Coding and Scripting Introduction

Lesson Goals

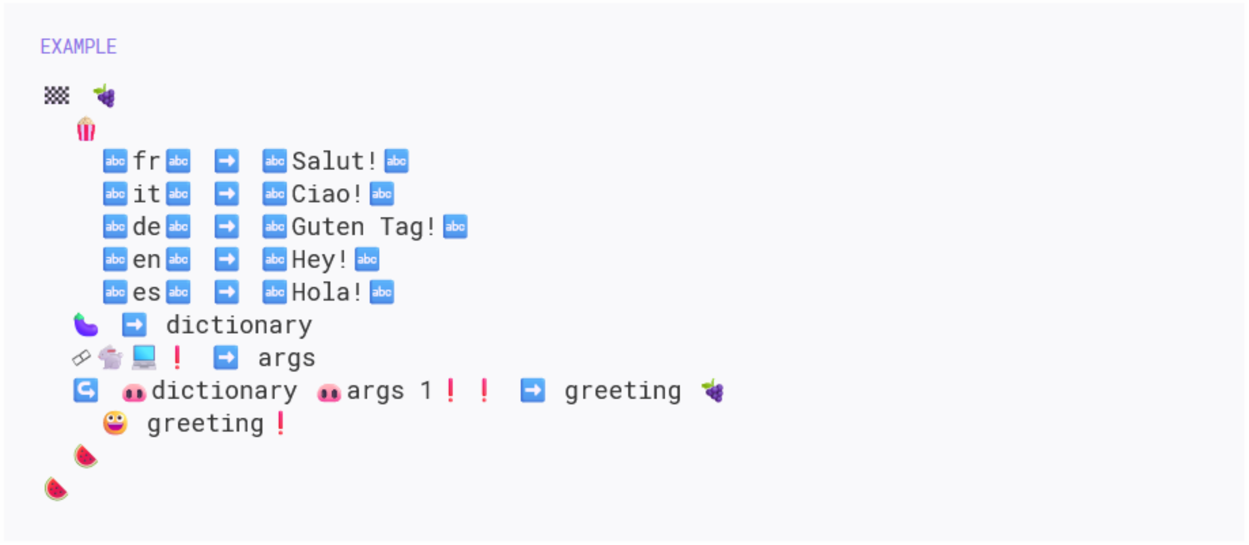
- Introduce what coding/scripting languages are and how they are used for game development.
- Learn how to read and begin writing C# code.
- Understand what a class is and how we can create our own.
- Understand what functions are, how we create them and why we want to use them.
- Learn about the variable types: Int, Float, Bool, Strings, Vectors, Arrays
- Understand the order in which our code will execute.

What is a scripting language?

1. A coding/scripting language is just like any other language, it has a set of rules we must follow in order to speak it, often this is referred to as grammar.

2. The are a variety of coding/scripting languages such as C, C++, Java, Python, Ruby, HTML and so many others. All languages have their own grammar/rules to them, so even if they share similar concepts, they are not the exact same.

3. To demonstrate this let's look at a few oddball languages. The first is Emojicode, a language that is written using emojis.



```
EXAMPLE
🏁 🍇
  🍿
    🔤fr🔤 ➡️ 🔤Salut!🔤
    🔤it🔤 ➡️ 🔤Ciao!🔤
    🔤de🔤 ➡️ 🔤Guten Tag!🔤
    🔤en🔤 ➡️ 🔤Hey!🔤
    🔤es🔤 ➡️ 🔤Hola!🔤
  🍆 ➡️ dictionary
  🔗🐇💻❗ ➡️ args
  ↪️ 🐽dictionary 🐽args 1❗ ❗ ➡️ greeting 🍇
    😀 greeting❗
  🍉
🍉
```

The second is Malbolge, a programming language designed to be a difficult to use as possible and was named after the 8th circle of hell in Dante's Inferno. Here are some examples of what the code looks like.

4. Luckily for us Unity uses the language C# for programming. C# is an offshoot of the language called C that opts to bring over some functionality from another language called Java.

5. We use coding/scripting languages to give instructions to the objects in our game. In order to write these instructions, we must first create a script.

Scripts and Creating a New Script

1. A script is simply something that holds the code we write for our game.

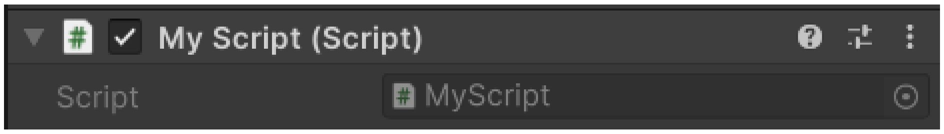
2. Again, we can think of a script as a place where we write our set of instructions for our game objects. Based on what we write our game object will try its best to read the code and execute the instructions.

3. Let's go ahead and create a script and attach it to a game object. Before we do, let's create a folder in our project called **Scripts**. Every script that we create will go inside this folder. Right click inside the **Assets** folder and then go to **Create > Folder**.

4. Inside the **Scripts** folder, right click again and go to **Create > C# Script**. Name this script **MyScript**.

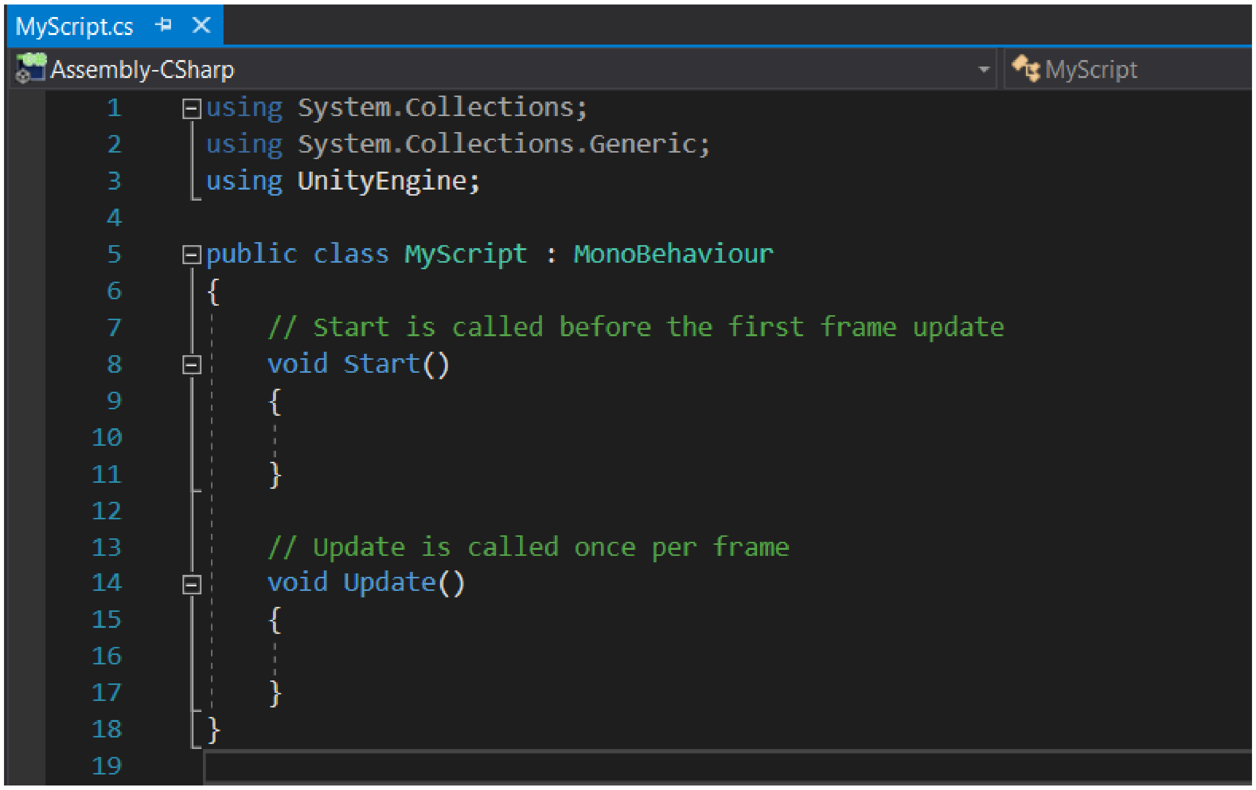
5. Before we start scripting, we need to attach this script to a game object. If a script is **not** attached to an object, **it will not run**. In our scene let's add in a **Cube** and name it **ScriptTester.** We should then be able to **drag** the **MyScript** script on to the object.

   If we click on the ScriptTester object and look at in the inspector we can see that the script is now attached to the object.



6. Let's now open up the script by double clicking on it. Once we do, the program Visual Studio should open up. You may need to go through some first-time setup options but once those are out of the way you should see something similar to the

image below.



MyScript.cs

Assembly-CSharp

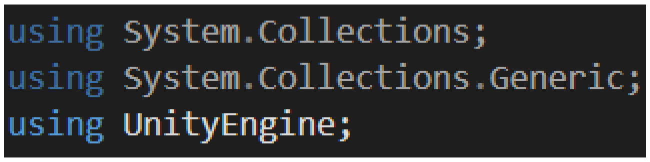
MyScript

```
1   using System.Collections;
2   using System.Collections.Generic;
3   using UnityEngine;
4
5   public class MyScript : 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  }
19
```

Script Breakdown and Classes

1. So, let's break down just what we are looking at here. First, we should notice at the top we can a tab with the script that we are currently working on. We should also notice that off to the side is a list of numbers. These numbers correspond to each line of code. Usually we want to know what line of code we are working on in case there is an error that we need to fix.

2. Let's next take a look the section at the top of the script.



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

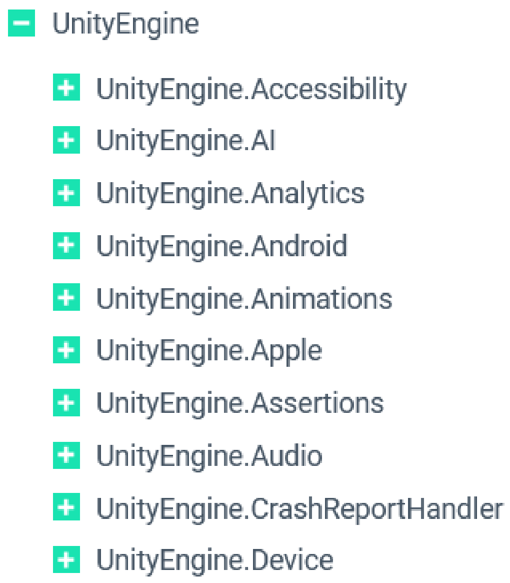
These are what is known as **Libraries**. A **Library** is a collection of data that we want our script to have access to. The data included in libraries are usually variables, functions or class definitions.

3. Let's look at the `using UnityEngine;` line.

The keyword **using** is blue. A **Keyword** is something in our code that has predefined functionality. We can't reuse the keyword throughout our code because of its functionality, if we do, we will get an error.

We will learn more keywords as we move along but a good rule of thumb is that any variable or class name is a keyword and should be avoided when used for a name.

4. The word **using** just means that we are including a library in our script. We then follow it up with the name of the library we are using. In our case it is **UnityEngine**. To see what the library is adding we can go to the **Unity Documentation**.

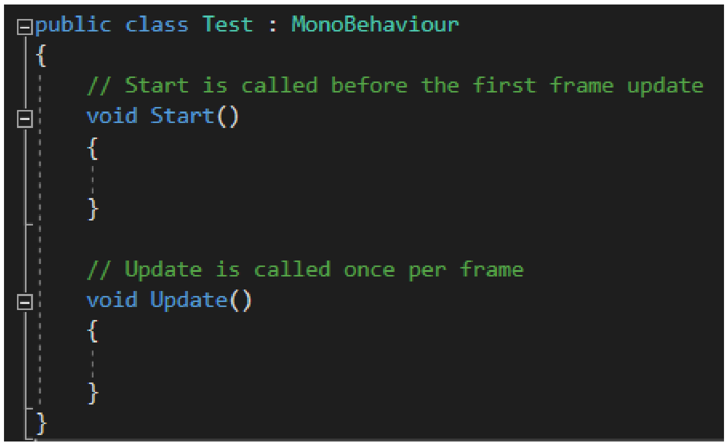


- UnityEngine
  - UnityEngine.Accessibility
  - UnityEngine.AI
  - UnityEngine.Analytics
  - UnityEngine.Android
  - UnityEngine.Animations
  - UnityEngine.Apple
  - UnityEngine.Assertions
  - UnityEngine.Audio
  - UnityEngine.CrashReportHandler
  - UnityEngine.Device

This is just a small snippet at what is included in the library.

We won't be touching on libraries too much now, but we might need to add in more as we make different projects.

5. Moving down in our script let's look at this chunk of code:

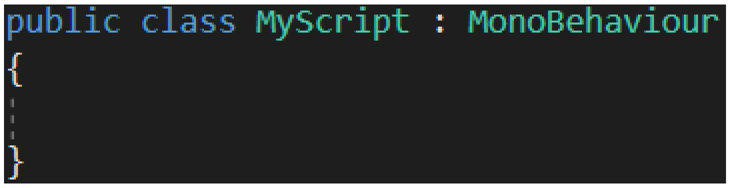


```
public class Test : MonoBehaviour
{
    // Start is called before the first frame update
    void Start()
    {

    }

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

    }
}
```

This is what is known as a **Class**. A class is what will hold all the information for an object. Typically the information the class holds are **variables** and **functions**. Our class right now actually already comes with two functions, these being **Start** and **Update.**

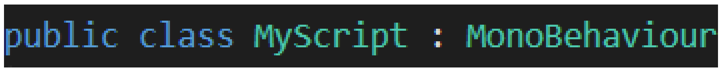
We will be getting into what variables and functions do later. For right now, a good way to think about classes is that they are just a type of variable that we can create ourselves. I know that is is a little strange that we are defining a class as something we haven't defined yet but just keep this idea in your mind and it will all makes sense later.

6. Let's now break down the code and see what our class is doing. We are going ignore the two functions that are inside of our class, so we can act as if we are ust looking at this:



```
public class MyScript : MonoBehaviour
{

}
```

7. Let's take a look at the line below:



```
public class MyScript : MonoBehaviour
```

This is what is known a **Class Definition**.

8. We first define the **Protection Level** of the class. This is what the `public` keyword does. We will talk about protection levels later when discussing variables.
9. We then tell the script the we are declaring a class with the `class` keyword and that we are naming our class `MyScript`.

10. Next we are making our script inherit from **MonoBehavior**, that's what the `: MonoBehaviour` part of the class definition is doing.

    Monobehavior is a class that is included in the UnityEngine library from earlier. To inherit code, other wise called **Inheritance,** is a key concept in programming. We won't use it much now, but it allows the class to have access to the functions and variables of whatever we are inheriting from.

11. Lastly, we just have the Curly Braces.

```
{

}
```

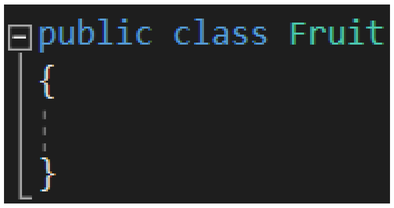
This is what will contain the data for our class and can otherwise be thought up as the body of our class.

A key concept with curly braces, parentheses or brackets in programming is that if you have an open curly brace, parenthesis or bracket you must also have a closing one.

You should also notice too that there is a dashed line that is running down to connect each parenthesis. This dash line helps us line up the parentheses. We want to make sure they are aligned with each other so that our code is easy to read.

Creating Our Own Class

1. So now that we understand what a class is let's create our own. Below the closing curly brace in our script lets write the code:



```
public class Fruit
{

}
```

2. And just like that we are done! We will come back to this class to add more in a little bit but for right now we just have a class called **Fruit** that is not inheriting from anything.

Functions

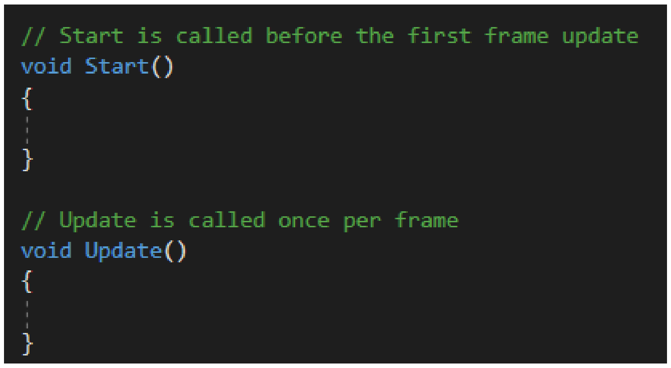
1. Next, we need to talk about functions. If you remember we skipped over talking about these when discussing our MyScript class. To put is simple a **Function** allows us to organize our code into small digestible parts.

   We can take the functions we create and do what is known as a **Function Call**. A function call tells the script to jump to where the function is written and run the code inside of it.

If we didn't use a function call, we would need to write the same code over and over and over again. So, by performing a function call we essentially create code snippets that can be reused. This effectively cuts down on the code we need to write.

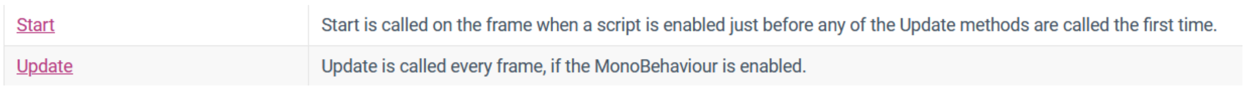
Lastly functions come in two main types. Function that returns a value and functions that do not. We will talk more about this later. It is also worth noting that we are able to pass in data to function for them to use. Again, we will cover this later.

2. Lets go back to our MyScript class and look at the functions within it.



```
// Start is called before the first frame update
void Start()
{

}

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

}
```

These functions come packaged with the MonoBehavior class. Taking a look at the Monobehavior documentation, we can see what they do and when Unity will call each function.



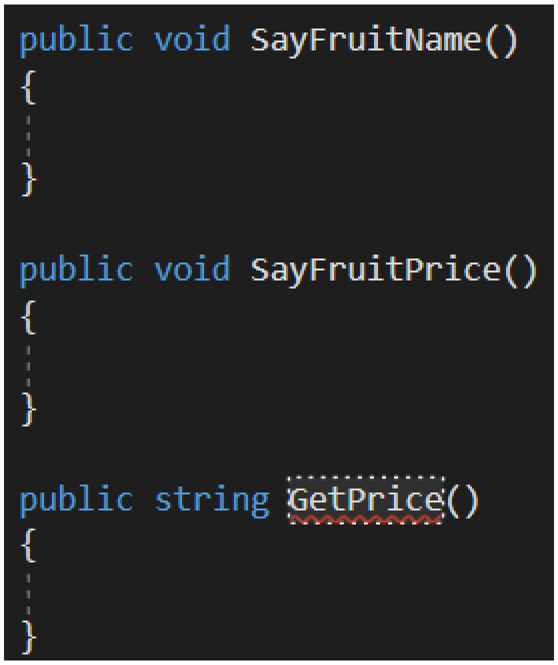
Start	Start is called on the frame when a script is enabled just before any of the Update methods are called the first time.
Update	Update is called every frame, if the MonoBehaviour is enabled.

It is also **important** to note that there are a lot more functions that come prepacked with the Monobehavior class. It is our job to figure out when to best use each function so be sure to look at them all!

One thing to allude to now is what is known as **Execution Order**. This is referring to when each built in unity function in our script will run. We will be coving this as well as how our script will run at the end of this lesson.

Creating Our Own Functions

1. Let's head back to our Fruit class and add a few custom functions. Make sure to put these functions within the body of the class or else these functions will not belong to the fruit class.



```
public void SayFruitName()
{

}

public void SayFruitPrice()
{

}

public string GetPrice()
{

}
```

2. We first define the **protection** level but again we are not worrying about that just yet. So next, we then determine what function will return. **Return** means that our function will run the code inside of it and then output a value.

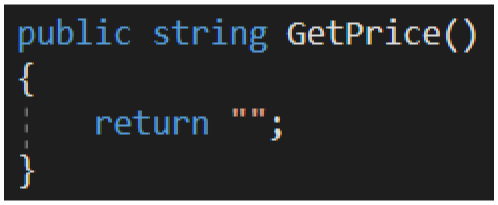
3. Out first two functions we use the **void** keyword. This means this function will return nothing. But our last function uses the **string** keyword, meaning it will return the value with the string type.

   We can return more value then just the string type however. We are allowed to return any type as long as that type has been defined.

4. Lastly, we just give the function a name that reflects what the function will do and then follow that name with a pair of parentheses.

   If we wanted to, we can pass in data to the function by putting the data inside the parentheses but for right now we will not pass in any data to keep things simple.

5. We will also see that we are getting an error with the last function. That is becaue we are not returning anything. So to fix this let's write the code:

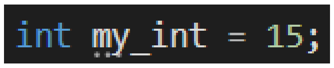


```
public string GetPrice()
{
    return "";
}
```

As you can see we use the **return** key word to return and empty string. The empty string is defined as a pair of double qoutation marks. We will get into this more when we talk about variables.

Variables

1. Let's now shift our focus to variables. In the MyScript class, at the top of the body lets create a new variable with the code:



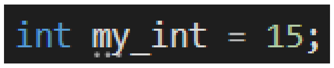
```
int my_int = 15;
```

This is what is known as a **Variable.** Variables allow us to store different types of information that we can use throughout our code.

Taking a look at the variable we created, we first define what type of variable it is by writing int and then we give it a name my_int. We then set it's value to 15. So in full, we declared a variable of type **int** with a name of **my_int** with a value of **15**.

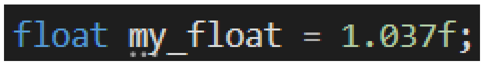
2. Let's now take the time to discuss some of the different types of variables that we may come across when programming. **Let's add the variables listed below to the top of our body in our MyScript class.**

Int – These variable stores integers, hence the name. Integers are whole numbers that do not have decimal places.



```
int my_int = 15;
```

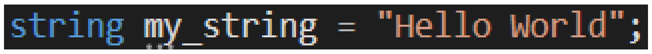
Float – Floats are numbers that have a decimal place. We want to use them when we need precise values, such as the player's position.



```
float my_float = 1.037f;
```

We put a **lower case f** at then end to signify the number is a float.

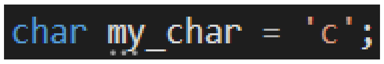
String – Strings are used to hold a series of characters. We need to make sure that our strings are enclosed in quotation marks “” or else we will get an error.



```
string my_string = "Hello World";
```

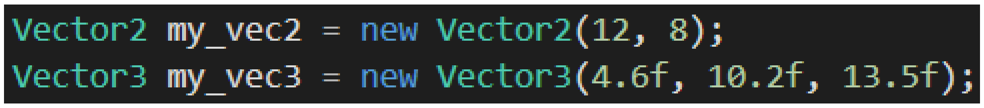
This is the type of variable that our GetPrice() function returns in our Fruit class.

Char – Char is short for character. As mentioned above these are what make up strings. However, if we just need a single character we can use this type of variable instead of a string. We need to make sure that the char is enclosed in single quotation marks ‘’ and that there is only one character assigned. If not, we will receive and error.



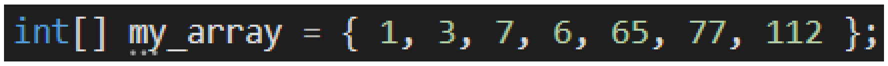
```
char my_char = 'c';
```

Vectors – Vectors are one of the most important types of variables for game development. They can be used to hold positions, determine the length of something or store a direction and so much more. The variable itself is used to store more than one number. For our projects we will be primarily using **Vector2** and **Vector3** type variables. Vector2 stores two numbers and Vector3 stores three numbers.



```
Vector2 my_vec2 = new Vector2(12, 8);
Vector3 my_vec3 = new Vector3(4.6f, 10.2f, 13.5f);
```

Array – Arrays are ways to package variables together. We can think of an array as a container with a set number of slots. We assign variables to each slot and then access the slot by using the number of the slot. We do this using square brackets [].

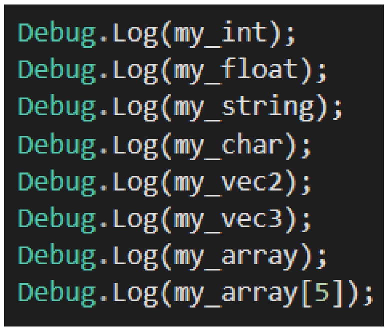


```
int[] my_array = { 1, 3, 7, 6, 65, 77, 112 };
```

Accessing a variable in an array may look something like **my_array[3]**. This will would return the value of **6** from the array above. Our array starts at a value of zero and would have a size of **7**. To get the last element of an array we could say something like **my_array[my_array.size - 1]**.

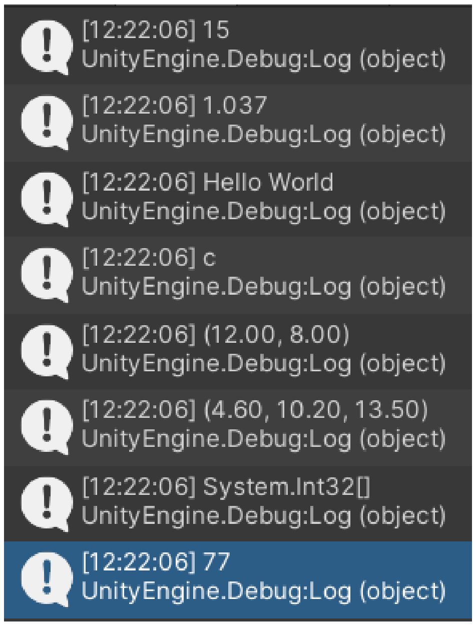
3. Now that we have our variables in order let’s write some code that will output them to Unity’s console.

In the start function, lets write the code:



```
Debug.Log(my_int);
Debug.Log(my_float);
Debug.Log(my_string);
Debug.Log(my_char);
Debug.Log(my_vec2);
Debug.Log(my_vec3);
Debug.Log(my_array);
Debug.Log(my_array[5]);
```

4. Save the script and run the game. We should then see the messages:



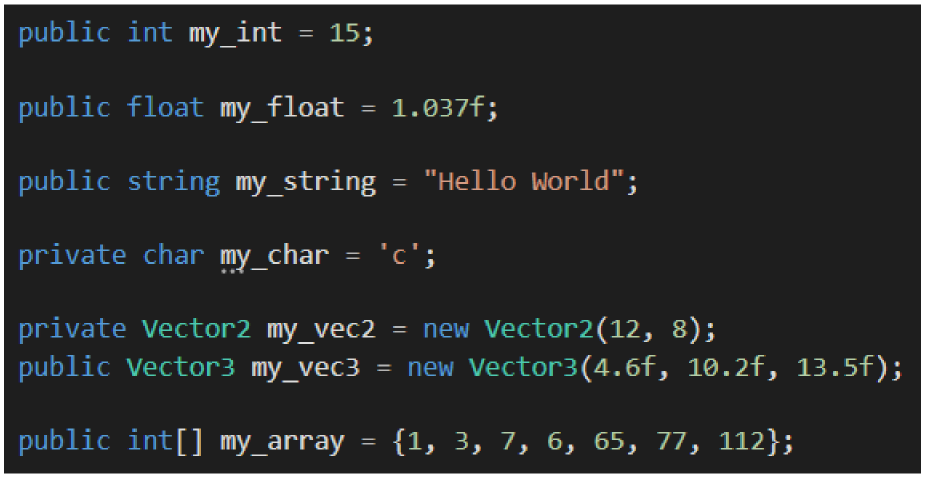
```
[12:22:06] 15
UnityEngine.Debug:Log (object)
[12:22:06] 1.037
UnityEngine.Debug:Log (object)
[12:22:06] Hello World
UnityEngine.Debug:Log (object)
[12:22:06] c
UnityEngine.Debug:Log (object)
[12:22:06] (12.00, 8.00)
UnityEngine.Debug:Log (object)
[12:22:06] (4.60, 10.20, 13.50)
UnityEngine.Debug:Log (object)
[12:22:06] System.Int32[]
UnityEngine.Debug:Log (object)
[12:22:06] 77
UnityEngine.Debug:Log (object)
```

If you do not see anything, be sure you attached your script to the ScriptTester object we made earlier.

5. Back in the script, let's finally talk about **Protection Levels**. **Protection Levels** determine how accessible variables and functions are to other scripts.

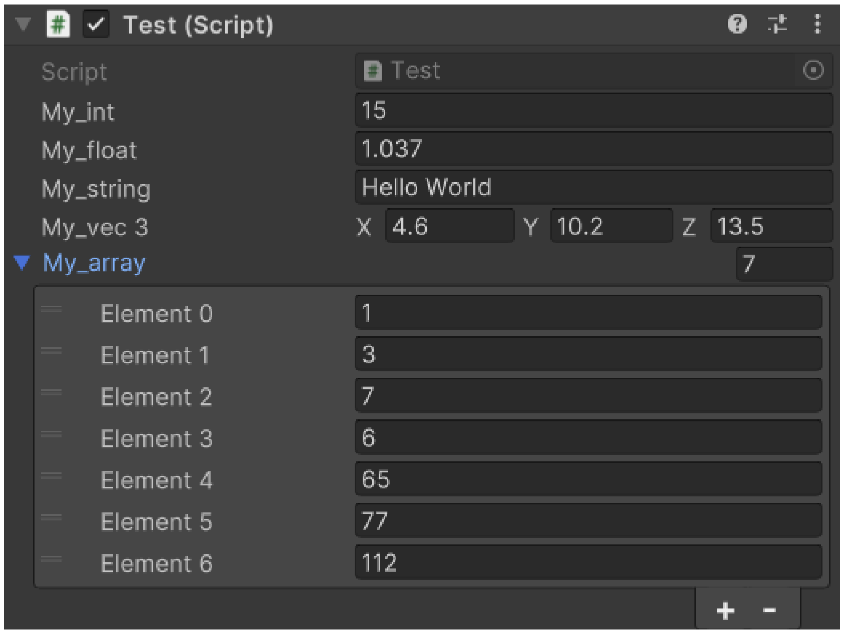
   We have already seen the `public` keyword when we were writing our functions but we can also introduce the `private` keyword. **Public** means the variable can be accessed by pretty much anything, this could be GameObjects or other scripts. While **Private** means that the variable can only be accesed in the class it was defined in.

6. Let’s add some protection levels to our variables.



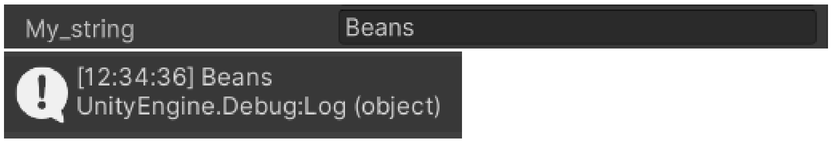
```
public int my_int = 15;

public float my_float = 1.037f;

public string my_string = "Hello World";

private char my_char = 'c';

private Vector2 my_vec2 = new Vector2(12, 8);
public Vector3 my_vec3 = new Vector3(4.6f, 10.2f, 13.5f);

public int[] my_array = {1, 3, 7, 6, 65, 77, 112};
```

Save the script and return to unity. If we find the **My Script Compoent** on the ScriptTester object we should be able to see the variables that we made public. However we will not see the variables we made private.



```
Test (Script)
Script        Test
My_int        15
My_float      1.037
My_string     Hello World
My_vec 3      X 4.6   Y 10.2   Z 13.5
My_array                       7
    Element 0     1
    Element 1     3
    Element 2     7
    Element 3     6
    Element 4     65
    Element 5     77
    Element 6     112
```

If we change the variables in the inspector and run the game, we will be shown the values we changed when the game is run. In the example the string “Hello World” was changed to “Beans” so “Beans” is what was ouput to the console.

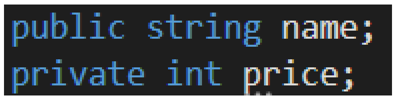


```
My_string     Beans
```

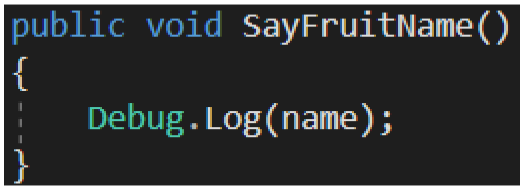
```
[12:34:36] Beans
UnityEngine.Debug:Log (object)
```

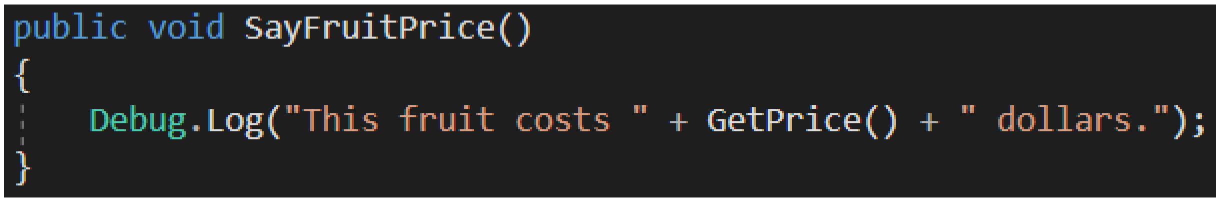
Finishing the Fruit Class

1. Now that we have discussed classes, functions and variables lets finish writing the code for our Fruit class.

2. Firstly, let's add the following variables:
   
   ```
   public string name;
   private int price;
   ```
   Name will be used to store the name of the fruit.
   Price will be used to store how much the fruit would cost.

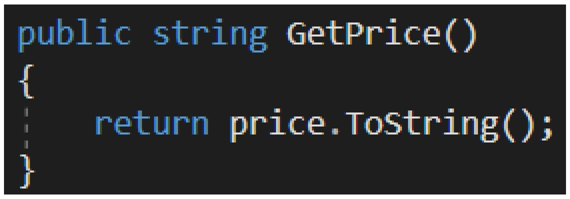
   Important: It is important to note that since these variables are in the Fruit class, the **Test** class does not know they exist. If we try to use them in the Test class, we will receive an error.

3. Next, let's have our functions actually do something. Let's fill out our functions with the code below:
   
   ```
   public void SayFruitName()
   {
       Debug.Log(name);
   }
   ```
   Our SayFruitName() function will output a message to the console with our fruit's name.

   
   ```
   public void SayFruitPrice()
   {
       Debug.Log("This fruit costs " + GetPrice() + " dollars.");
   }
   ```
   Our SayFruitPrice() function output one long string. However this string is made up of three individual strings:

   "This fruit costs "
   The string returned by the GetPrice() function.
   " dollars."

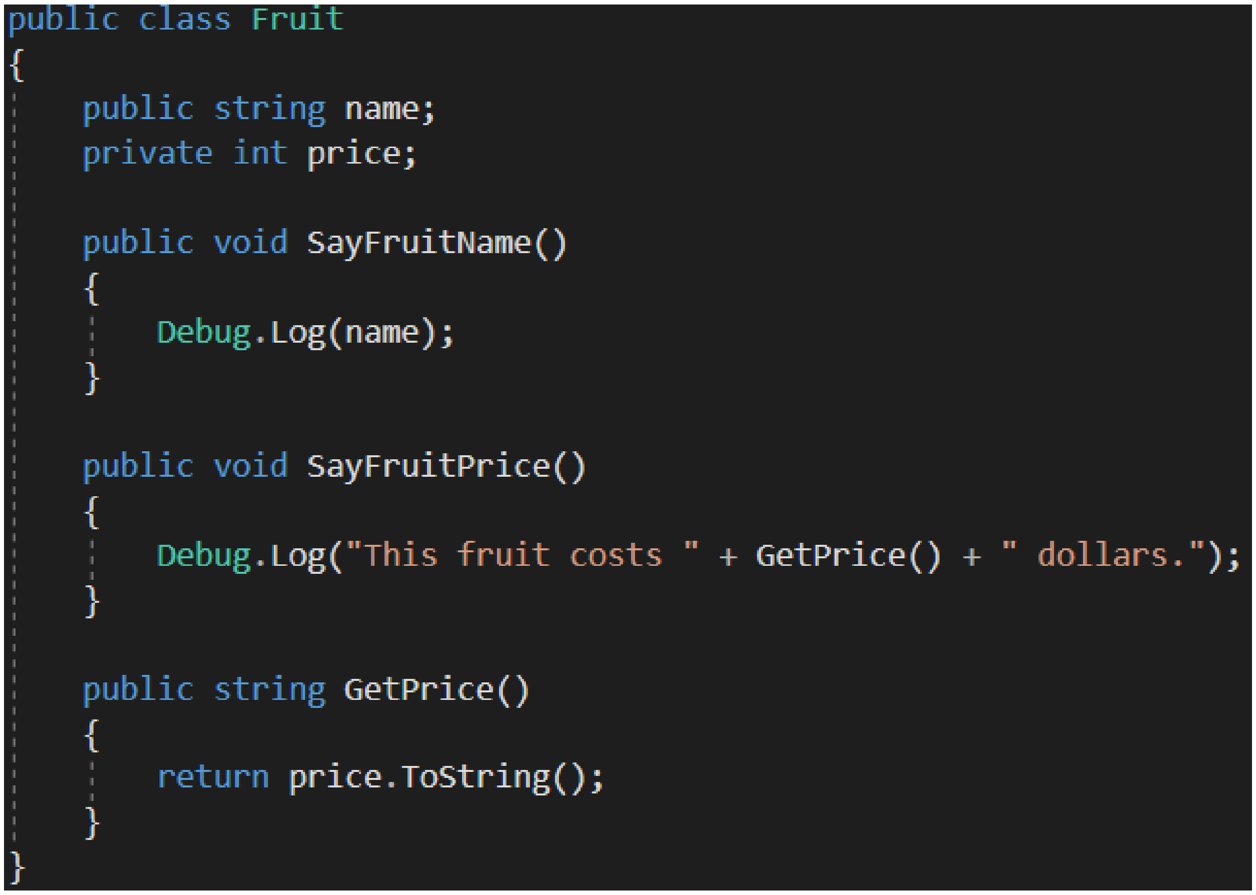
   We can add each string together using the plus sign to create a larger string. This is what is known as **String Concatination**.



```
public string GetPrice()
{
    return price.ToString();
}
```

Our GetPrice() function returns the price variable converted to a string.

4. Here is what the finished Fruit class looks like:

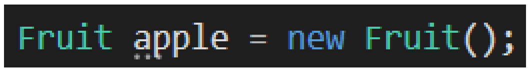


```
public class Fruit
{
    public string name;
    private int price;

    public void SayFruitName()
    {
        Debug.Log(name);
    }

    public void SayFruitPrice()
    {
        Debug.Log("This fruit costs " + GetPrice() + " dollars.");
    }

    public string GetPrice()
    {
        return price.ToString();
    }
}
```

In summation, our Fruit class has two variables, **Name** and **Price**. It also has three functions, **SayFruitName()**, **SayFruitPrice()** and **GetPrice()**.

Using the Fruit Class

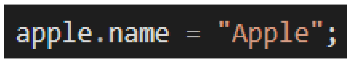
1. Now that our fruit class is complete, we can use it in the Test class. Up at the top of the test class lets add a variable with the type of Fruit.



```
Fruit apple = new Fruit();
```

We declare a new variable called **apple** of the variable type Fruit.

2. Then in the **Start()** function, lets remove all of our previously written code and then write the line:

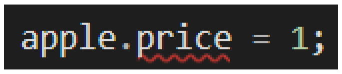
   ```
   apple.name = "Apple";
   ```

   Here we set our apple variable name value to “Apple”. **Remember:** Each varialble of type fruit will have a name variable inside of it. As well as a variable for price and the three other functions we wrote.

   We are able to set the name variable because we were able to access it using the **period.** The period is known as the **Accessor** in programing. It allows us to access the variables and functions that are within a class.

   We will be talking about the accessor more in a later lesson.

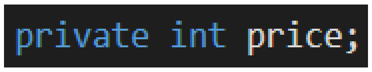
3. Next lets set the price of the apple with the code:

   ```
   apple.price = 1;
   ```

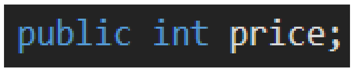
   However we seem to be getting an error with the price variable.

4. If we look in our fruit class, it seems like we are unable to access the price due to its protection level. Currently it is set to **private**.

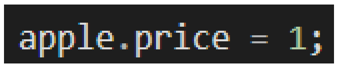
   ```
   private int price;
   ```

   So let’s change it to **public**.

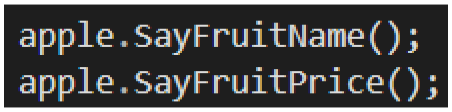
   ```
   public int price;
   ```

   This should then fix our issue.

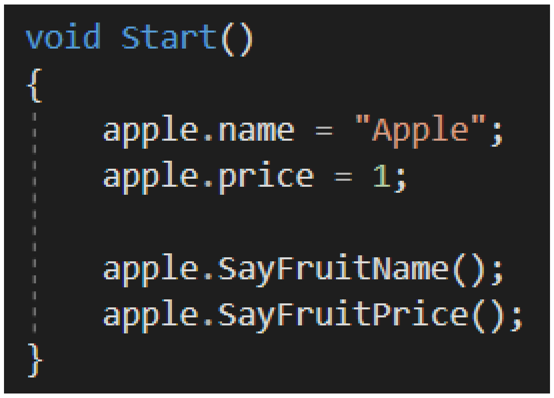
   ```
   apple.price = 1;
   ```

5. Back in the start fucntion lets then write the code:

   ```
   apple.SayFruitName();
   apple.SayFruitPrice();
   ```

6. Our full **Start()** function will look something like:



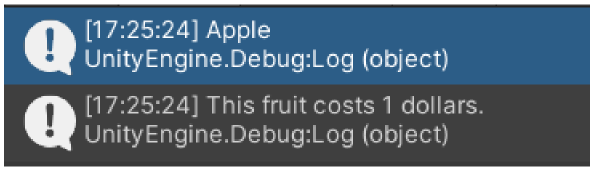
```
void Start()
{
    apple.name = "Apple";
    apple.price = 1;

    apple.SayFruitName();
    apple.SayFruitPrice();
}
```

   Before we run our game, let's step through the logic of our code. We first set the name of the apple variable to "Apple" and set its price variable to 1.

   We then call the apple variable's SayFruitName() function. This causes our code to jump from where it is to the SayFruitName() in the Fruit class. The function will then output the name of the fruit to the console.

   Once finished, it will jump back to where it left off. The code will then go through this logic again, except for the SayFruitPrice() function.

7. Save the script and run the game we should then see the console display:



```
[17:25:24] Apple
UnityEngine.Debug:Log (object)
[17:25:24] This fruit costs 1 dollars.
UnityEngine.Debug:Log (object)
```

Execution Order

1. Alright we are almost done but we still need to talk about a key concept that was mentioned before and that was **Execution Order**.

2. We actually just got a taste of this concept when we stepped through the logic of our Start() function. We started at the top of our function and then worked our way down. This is generally how the code of a script will run, it will start at the top and the run until it hits the bottom.

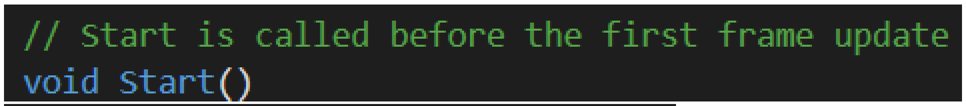
   We also performed some function calls too. Remember that when we perform a function call the script essentially jumps to where the function being called is. It then reads the code inside of it from the top down. Then when it hits then end, it returns back to where the code was originally being ran from.

Our function calls all happened in the same script, but it we were to call a function in a different script we would then hope over to that script to read from.

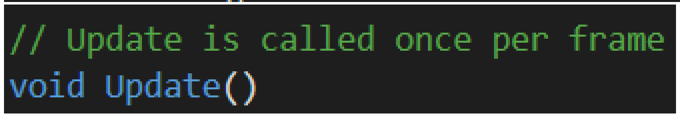
So, in summary, we will run our code from the top down. If we call a function, we jump to that function read what is inside of it and then go back to where we were.

3. However, there is still one last thing to talk about and that is the order in which Unity's built in functions run.

   We already saw two of these built in functions already. These being the Start() and Update() functions but you should have hopefully noticed some green text that was above them.



```
// Start is called before the first frame update
void Start()
```



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

   This green text tells us when these functions will run during gameplay. It is super important to know when each of these built in functions run, that way we can ensure that the code we write executes in the order we need it to.

4. Luckily for us, finding out Unity's execution order is pretty easy all we need to do is look at the Unity Documentation on execution order.

GitHub

1. Push the project to the GitHub repository.