Dice Game

Here we will be talking about boolean algebra and how we can use logic in our code to create smarter code.

Boolean Logic

We've already talked about how booleans can be a true or a false value but now we will use them to decide what to do with our code.

Boolean Operators

In the previous lesson we mentioned that booleans can be set using the "Greater Than" `>` and "Less Than" `<` signs. These are what we call "operators".
Operators do things to our values. For instance the "Addition Operator" `+` and similar are also operators, just not for booleans.
Here are all commonly used operators.

Equal To

This might seem a little weird at first but if we want to seem if two things are equal we use "Equality Operator" `==` .
The reason being that a single `=` , the "Assignment Operator" assigns a value to a variable.
If we tried to use this when comparing values the computer would get confused.
`=` sets the value of a variable whereas `==` compares two values.

These can be used with most variable types.

```
// This is rather a pointless statement but it works.
// myBool is true.
bool myBool = true == true;

// The equality operator is not exclusive to booleans.
// myBool is false.
myBool = 30 == 23;

// While this will not error, it will give you results you might not expect.
myBool = 40 = 40;
```

Greater And Lesser Than

As stated above we've talked about `<` and `>` being used to see which side of the operator is larger.
A very common situation is needing to know if a value is larger **or** equal to another value.
The single signs only count if the value is larger or lesser. If the two are equal they return false.
To compare greater / lesser or equal to we can use two shorthand operators. `<=` and `>=` .

These are most useful for number values.

```
// myBool is false.
bool myBool = 3 < 3;

// myBool is true.
myBool = 3 <= 3;
```

NOT Operator

Now we start into some new territory. The "NOT Operator" is specifically for booleans. We use the `!` exclamation mark to denote this operator.
This simply switches our boolean. `true` becomes `false` and `false` becomes `true` .

We write it like this.

```
// myBool is false.
bool myBool = !true;

// myBool is true.
myBool = !myBool;

// myBool is still true.
myBool = !!myBool;
```

Here we will show you a new tool to see how these work. Truth tables.
Now that our operator works only with bools we can easily lay out all possibilities. Here represented using `1` for `true` and `0` for `false`.

`B = !A`

A	B
0	1
1	0

This is a simple one but shows off the use of truth tables well.

AND Operator

The "AND Operator" checks to see if both of two booleans are true. If either or both are false the operator returns false.
We use the `&&` double ampersand with a boolean on each side to denote this operator. Just like this :

```
// myBool is false.
bool myBool = true && false;
```

And for all possibilities here's the truth table.

`C = A && B`

A	B	C
0	0	0
1	0	0
0	1	0

A	B	C
1	1	1

OR Operator

The "OR Operator" checks if either of two booleans are true. We use `||` double pipe separators to denote this function.
Do note, if both booleans are true the operator will still return true.

```
// myBool is true.
bool myBool = true || false;
```

`C = A || B`

A	B	C
0	0	0
1	0	1
0	1	1
1	1	1

There are other options for operators but these are the main ones we use.

Boolean Order Of Operations

Remember that boolean statments still use an order of operations when calculating.
The order is as follows :

1. Parentheses `()`
2. NOT operator `!`
3. AND operator `&&`
4. OR operator `||`

If there are two or more of the same on a single line it will be evaluated left to right.

```
// myBool is false.
// !true is evaluated first followed by the ||.
bool myBool = !true || false;
```

```
// myBool is true.
// The || is evaluated first followed by the !.
myBool = !(true || false);

// myBool is true.
// The && is evaluated first followed by the ||.
myBool = true && false || true;
```

When you are in doubt, just use parentheses.

Control Structures

Now that we can create more complex boolean statements we need a way to use them. This is where "Control Structures" come in.

If Statements

The "if statement" is used to determine which block of code should be run based on a boolean value. Let's hop right into an example.

```
bool myBool = true;

// The below code block will run as myBool was declared as true.
if(myBool) {
        // Code within these curly braces will be executed if myBool IS true.
        Debug.Log("Hi");
        myBool = false;
}

// The below code block will NOT run as myBool was set to false in the first block.
if(myBool) {
        Debug.Log("Bye");
}
```

The output of that program is :

```
Hi
```

`Bye` is not printed because in the first statement we set the value of `myBool` to false.
The structure of an `if` has three parts :

1. The keyword `if` denotes that this is an `if` statement. Pretty simple.

2. Next you have parenthesis `()`. These contain the boolean value, or boolean expression, that determines if the statement body should run.
3. Then you have curly braces `{}` with the code to run in between them. Indent the code in here to keep it pretty.

To determine whether the code executes you can use any boolean expression.
Above we directly use a bool to say `true` or `false`.
Similarly we can use expressions (like `if(true || false)`, `if(4 >= 2)`, and others shown above).

Else Statements

The "else statement" is basically just a second `if`.
The code within an `else` statement is executed if the `if` does not execute. So if the `if` is `false`.

```
bool myBool = true;

// The below code block will run the if statement.
if(myBool) {
        // Code within these curly braces will be executed if myBool IS true.
        Debug.Log("Hi");
        myBool = false;
} else {
        // Code within these curly braces will be executed if myBool IS false.
        Debug.Log("Bye");
}

// The below code block will run the else statement.
if(myBool) {
        Debug.Log("If");
} else {
        Debug.Log("Else");
}
```

The output of that program is :

```
Hi
Else
```

`Bye` is not printed because in the first statement the `if` is executed.
In the second the `else` is executed because the `if` is false.
Think of it like :

> If a condition is met, do this. Else do this.

Almost seems like they thought about the name.

Else If Statements

Now the last thing we need to talk about for `if` statements is that we chain them together using an "else if statement".
We do this like so :

```
bool myBool = false;
bool mySecondBool = true;

if(myBool) {
        // This code will be run if the "if" is evaluates to true.
        Debug.Log("Hi");
} else if(mySecondBool) {
        // This code will be run if the first "if" is false and this one evaluates t
        Debug.Log("Bye");
} else {
        // This code will run if the other statements fail.
        Debug.Log("See ya later");
}
```

We are effectively chaining the `if` 's together. If the first one fails we check against the next one, then the next, until finally if all else fails, our `else` runs.
It is important to note that only **ONE** block of code can run like this. If two `if` statements are both true, only the first one in the list will be run. If you need more than one `if` statement to be checked and run do not chain them with `else` statements.
Go and make your own series of `if` statements.

Scope

The last part of this lesson we are talking about "Scope". Scope is the idea of where our program can "see" different variables.
Let's do a quick example.

```
class Test {

        int var1;

        function myFunc() {
                int var2;

                if(true) {

                        int var3;

                }


        }

}
```

Here we see three "levels" or scopes to our code. Inside our class, inside our function, and inside the if statement. These scopes can be seen usually with the curly braces `{}` .
Let's change this program up to see how these scopes affect our variables.

```
class Test {

        int var1;

        function myFunc() {

                var1 = 3;

                // This works. Now var2 == 3.
                int var2 = var1;

                if(true) {

                        // This works. Now var3 == 3;
                        int var3 = var1;

                }

                // This errors. var3 is not declared within this scope.
                var2 = var3;

        }

}
```

As shown by the comments above some of these assignments work and others don't.
Since `myFunc` is within the scope (the curly braces `{}`) of the class `Test` we have access to the var `var1`.
Within the `if` statement we can also access `var1`. Again this is because the `if` is within the scope of the class `Test`. Though you might say the `if` is within `myFunc` which is within `Test`. Either work.
Then after our `if` statement we can see that trying to access `var3` and it errors. Since `var3` exists within the scope of the `if` we cannot access it from the scope of `myFunc`.
This is because scopes only work one way, you can access something in a higher scope but not a lower scope.

A comparison often used is to think of scopes as a series of doors.
You start in one room (the class) with some variables. You can use these variables however you'd like. Then you go into the next room (a function) and you leave the door open behind you.
In this new room there are new variables that you can use. Since you left the door open you can still look back to the previous room and use those variables.
Once you are done in the function you leave the room and close the door meaning you can't see those variables (in the function) anymore.
Hopefully that makes some sense.

// GRAPHIC NEEDED

Note A side part of scopes would be this :

```
// ERROR, var1 is not defined.
var1 = 3;

// var1 is defined here.
int var1;
```

Since our script runs top to bottom (ish) we cannot use a variable unless it's been declared already, Just like above.

On Your Own

Alright that was a lot of content to get through. We've got a little more to come still. To prepare for that try out some activities on your own to familiarize yourself with these concepts.

See if you can combine boolean operators to create these truth tables (where C is the output) :

A	B	C
0	0	0
1	0	1
0	1	1
1	1	0

A	B	C
0	0	1
1	0	1
0	1	1
1	1	0

A	B	C
0	0	1
1	0	0
0	1	0
1	1	0