Scripting Logic

Lesson Goals

- This lesson aims to look at the various ways we can control the logic of our game.
- We will learn about Arithmetic Operators that are useful for manipulating the numbers in our game.
- We will learn about Assignment Operators and how we can use them to set the values of variables.
- We will learn about Comparison Operators and how we use them to check for true/false conditions within our script.
- We will learn about Logical Operators and how we can use them to modify true/false conditions.
- We learn about the various types of statements we can use to control the logic of our script.
- We will then learn about the different types of Loops and how we can use them to repeat or automate parts of our code.

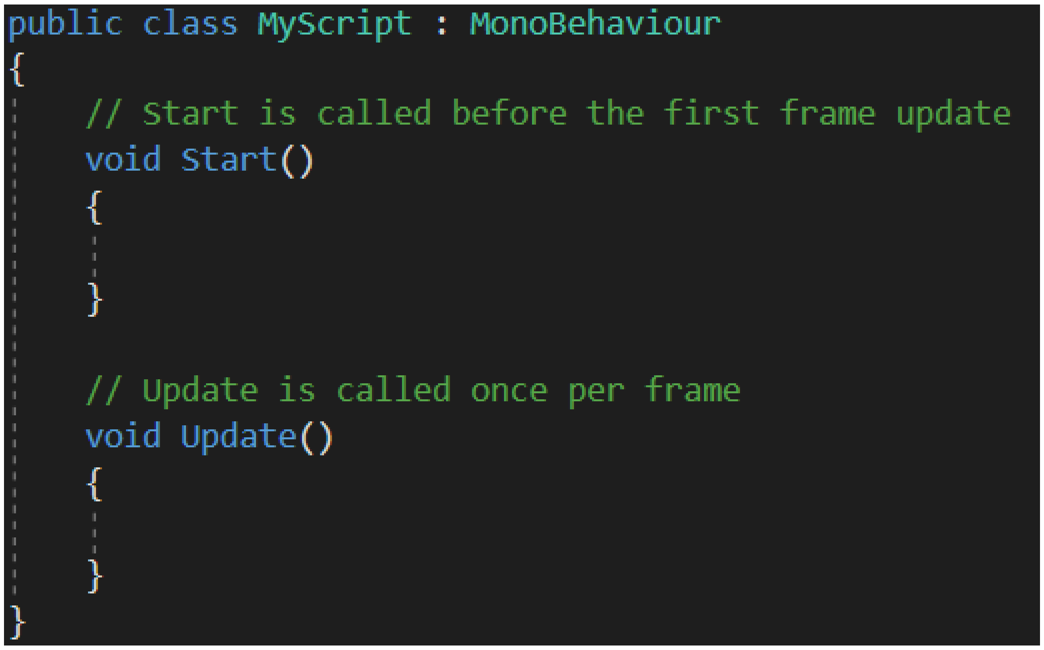
Set Up

1. Before we dive into this lesson, lets clear out a few things in our **MyScript** script we created in the last lesson.

2. First, in the MyScript class lets delete all the variables that we created.

3. Second, lets remove all of the code that is inside the start function.

4. Our MyScript class should look like:



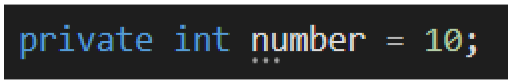
```
public class MyScript : MonoBehaviour
{
    // Start is called before the first frame update
    void Start()
    {

    }

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

    }
}
```

5. We are just removing our code from before so that we can start fresh. Except I wanted us to keep the fruit class that we made.

Arithmetic Operators

1. The first type of logic control that we are going to look at are the **Arithmetic Operators**. These are responsible for the manipulation of numbers in our scripts. So just think addition, subtraction, multiplication, division and so on.

2. In our script, in the MyScript class, let's declare a new variable.



```
private int number = 10;
```

3. The first batch of arithmetic operators should look pretty familiar, these are the **+**, **-**, ***** and **/**.

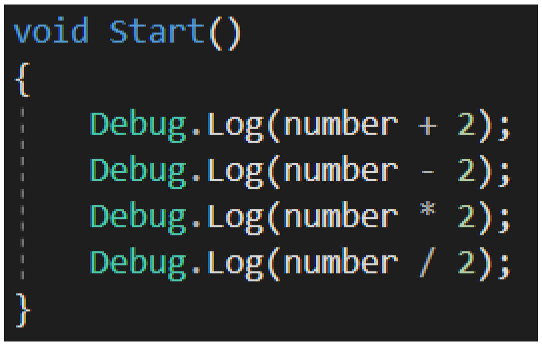
   The **Addition** operator, represented by the **+** symbol is used for addition.

   The **Subtraction** operator, represented by the **-** symbol is used for subtraction.

   The **Multiply** operator, represented by the ***** symbol is used for multiplication. In the past you may have used an **X** or a **•** as the sign for multiplication. These symbols actually have a different meaning in mathematics. **X** represents what is known as **Cross Product** and **•** represents what is known as **Dot Product.** So just know that moving forward we will be using the * symbol to represent multiplication.

The **Divide** operator, represented by the / symbol is used for division.

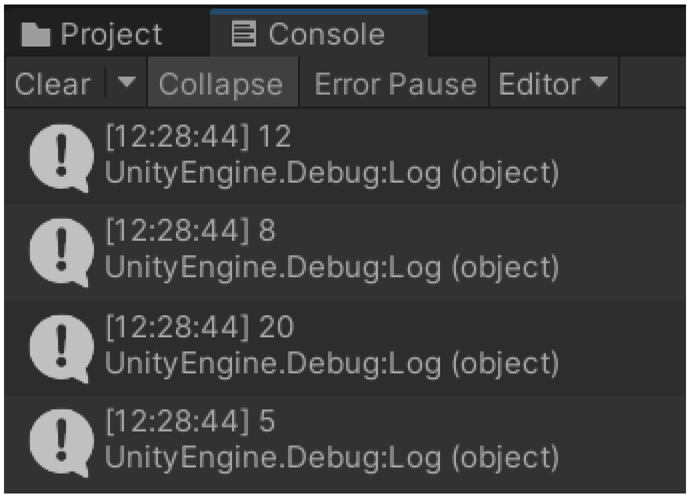
4. Let's go ahead and try out some of these operators. In our start function let's write the code.



```
void Start()
{
    Debug.Log(number + 2);
    Debug.Log(number - 2);
    Debug.Log(number * 2);
    Debug.Log(number / 2);
}
```

All we are doing is logging the result of our number when it is affected by each arithmetic operator by a value of two to the console.

5. If we save the script and run the game and check the console we should see:

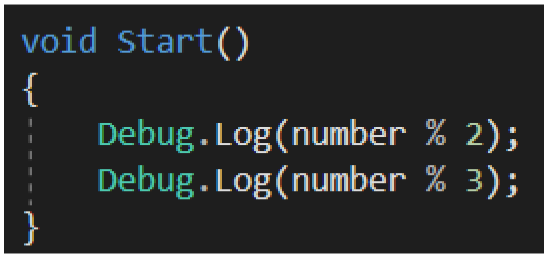


```
[12:28:44] 12
UnityEngine.Debug:Log (object)
[12:28:44] 8
UnityEngine.Debug:Log (object)
[12:28:44] 20
UnityEngine.Debug:Log (object)
[12:28:44] 5
UnityEngine.Debug:Log (object)
```

We can see how each operator effected our number.

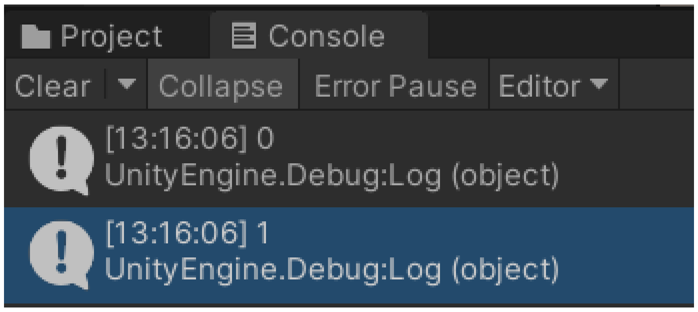
6. With the basics out of the way let's next take a look at the **Modulus** operator. The Modulus operator is represented by the **%** symbol. It is used to determine if a number is divisible by another number and will show the remainer of that division.

7. To see this operator in action lets change our start function to say:



```
void Start()
{
    Debug.Log(number % 2);
    Debug.Log(number % 3);
}
```

If we save and run the game we will see the messages:



```
[13:16:06] 0
UnityEngine.Debug:Log (object)
[13:16:06] 1
UnityEngine.Debug:Log (object)
```

In the first message we are checking to see if ten is divisible by two. We can see that it is because the first message returned is zero meaning that there is no remainder.

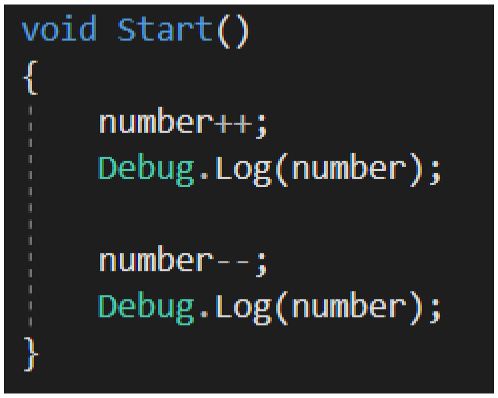
In the second message we are checking to see if ten is divisible by three. We can see that it is not because the remainer is one. We got this because the number three can only go into the number ten three times, this just leaves the number one which is the remainder.

8. The last two operators that we are going to look at are the **Auto Increment** and **Auto Decrement** operators.

   The **Auto Increment** operator is represented by the **++** symbols. This is shorthand for adding one to a value.

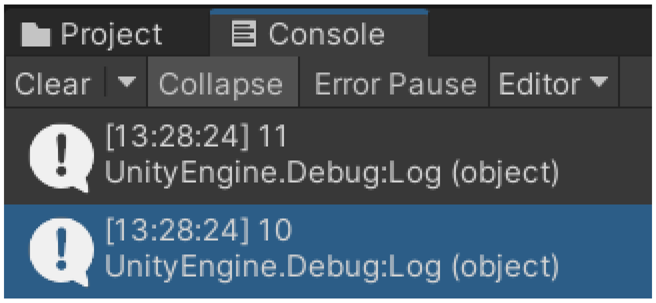
   The **Auto Decrement** operator is represented by the **--** symbols. This is shorthand for subtracting one from a value.

9. Let's change the code in our start function to say:



```
void Start()
{
    number++;
    Debug.Log(number);

    number--;
    Debug.Log(number);
}
```

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



```
[13:28:24] 11
UnityEngine.Debug:Log (object)
[13:28:24] 10
UnityEngine.Debug:Log (object)
```

The number variable was first increased by one using the auto increment operator, so when the first message was output, it read out 11. The number was then decreased by one using the auto decrement operator, so when output the message read 10.

Operators Overloading

1. Now that we have covered the Arithmetic operators, as a small aside, let's look at something goofy that we can do with them as well as other operators we will look at.

2. Let's create a new variable.

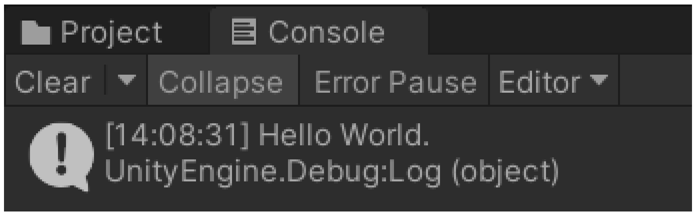
```
private string word = "Hello";
```

3. With the Arithmetic operators we have only been using an integer as an example but what if we were to use a string?

   In our start function let's write the code:

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

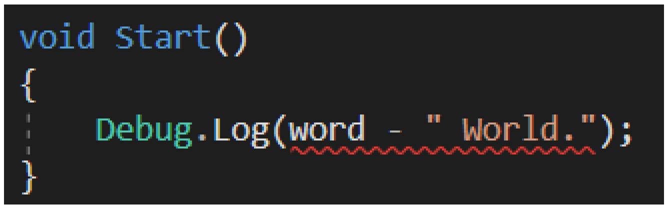
   Save the script and run the game, we should see the message:



```
[14:08:31] Hello World.
UnityEngine.Debug:Log (object)
```

4. I brought up in the last lesson. This is known as **String Concatenation**. This is where two strings can be "added" together. But now what were to happen if we try to subtract a string from another string.

   In the start function lets change out code to:



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

And it doesn't look like we can do that, we get an error. So now the question is why can we add two strings together but not subtract them?

5. Well, the answer is **Overloaded Operators**. To overload an operator means that we can **give the operator different functionality depending on how we want that operator to function** or **the types of variables we are using for the operator**.

   Sort of a double definition here, so I apologize for any confusion.

   But to answer our question, the reason why we are able to add strings together and not subtract them is because the + operator for the string class has been **overloaded**. Whoever was writing the string class decided that the user should be able to add strings together and defined that when the strings are added together the two strings will merge into one.

   In theory, since we create the fruit class, we could overload the operators for it. For instance, we could overload the addition operator so if we were to add an apple and an orange together it could result in the value of cherry.

6. Overloading operators is a little beyond the scope of what should be doing right now. But I wanted to make you aware of this concept to explain why we can add strings and other variable types together. Feel free to try and see what operator works with each type of variable.

Assignment Operators

1. With that aside out of the way let's look at the next type of operators known as **Assignment Operators**. These types of operators allow us to assign or set the values of variables.

2. There are five assignment operators total. They are:

   The **Equals** operator is represented by the **=** symbol. This operator is responsible for setting a value equal to another.

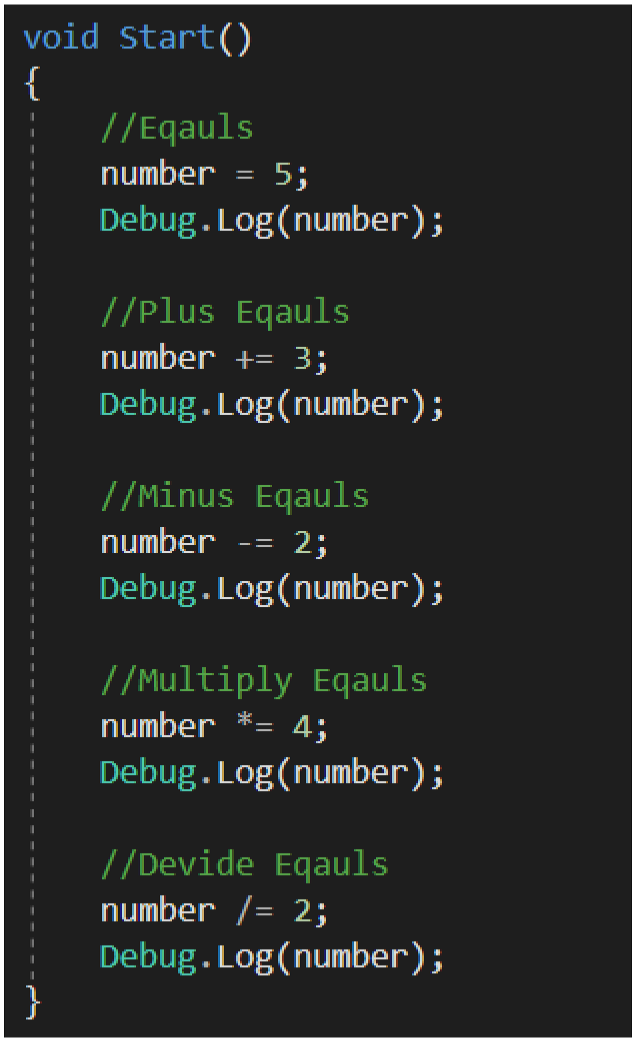
   The **Plus Equals** operator is represented by the += symbols. This value takes the current value of a variable and adds a value to it.

   The **Minus Equals** operator is represented by the -= symbols. This value takes the current value of a variable and subtracts a value from it.

The **Multiply Equals** operator is represented by the *= symbols. This value takes the current value of a variable and multiplies it by a value.

The **Divide Equals** operator is represented by the /= symbols. This value takes the current value of a variable and divides it by a value.

3. Let's take a look at these operators in code. In our MyScript script lets clear out what is inside of our start function and write the code:

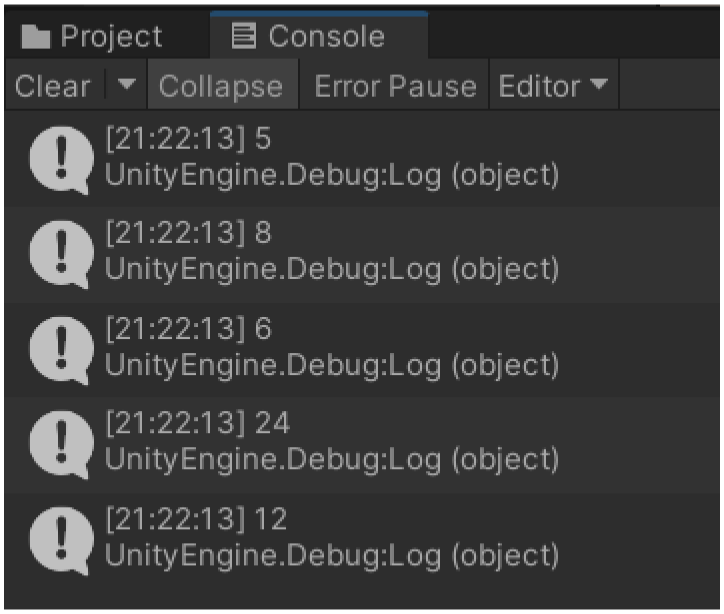


```
void Start()
{
    //Eqauls
    number = 5;
    Debug.Log(number);

    //Plus Eqauls
    number += 3;
    Debug.Log(number);

    //Minus Eqauls
    number -= 2;
    Debug.Log(number);

    //Multiply Eqauls
    number *= 4;
    Debug.Log(number);

    //Devide Eqauls
    number /= 2;
    Debug.Log(number);
}
```

Stepping through our code we first set the value of our number to five using the equals operator.
Using the plus equals operator, we add three to our number setting its value to eight.
We then use the minus equals operator to subtract two from our number setting its value to six.
We then use the multiply equals operator to multiply the number by four setting its value to

twenty-four.
Lastly, we divide the number two using the divide equals operator, giving us the final value of 12.

4. Saving the script and running the game will produce the messages:



```
[21:22:13] 5
UnityEngine.Debug:Log (object)
[21:22:13] 8
UnityEngine.Debug:Log (object)
[21:22:13] 6
UnityEngine.Debug:Log (object)
[21:22:13] 24
UnityEngine.Debug:Log (object)
[21:22:13] 12
UnityEngine.Debug:Log (object)
```

Comparison Operators

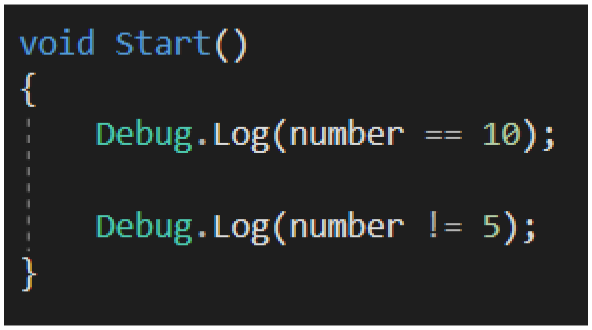
1. The next type of operators we are going to look at are the **Comparison Operators**. Comparison operators allow us to compare different variables to each other in a multitude of ways. It is important to remember that the result of using a comparison operator will be either **true** or **false**.

2. The first batch of comparison operators that we will look at relates to whether a variable is equal to or not equal to a value.

   The **Equal To** operator is represented by the **==** symbols. This operator is used to check if two values are equal to each other.

   The **Not Equals** operator is represented by the **!=** symbols. This operator is used to check if two values are not equal to each other.

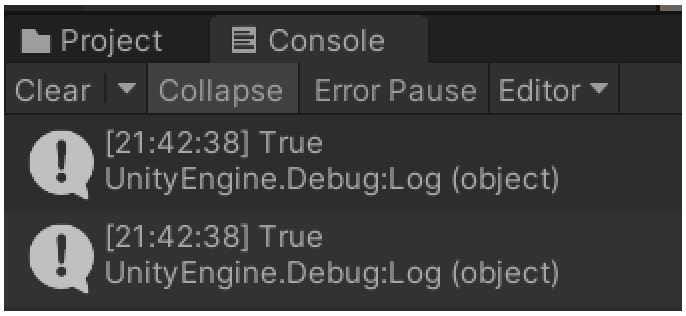
3. Once again, let's clear out start function in our script and write the code:



```
void Start()
{
    Debug.Log(number == 10);

    Debug.Log(number != 5);
}
```

Remember, for comparison operators the answer returned will be true or false. The first message output should be true, while the second message output will be false.

4. If we save the script and run the game, we get the message:



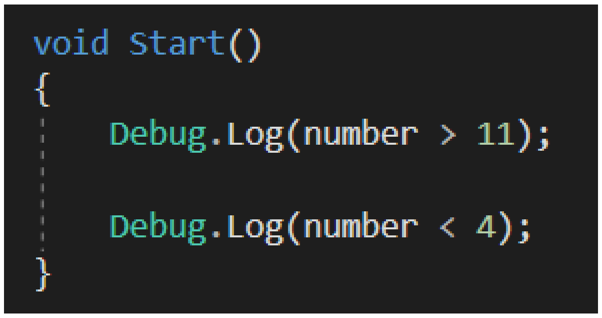
```
[21:42:38] True
UnityEngine.Debug:Log (object)
[21:42:38] True
UnityEngine.Debug:Log (object)
```

5. The next batch of comparison operators determine if a variable is greater than or less than a value.

   The **Greater Than** operator is represented by the **>** symbol. This operator checks to see if one variable is greater than another.

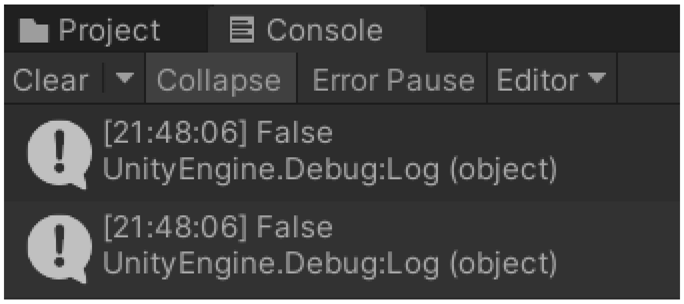
   The **Less Than** operator is represented by the **<** symbol. This operator checks to see if one variable is less than another.

6. Clear what is in the start and then write the code:



```
void Start()
{
    Debug.Log(number > 11);

    Debug.Log(number < 4);
}
```

Saving the script and running the game will produce the message:



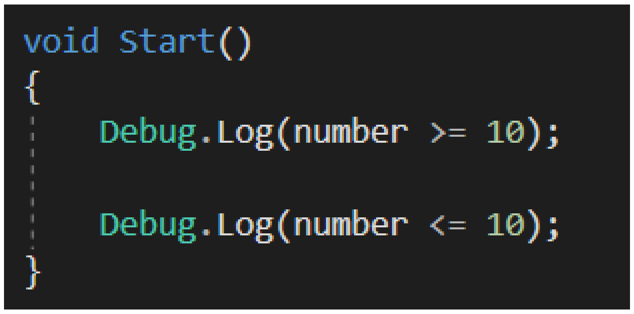
```
[21:48:06] False
UnityEngine.Debug:Log (object)
[21:48:06] False
UnityEngine.Debug:Log (object)
```

7. The last batch of comparison operators determine if a variable is greater than or equal to a value or less than or equal to a value.

   The **Greater Than Equals** operator is represented by the **>=** symbols. This operator checks to see if a variable is greater or equal to another value.

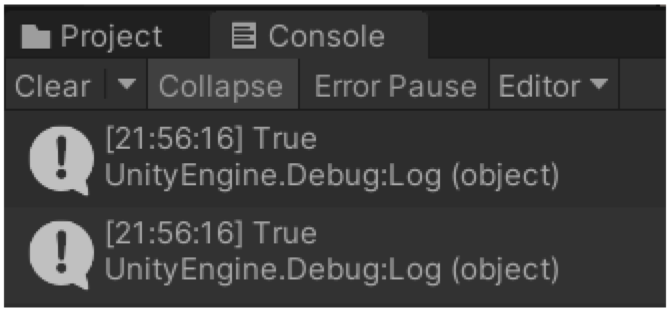
   The **Less Than Equals** operator is represented by the **<=** symbol. This operator checks to see if a variable is less or equal to another value.

8. Clear what is in the start and then write the code:



```
void Start()
{
    Debug.Log(number >= 10);

    Debug.Log(number <= 10);
}
```

Saving the script and running the game will produce the message:



```
[21:56:16] True
UnityEngine.Debug:Log (object)
[21:56:16] True
UnityEngine.Debug:Log (object)
```

Logical Operators

1. The last type of operator we are going to look at are the **Logical Operators**. They can be used to modify the true/false conditions as well as check for more than one condition at the same time.

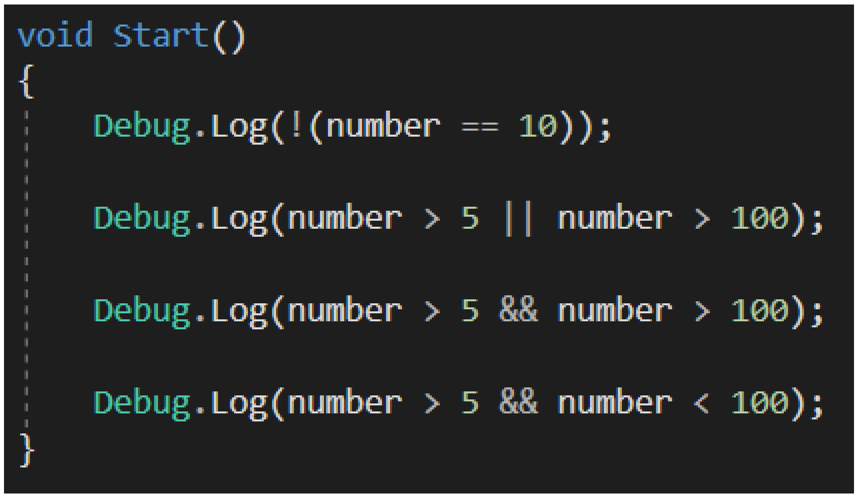
2. The types of logical operators are:

   The **Not** operator is represented by the **!** symbol. This operator inverts the true/false condition.

   The **Or** operator is represented by the **||** symbols. This operator checks for two or more conditions. It outputs true if one or both of the conditions are true.

   The **And** operator is represented by the **&&** symbols. This operator checks for two or more conditions. It outputs true if all of the conditions are true. If any of the conditions are false, it outputs false.

3. Let's take a look at how we can use these operators in code. Clear out the start function and the code:



```
void Start()
{
    Debug.Log(!(number == 10));

    Debug.Log(number > 5 || number > 100);

    Debug.Log(number > 5 && number > 100);

    Debug.Log(number > 5 && number < 100);
}
```

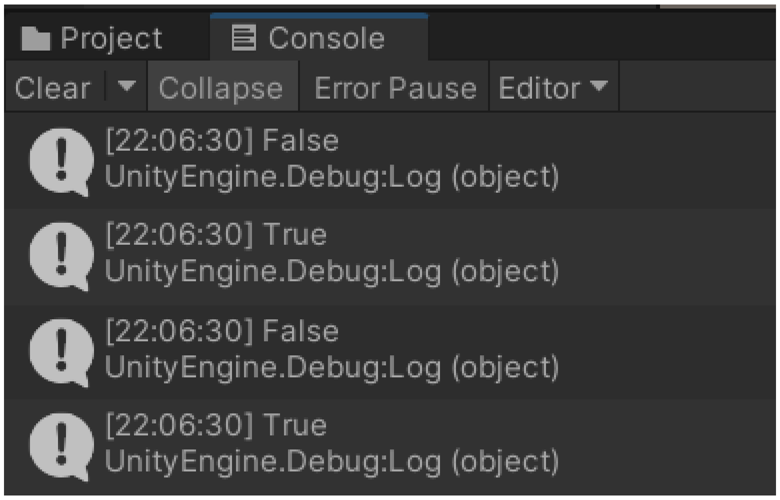
   Lets step through our code. The first message we output is wraps the condition in a pair or parentheses. We do this so that we can invert the whole statement. We know that the condition inside the parentheses will be true because our number is equal to ten. But because we have that **Not** operator outside of the parentheses, the condition becomes inverted, so it will return false.

   The second message we output checks two conditions. If our number is greater than five and if our number is greater than one hundred. Since we are using the **Or** operator we just need one of these conditions to be true. Since our number is greater than five, the whole statement will return true.

   The third message we output checks to conditions. If our number is greater than five and if our number is greater than one hundred. Since we are using the **And** operator we just need both of these conditions to be true. Since our number is not greater than one hundred, the whole statement is false.

The last message we output checks to conditions. If our number is greater than five and if our number is less than one hundred. Since we are using the **And** operator we just need both of these conditions to be true. Since our number is greater than five and less than one hundred, the whole statement is true.

4. Saving the script and running the game will output the message:



```
[22:06:30] False
UnityEngine.Debug:Log (object)
[22:06:30] True
UnityEngine.Debug:Log (object)
[22:06:30] False
UnityEngine.Debug:Log (object)
[22:06:30] True
UnityEngine.Debug:Log (object)
```

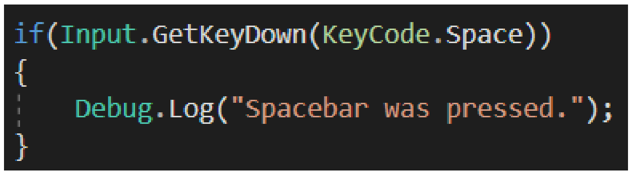
If Statements

1. The first statement type we will look at is the **If Statement**. An If Statement looks for its condition to be true, if it is then it will run the code inside of it.
2. For example, if the player presses the fire button, the player will shoot a bullet. Let's write some pseudocode to help visualize the example above.

```
If (player is holding down the fire button)
{
        Make the player shoot.
}
```

So, if the player is holding down the fire button, the if statement will register that as true. Then because it is true it will run the code to make the player shoot.

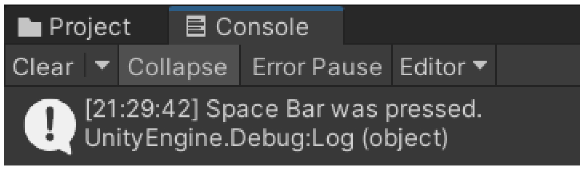
3. Let's try writing and example for ourselves. Let's open up the **MyScript** script, clear out the **Start()** function. Then in the **Update()** function let's write the code:



```
if(Input.GetKeyDown(KeyCode.Space))
{
    Debug.Log("Spacebar was pressed.");
}
```

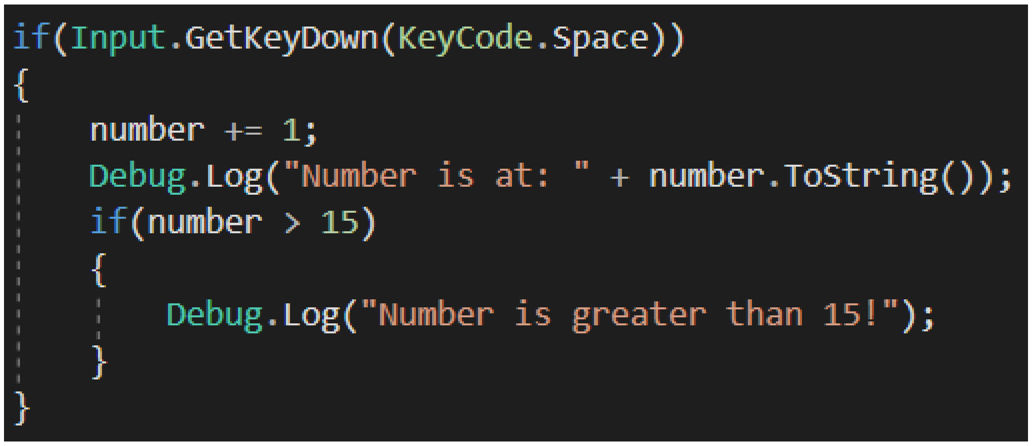
We won't be getting into the specifics of our code but just know that we are checking to see if the spacebar is being pressed and if it is we output a message.

4. If we run the game and hit the spacebar, we will output the message we wrote.



```
[21:29:42] Space Bar was pressed.
UnityEngine.Debug:Log (object)
```

5. Let's do one more example, lets change out the if statement to say:

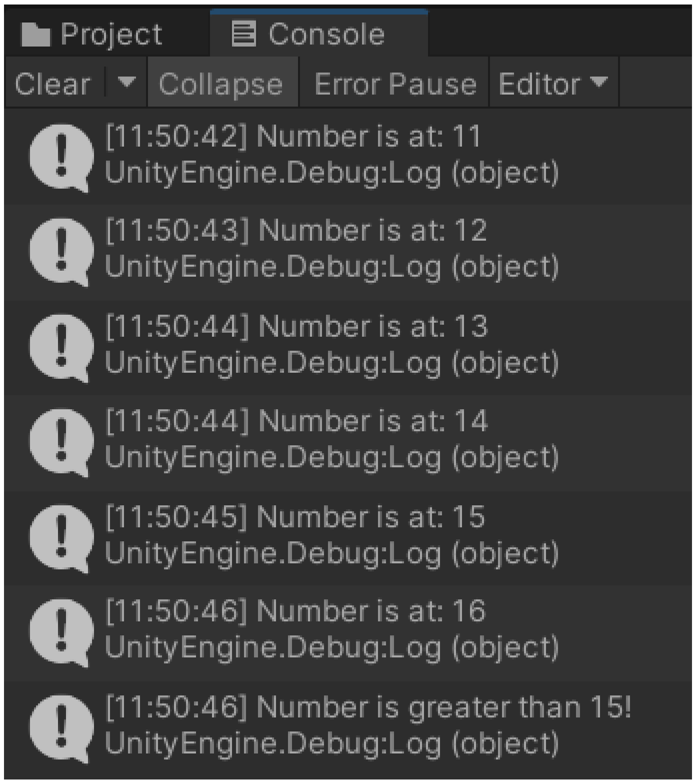


```
if(Input.GetKeyDown(KeyCode.Space))
{
    number += 1;
    Debug.Log("Number is at: " + number.ToString());
    if(number > 15)
    {
        Debug.Log("Number is greater than 15!");
    }
}
```

We are still checking if the spacebar was being pressed.
If it is, we add 1 to the number variable.
We then print output a message saying what the value of the number is.
Next we use another if statement to see if the number's value is greater than 15.
If it is we output a message saying that it is greater than 15.

6. Save the script and run the game, when the spacebar is pressed, we will see the message outputting the value of the integer. But we will only see the second if statement if the

integer's value is greater than 15.



```
[11:50:42] Number is at: 11
UnityEngine.Debug:Log (object)
[11:50:43] Number is at: 12
UnityEngine.Debug:Log (object)
[11:50:44] Number is at: 13
UnityEngine.Debug:Log (object)
[11:50:44] Number is at: 14
UnityEngine.Debug:Log (object)
[11:50:45] Number is at: 15
UnityEngine.Debug:Log (object)
[11:50:46] Number is at: 16
UnityEngine.Debug:Log (object)
[11:50:46] Number is greater than 15!
UnityEngine.Debug:Log (object)
```

7. Overall, I like to think about if statements as if we are asking the computer if a condition is true and if it is, an action is performed. If the condition is to see if the space bar was pressed is true, the action is to add one to our number variable.

Else

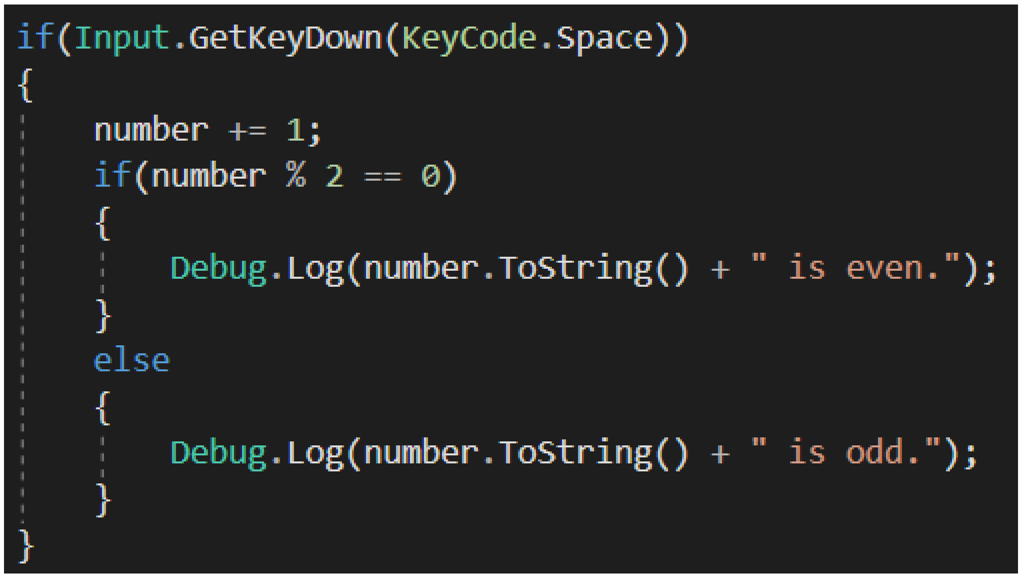
1. If an If Statement is asking the computer to check if a condition was true, an **Else Statement** is the response to if the condition was false.

2. Let's look at some pseudocode to see what this means:

```
If (a number is divisible by two)
{
        Debug.Log("The number is even.")
}
else
{
        Debug.Log("The number is odd.")
}
```

Here we check to see if a number is divisible by two, if it is we print out the message "The number is even." **Else** we print out the message "The number is odd."

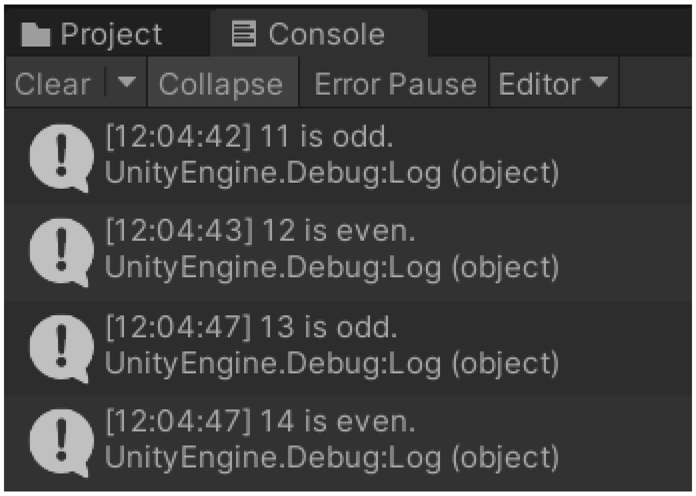
3. Let's try this out in our Update() function. Let's change the code to:



```
if(Input.GetKeyDown(KeyCode.Space))
{
    number += 1;
    if(number % 2 == 0)
    {
        Debug.Log(number.ToString() + " is even.");
    }
    else
    {
        Debug.Log(number.ToString() + " is odd.");
    }
}
```

We first check if the spacebar is being held down.
If so, we add one to our number.
We then check to see if the number is divisible by two by seeing if it's remainer is equal to zero.
If it is we output the number is even. Else, we output the number is odd.

4. Save the script and run the game. If we hit spacebar we should see the following messages:



```
[12:04:42] 11 is odd.
UnityEngine.Debug:Log (object)
[12:04:43] 12 is even.
UnityEngine.Debug:Log (object)
[12:04:47] 13 is odd.
UnityEngine.Debug:Log (object)
[12:04:47] 14 is even.
UnityEngine.Debug:Log (object)
```

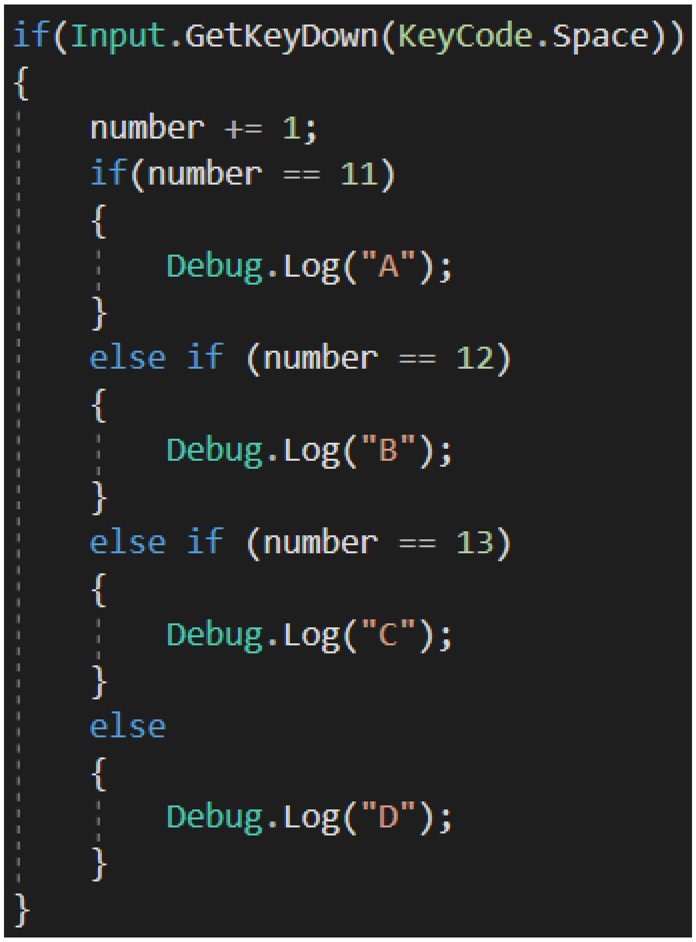
Else If Statements

1. To wrap on If Statements we also have **Else If Statements**. These act as a sort of middle ground between If and Else statements. Let's look at the pseudocode below to understand how to use them.

```
If (my_color == Red)
{
        Debug.Log("Red.")
}
else if(my_color == White)
{
        Debug.Log("White.")
}
else if(my_color == Blue)
{
        Debug.Log("Blue.")
}
else
{
        Debug.Log("You have some other color.")
}
```

We first have an If Statement check to see if the variable my_color is red. If so, we output the text "Red."
If not, we have an Else If Statement that checks if my_color is white, then if so, we output the text "White."
We repeat this for the color blue.
Then finally we get to the Else Statement. This will only run, if my_color was not red, white or blue. So, the message "You have some other color." will be output to the console.

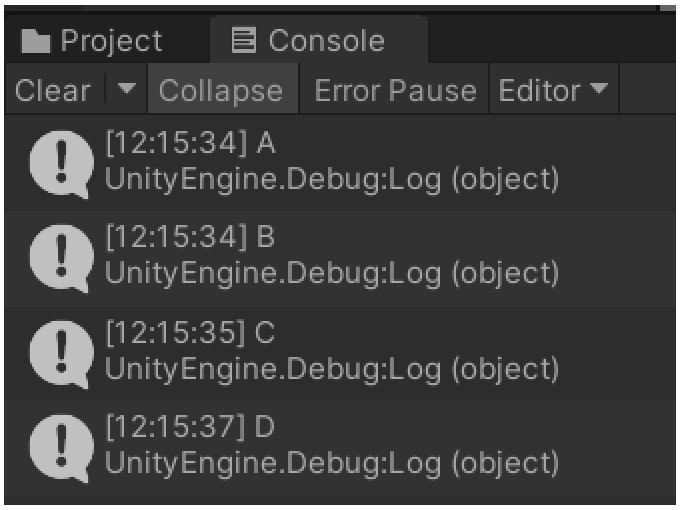
2. Let’s test adding Else If Statements by changing the code in our update function to:



```
if(Input.GetKeyDown(KeyCode.Space))
{
    number += 1;
    if(number == 11)
    {
        Debug.Log("A");
    }
    else if (number == 12)
    {
        Debug.Log("B");
    }
    else if (number == 13)
    {
        Debug.Log("C");
    }
    else
    {
        Debug.Log("D");
    }
}
```

Like before we check if the space was was pressed and increase the number variable by one.
We then check to see if number is at 11, and if so we output the message “A”. It is important to note that the code following the if, statement will not be read.
If we hit the space bar once more, number will be 12, so the message “B” output. The same can be said about the number reaching 13 and output “C” as a message.
Once number reaches 14, none of the if or else if statements will trigger leaving only the else to output the message “D”.

3. If we save the script and run the game we will see the messages:



```
[12:15:34] A
UnityEngine.Debug:Log (object)
[12:15:34] B
UnityEngine.Debug:Log (object)
[12:15:35] C
UnityEngine.Debug:Log (object)
[12:15:37] D
UnityEngine.Debug:Log (object)
```

Switch Statements

1. Another way of controlling the logic of our code is to use **Switch Statements**. Switch Statements are supplied with a value and then check what is known as a **Case**, this is a potential value that the supplied value could be.

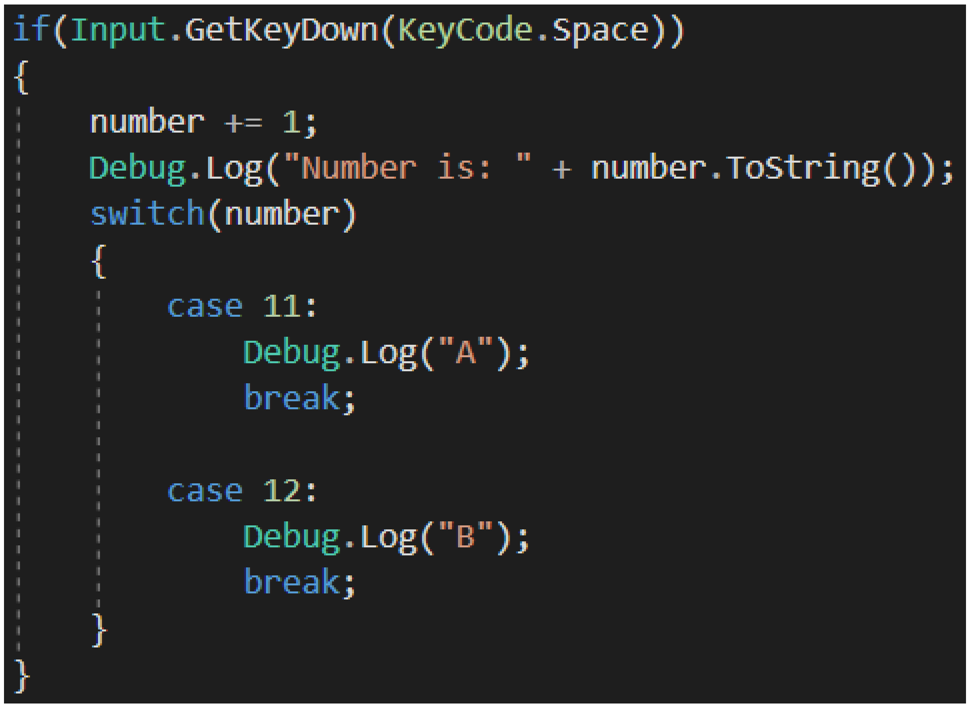
2. Let's look at some pseudocode to see how a switch statement might work.

```
switch (my_fruit)
{

    case Apple:
        Output the message Apple.
        break;

    case Orange:
        Output the message Orange.
        break;

}
```

We write the word switch and then supply it with a value in parenthesis, this is the my_fruit variable. The switch statement has two potential cases, Apple and Orange. If my_fruit is Apple it will print out the message "Apple". But If my_fruit is Orange it will print out the message "Orange".

The word **break** was also introduced. All this does is stop any code from under the word break from running. This allows us to essentially "break out" of our switch statement.

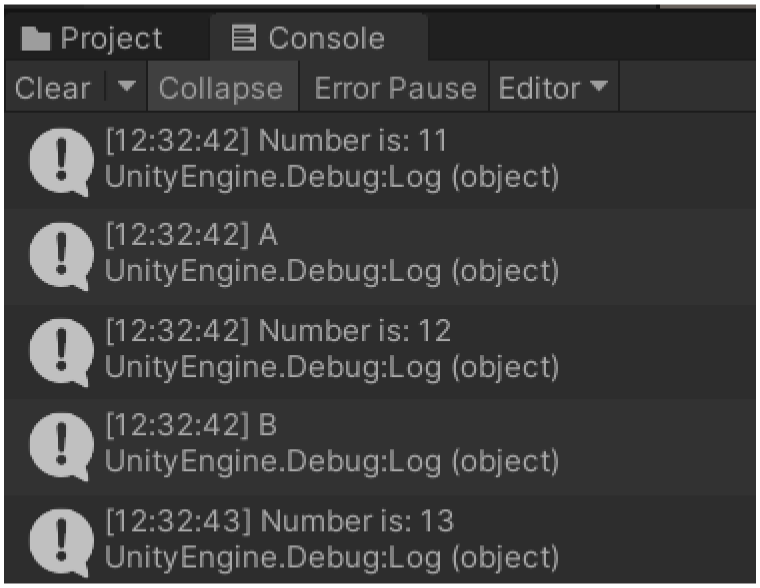
3. Let's head back to the script and change our **Update()** function to have the following code:



```
if(Input.GetKeyDown(KeyCode.Space))
{
    number += 1;
    Debug.Log("Number is: " + number.ToString());
    switch(number)
    {
        case 11:
            Debug.Log("A");
            break;

        case 12:
            Debug.Log("B");
            break;
    }
}
```

Our switch statement uses the number variable as the supplied value. It then has a case for 11 and 12. Like our pseudocode it will output a message based on the value of the case.

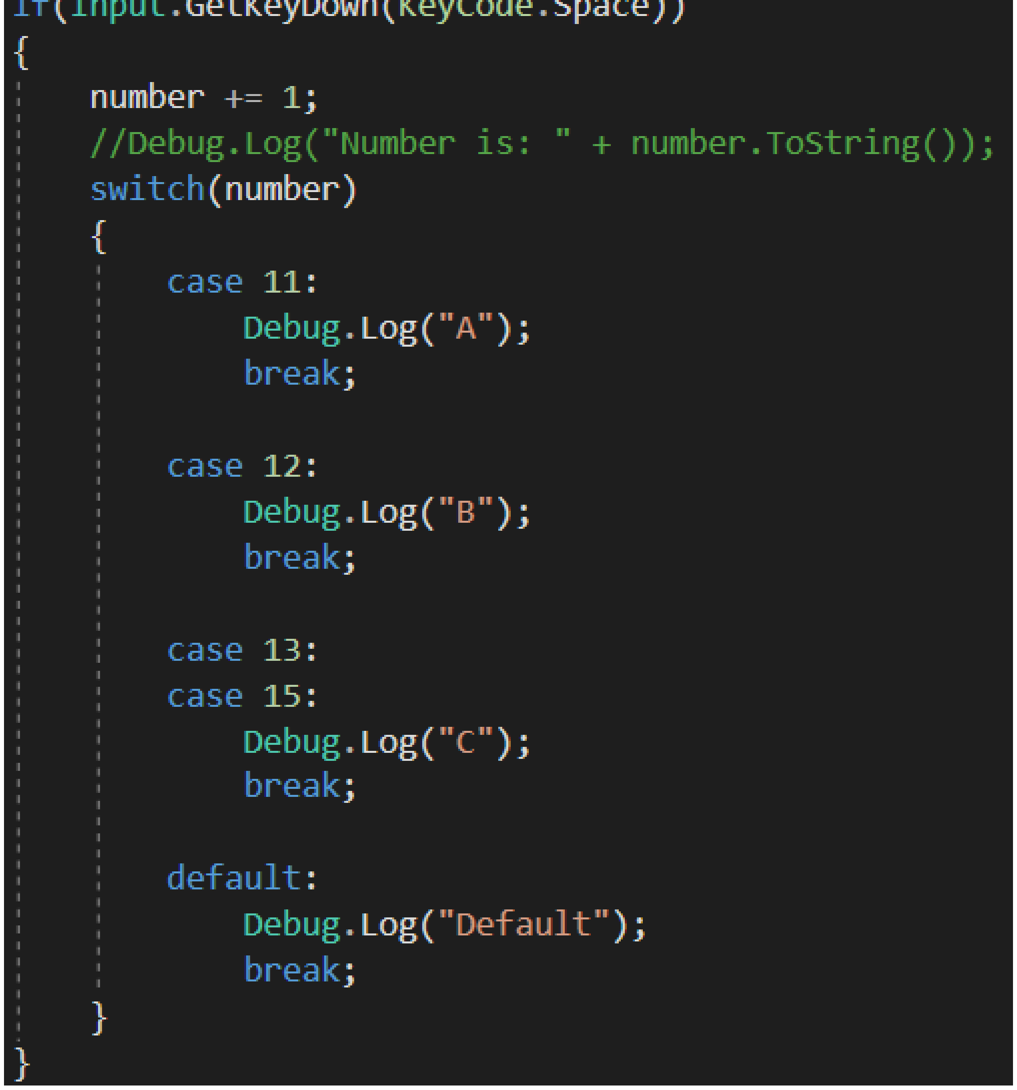
4. Save the script and run our game. We should see the following results when we hit the space bar.



```
[12:32:42] Number is: 11
UnityEngine.Debug:Log (object)
[12:32:42] A
UnityEngine.Debug:Log (object)
[12:32:42] Number is: 12
UnityEngine.Debug:Log (object)
[12:32:42] B
UnityEngine.Debug:Log (object)
[12:32:43] Number is: 13
UnityEngine.Debug:Log (object)
```

When the number variable reaches 11, the string "A" is output. When the number variable reaches 12, the string "B" is output. But if number reaches 13 no string is output because no case was met.

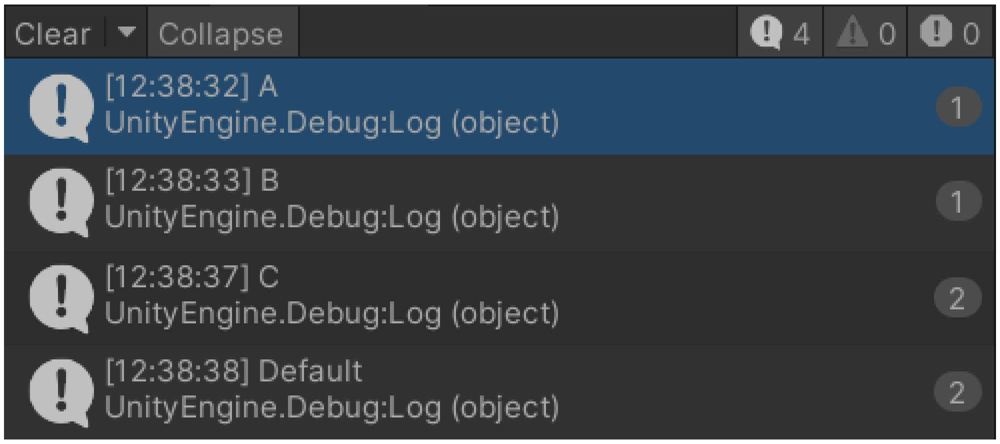
5. Let’s go back to our code and make the changes:



```
if(Input.GetKeyDown(KeyCode.Space))
{
    number += 1;
    //Debug.Log("Number is: " + number.ToString());
    switch(number)
    {
        case 11:
            Debug.Log("A");
            break;

        case 12:
            Debug.Log("B");
            break;

        case 13:
        case 15:
            Debug.Log("C");
            break;

        default:
            Debug.Log("Default");
            break;
    }
}
```

We add two new cases, case 13 and 15. We also add what is known as the **Default** case. This case will trigger when the supplied value does not match any of the provided cases.

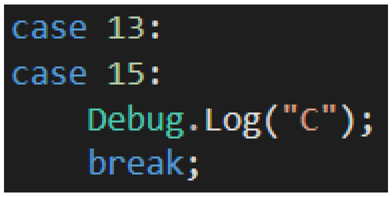
6. Let's save the script and run the game. We should see the results:



```
Clear | ▾ | Collapse                                   4   0   0
[12:38:32] A
UnityEngine.Debug:Log (object)                                  1
[12:38:33] B
UnityEngine.Debug:Log (object)                                  1
[12:38:37] C
UnityEngine.Debug:Log (object)                                  2
[12:38:38] Default
UnityEngine.Debug:Log (object)                                  2
```

Note: The code to output the value was commented out, so we will not see the message the displays the value of the number.

Like before when number is 11, the output is "A" nad when number is 12, the output is "B". When number reaches 13 and 15 the output is "C". (We can tell this because the message was output twice.)



```
case 13:
case 15:
    Debug.Log("C");
    break;
```

This happens because case 13 does not have the keyword break within it. Without the break the code continues to run until it finds another break keyword. This is what is known as **Fall Through**.

Then if number is not eqaul to any of the cases the default statement will run, ouputting the message "Default".

If vs Switch Statement

Now that we have covered both if and switch statements you might be asking yourself "Hey both of these seem pretty similar to each other, when would I use one over the other?"

And that is honestly a really good question to ask. I didn't have a good answer to this until recently.

We want to try and use Switch Statements if we know what the exact value of a variable will be. We are eliminating any guess work with these types of statements.

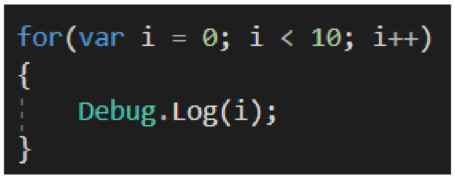
So on the other hand If Statements allow for that guess work. If we do not know what a particular value will be an if statement would be the way to go.

Looping

The last form of logic control we will be looking at are **Loops**. Loops are a good way for a chunk to run multiple times. We supply a loop with a condition and then once that condition is met the loop stops.

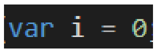
For Loops

1. In the **Start()** function of our MyScript script let's write the code:



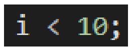
```
for(var i = 0; i < 10; i++)
{
    Debug.Log(i);
}
```

For Loops work by first stating a set of parameters the loop needs to follow. First we state the value that will be used to increment through the loop.



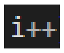
```
var i = 0
```

We then state the condtion for breaking our of the loop. For the loop we wrote, the loop will only run if i less than 10. Once i is greater or eqaul to 10 the loop will stop.



```
i < 10;
```

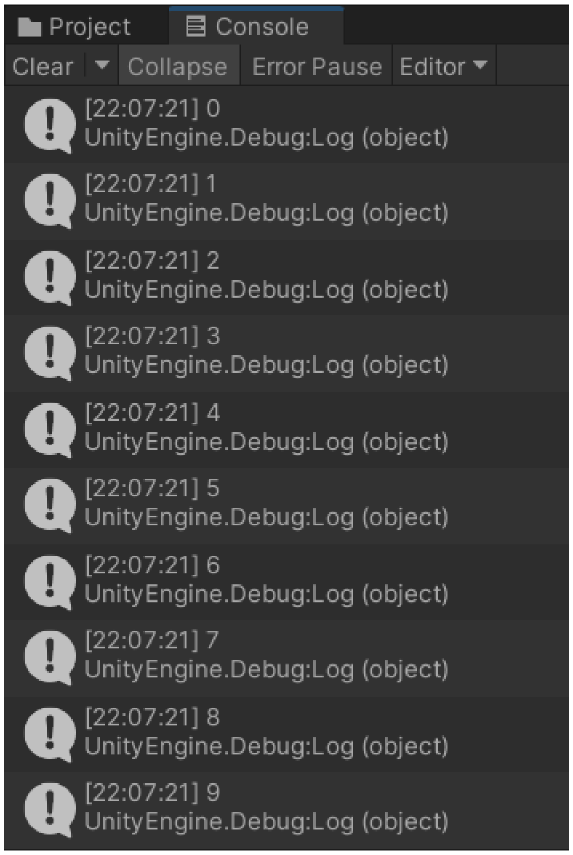
We then tell the loop how the value will be incremented at the end of each loop. In out case we are just adding 1 to the value of i.



```
i++
```

Then in the body we can run our code for the loop, in our case we are just going to output the value of i.

2. If we save the script and run the game we will see the output:

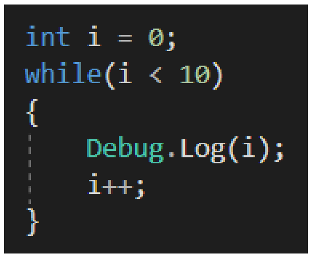


```
[22:07:21] 0
UnityEngine.Debug:Log (object)
[22:07:21] 1
UnityEngine.Debug:Log (object)
[22:07:21] 2
UnityEngine.Debug:Log (object)
[22:07:21] 3
UnityEngine.Debug:Log (object)
[22:07:21] 4
UnityEngine.Debug:Log (object)
[22:07:21] 5
UnityEngine.Debug:Log (object)
[22:07:21] 6
UnityEngine.Debug:Log (object)
[22:07:21] 7
UnityEngine.Debug:Log (object)
[22:07:21] 8
UnityEngine.Debug:Log (object)
[22:07:21] 9
UnityEngine.Debug:Log (object)
```

Notice how the output stops at 9. The value of i reached 10, breaking out of the loop.

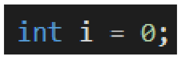
While Loops

1. In the **Start()** function, lets delete the code for our For Loop and write the code:



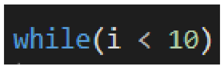
```
int i = 0;
while(i < 10)
{
    Debug.Log(i);
    i++;
}
```

This should look very similar to the For Loop we just removed. Like before we state the value that will be used to increment through the loop.



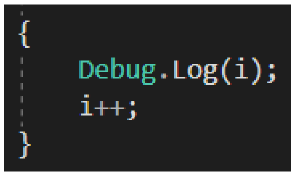
```
int i = 0;
```

Instead of writing the word for, we write the word **While** and then state the condtion for breaking our of the loop.



```
while(i < 10)
```

We then write the body of out loop.

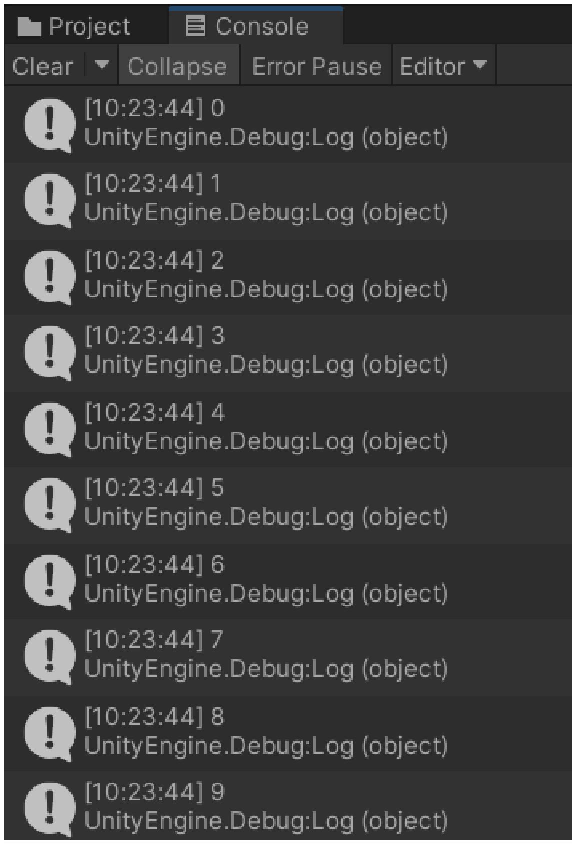


```
{
    Debug.Log(i);
    i++;
}
```

Again, we are just printing out the value of i. But we also write the code to increment i so that the condition for ending the loop can be reached.

Without the code to satisy the loop's condition, the loop would continue to run forever. Our game would halt at this point in the code since it would never break out of the loop.

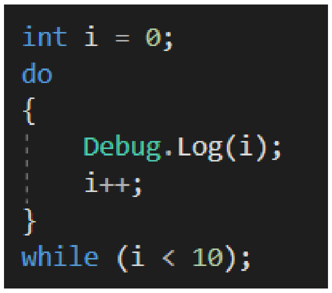
2. If we save the script and run our game, we should see the same output to the For Loop we wrote.



```
[10:23:44] 0
UnityEngine.Debug:Log (object)
[10:23:44] 1
UnityEngine.Debug:Log (object)
[10:23:44] 2
UnityEngine.Debug:Log (object)
[10:23:44] 3
UnityEngine.Debug:Log (object)
[10:23:44] 4
UnityEngine.Debug:Log (object)
[10:23:44] 5
UnityEngine.Debug:Log (object)
[10:23:44] 6
UnityEngine.Debug:Log (object)
[10:23:44] 7
UnityEngine.Debug:Log (object)
[10:23:44] 8
UnityEngine.Debug:Log (object)
[10:23:44] 9
UnityEngine.Debug:Log (object)
```

Do While Loops

1. The last type of loop that we will look at is a Do While Loop. Let's take the While Loop we wrote and change it to match the code below.

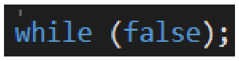
   ```
   int i = 0;
   do
   {
       Debug.Log(i);
       i++;
   }
   while (i < 10);
   ```

   The code should be pretty similar to what we had before, but now the order of events in our code is different. The main differentce being the condtion is **after** the body.

2. Saving the script and running the game we will get the same output as the While Loop we had.

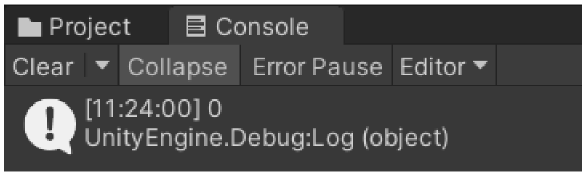
3. What this type of loop allows us to do is **ensure the body of the code runs atleast once**. Even if the condition for the loop is initially false.

   To show this off lets change the condition to be:

   ```
   while (false);
   ```

4. Now if we save the script and run the code we should see:

   ```
   [11:24:00] 0
   UnityEngine.Debug:Log (object)
   ```

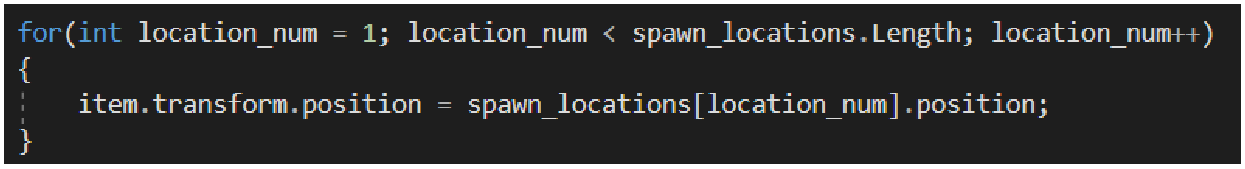
   We only get the message outputting 0 since the body of the loop only ran once.

For vs While Loops

You may have a couple questions about For and While Loops since both seem to perform the same task.

For Loops are useful for when you know how many times you want to loop through a piece of code. This could particularly be useful for going through a set of data.

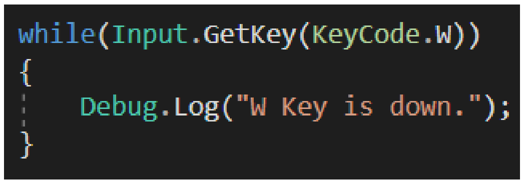
For instance, a player's inventory might have a set number of inventory slots. We can use a For Loop to go through each slot and output what the player has. We could also use For Loops for spawning objects at a set number of locations.



```
for(int location_num = 1; location_num < spawn_locations.Length; location_num++)
{
    item.transform.position = spawn_locations[location_num].position;
}
```

One the other hand, while loops allow for more flexibility. We aren't only limited to looping through for a set number of times. We can also write conditional statements that will return true/false value.

For instance, we can check to see if the player is holding a key down. While that key is held down the loop will run. Or maybe while the player has a health value above zero, we can have the player regenerate health. Again, we can be more flexible with our conditions for this type of loop.



```
while(Input.GetKey(KeyCode.W))
{
    Debug.Log("W Key is down.");
}
```

GitHub

1. Push the project to the GitHub repository.