamelCase is a naming convention in which words are written with no spaces and instead use capitals to denote each word.
UpperCamelCase just means that the first letter is capitalized along with the first of each subsequent word.

So `my code program` is written in UpperCamelCase as `MyCodeProgram`. As you'll notice this is the same as Unity's functions like `Start` or `UnityEngine`.
It is typically recommended to use this yourself in order to have your code look the same as to avoid confusion.

Now that is for Unity's functions, classes, and the likes. Unity does use lowerCamelCase for its variables. We will talk about what variables are in the next lesson. For now just know they are written in lowerCamelCase. You can likely guess what that means.

A variable we wanted to call `my great variable` would actually be called `myGreatVariable`. We'll leave the examples alone for now. Again it is usually recommended to follow these guidelines.

Comments

We already talked about comments in the last lesson but will mention them again.
Comments are invaluable to keep track of code you write. You might think you can remember it all but trust us, you can not.
Use comments to quickly tell you what a block of code, or function, does. You don't need to use it for obvious things.

Good : `// Returns the average of all elements in an array`

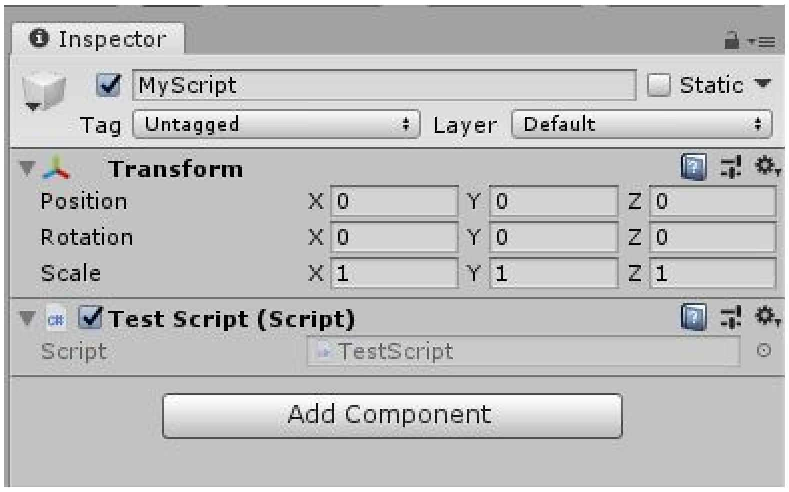
Bad : `// Adds two numbers` or `// blahh it does some stuff`

All the code written in these lessons will be using a preferred syntax. It's recommended to use it as well until you understand enough to figure out what you'd prefer and set up your editor to help.
Just follow along with my spacing and indenting but remember it does not need to be exact. You'll pick up these skills along the way.

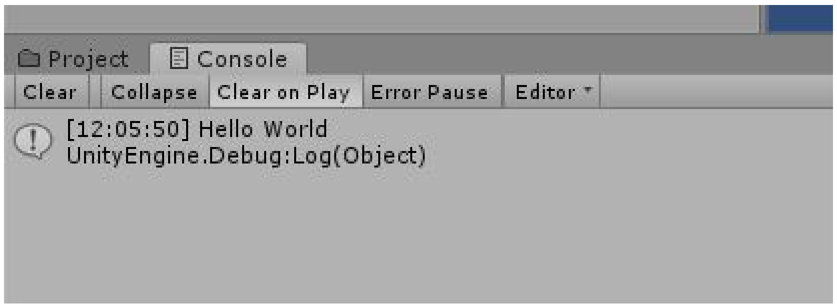
Here is a little write-up of the syntax used in these courses if you'd like to use it.

Adding Scripts To Objects

Save this script and return to Unity.
Now we need to put our script into the scene. We do this the same way we put a component on our object.
Let's just create a new "Empty Object" and give it a name.
Click onto it and add our script to it. Just like you would with the collider. The name of the component will be the same you gave the script.
It should look like this after :



Now hover over "Window" along the top bar, go down to "General", and then find and click on "Console". This brings up the Console which is where Unity will type messages for us from errors, code, and the likes.
With this open, run the game. You should see :



```
[12:05:50] Hello World
UnityEngine.Debug:Log(Object)
```

And there you go. You've written code and passed the age-old tradition of saying "Hello World".
Now how did we do this? This might seem like a lot of info but trust me, once you get it all on paper you can break it down easier.
Let's start small and work upwards.

How This Works

What does `Debug.Log("Hello World");` actually do? There's four parts to this :

1. `Debug` is a class in Unity. We mentioned classes in the last lesson. In this case the class `Debug` contains lots of different functions we can use to "debug" or find and fix problems in our game.
   To use the `Debug` class simply type it and make sure you have `using UnityEngine;` at the top of your code.

2. `Log()` is one of the functions in the class `Debug`. In this case the `Log` function "prints" out text into the Console that we opened up a moment ago.
   Because this function is part of the `Debug` class we need to call from the class. What this means is if were to just call `Log("Hello World");` our code would error. Feel free to try this if you want to see for yourself.
   They way we call `Log` on `Debug` is by using the period `.` or the dot as it is commonly referred to. The dot allows us to "access" function or variable from something like a class.
   When we write `Debug.Log` we are calling the `Log` function that is part of the `Debug` class.
   We also have the brackets, `()`. These are similar to the curly braces but rather than containing the code to make up a function or class they contain the values "passed" into our function.

3. `"Hello World"` is a "string" passed as a value into the function `Log`. These values must be within the brackets `()` of the function.
   In the next lesson we'll talk about what string means so for now just call it a value. A function can take values and use them to do just about anything, in our case write out the value.
   The quotation marks `""` are specific to strings and not any value passing into a function.

4. `;` denotes the end of a command. A command can be setting variables, functions, and a bunch more. Typically, every line of code (not of the text editor) needs to end with a semicolon.
   Obviously there is not a semicolon after the curly braces. We should hesitate to say this is a rule but it is the norm.

As shown above we have looked the structure that upholds the rules of code. Code not written with these rules will error and not run.
In the next lesson we will go over some more of the building blocks of code and will begin to manipulate data.