Lesson Summary

Key Points

- At a low level, most functionality involves using **Operators** to compare, check, or change data.
- Operators are used with **Operands**, which is the data that is typed to the left and right of the operator (the left and right hand operands).
- Common operators include the dot operator (.), the assignment operator (=), arithmetic operators (+ - / *), and boolean and equality operators (! == !=).
- The null conditional access operator (?.) works with delegates but should not be used with Unity objects (use the equality or inequality operators instead.
- The **Ternary Operator** is sometimes useful as a quick and easy if/else statement.
- Logical operator comparisons (&, |, ^) always check both operands, Conditional logical operator comparisons (&&, ||) only check the right hand operand if they actually need to.
- This means that you're unlikely to ever use the **logical and** (&) or **logical or** (|), unless you specifically still want both operands to be executed, such as if the operands are functions that return booleans.

Transcription & Code Examples

While variables store data, functions are used to create logic.

But how does that work? How can you create the logic inside of a function.

While you'll typically execute behaviour inside of a function by calling another function, logic can be created by assigning, comparing and evaluating variables.

Typically, this is done using operators.

In very simple terms, an operator is a symbol that allows you to assign, compare or otherwise manipulate data to create logic.

For example, you've probably already used the assignment operator, which uses an equals symbol, and sets the value of the left hand operand to the value of the right.

= Assignment Operator

1. public float health = 100; // Assigns a literal value to the health variable

Like the assignment operator, most operators are simple, self-explantory and easy to use.

However, there are many that are less intuitive but that can be extremely useful once you know that they exist.

Let’s start with the basics.

Access Operators

. member access

1. player.health = 100; // Set's the player's health to 100

One of the most common operators you’re likely to use is the member access operator, or dot operator.

This provides access to the members, meaning the publicly available variables and methods, of the left hand operand.

If you’ve written any code in Unity before then, chances are, you’ve already used this operator, and you’ll typically use it to dig down into subdirectories of a reference to reach a particular variable or method.

[] index access

1. array[0] // gets the first element

The Index access operator, which uses square brackets, allows you to reference an element of a collection, where the number that you pass into the operator is the element that you want to use.

This allows you to access one element in an array or a list or, more often, allows you to use every item in a collection by passing in a dynamic reference, such as in a for loop, for example.

You’ll learn more about loops in the lesson Iterators however, for now, in this example, i is simply an integer that’s added to and reused to access each item in the collection in turn.

?. null conditional access

1. delegate?.Invoke(); // Invokes the delegate if it's not null

It’s possible to use a member access operator conditionally by using the null conditional access operator.

This takes the form of a question mark that precedes the access operator and checks to see if the left hand operand is null before trying execute the right.

It basically works as as short circuit where, if the operand may be null, a check can be performed before going any further.

You'll commonly see this used when invoking delegates, as it's a simpler way to check that the delegate is set to something before it's called.

It's basically a short hand null check, which is easier to type and simpler to read.

But, if that's true, why is it only commonly used with delegates? Why not all kinds of nullable references?

The null conditional operator doesn't work correctly with object references.

Meaning game objects and components.

This is a quirk of Unity where, in order to check if a game object reference is actually null, the underlying C++ engine needs to be checked.

The null conditional operator bypasses this check, meaning that it doesn't work, causing an error if the reference is null.

Generally speaking, all this means is that you shouldn't use the null conditional operator to check for object or object related references.

Instead, use the Equality or Inequality operators, which have a custom implementation in Unity, meaning that they work how you'd expect them to.

Comparison Operators

== Equality

1. if(reference == this){} // Checks if the reference and this script are the same

!= Inequality

1. if(reference != null){} // Checks if the reference is null

The equality operator can be used to check if one operand matches another.

While the Inequality operator can be used to check if it doesn't.

Typically, you might use the these operators to check if a reference is or is not null, either by only executing a code block if the reference exists, or by returning out of a function early if it doesn't.

The Equality operator can also be used to check if two things are the same.

For example, comparing two variables, such as numbers, allows you to check if something is a specific value or not.

While comparing two references will let you check if two things are the same or not.

This works on an instance basis, where the operands can be of the same type but will only evaluate to true if they actually refer to the same thing.

This can be useful when creating a singleton, for example, where you'd typically check if a reference matches the specific script it's being checked from.

Or allowing an object to exclude itself when looping through a list.

< less than

```
if(player.health < 0)
{
   // The player is dead!
}
```

> greater than

```
if(player.health > 100)
{
   // The player has too much health!
   player.health = 100;
}
```

<= less than or equals

```
if(number <= 0){} // The number is zero or lower
```

>= greater than or equals

```
if(number >= 0){} // The number is zero or higher
```

The Less than, greater than, less than or equals and greater than or equals operators work pretty much exactly as you'd expect them to, and allow you to perform numerical checks that either include or exclude the target number.

For example, the less than operator is commonly used when looping through an array, where the last element inside of its range would normally be one less than the length value of the array itself.

Using less than, instead of less than or equals, avoids an out of range exception.

Arithmetic Operators

+ Addition

1. int two = 1 + 1;

- Subtraction

1. int five = 6 - 1;

+= Addition Assignment

1. int hp += 50; // Add 50 points of health
2.
3. transform.position += Vector3.forward * 5 * Time.deltaTime; // Moves an object forward at five units per second

-= Subtraction Assignment

1. int hp -= 50; // Remove 50 points of health

Arithmetic operators can be used to manipulate numbers.

For example, the addition and subtraction operators can be used to add or subtract numbers together, while the Addition Assignment and Subtraction Assignment operators can be used to add to or subtract from an existing value without overwriting it.

Essentially, this is the same as assigning a value to itself, plus or minus an additional amount, and you'd typically use this when adding to a value or taking away from, such as a health value, for example.

Typically, arithmetic operators are very straightforward, however there are a couple of use cases where they can be used with non-numerical data.

For example, string concatenation works by adding strings and values together using the addition operator.

While the addition assignment and subtraction assignment operators are commonly used to add or remove functions from a delegate.

++ Increment

1. counter++; // Adds one to a number called counter

-- Decrement

1. counter--; // Subtracts one from a number called counter

The Increment and Decrement operators increase or decrease a value by one.

Typically, you'd use these for loops, where incrementing a counter, for example, would allow you to loop through all of the elements in an array.

*** Multiply**

1. Vector3 forwardDistance = Vector3.forward * 10; // Creates a forward vector of 10 units

/ Divide

1. double beatTime = 60d / 80; // Creates a double beat duration at 80bpm

Multiplication can be used to multiply numbers, and can also be used to rotate a vector, where multiplying a vector by a quaternion will rotate it by that amount.

The division operator can be used to divide one number by another. This generally works how you'd expect it to, except that it's worth paying attention to the implicit type of the left hand operand. For example if you assign a double value to the result of a divided integer, you'll only get an integer back.

Adding a d suffix fixes this problem, as the whole number is treated as a double when it's divided.

% Remainder (modulo)

1. float three = 13 % 5; // Returns the remainder, after division, of one number by the other.

The remainder operator, or modulo, returns the remainder of the left hand operand after division by the right.

For example 5 modulo 4 returns 1, while 13 modulo 5 returns 3.

This happens because 5 goes into 13 twice, leaving 3 behind.

This can be useful for all kinds of mathematical calculations, especially when calculating tempo, or similar measurements of progress inside of a fixed unit.

Mathf class for other maths functions

While there only are a limited number of arithmetic operators, it's worth mentioning that Unity includes a large selection of mathematical functions via the **Math F class**.

While there's too much in this class to cover in the scope of this video, we'll be looking at some of the more common mathematical functions in the **Working with Numbers** lesson later in this course.

Boolean Operators

Boolean operators return a true or false value and are typically used in if statements to determine if a block of code should be executed or not.

! Logical negation

```
if(!reference)
{
   // Exits a function if the reference is null
   return;
}
```

The logical negation operator is used to return the opposite state of a value.

For example, when using a boolean value in an if condition, it's possible to check if the value is true by simply passing in the variable.

If the boolean is true, the code block executes.

Likewise, the logical negation operator can be used to check if something is the opposite of true, meaning false.

This is functionally the same as using the equality operator to see if a value is true or not, it's just simpler to write, and should generally be used when you want **false**.

It's also possible to use the logical negation operator to check if a reference is null. This works because it's possible to check if a reference exists by passing it into an if condition directly, where a valid reference returns true, without needing to use the equality operator to check it.

The logical negation operator can be used to check if a reference does not exist, by checking for the opposite of a reference.

& logical and operator

```
if(conditionOne & conditionTwo)
{
   // True if both are true, always checks both operands.
}
```

&& conditional logical and operator

```
if(conditionOne && conditionTwo)
{
   // True if both are true, checks the left operand first.
}
```

| Logical or

```
if(conditionOne | conditionTwo)
{
   // Checks if one is true, always checks both operands.
}
```

|| conditional logical or

```
if(conditionOne || conditionTwo)
{
   // Checks if one is true, only checks the right operand if the left is false.
}
```

^ Exclusive OR (XOR)

```
if(conditionOne ^ conditionTwo)
{
   // Checks that both are different, always checks both operands
}
```

Logical operators can be used to evaluate multiple conditions together and typically come in two variations, logical operators and conditional logical operators.

Standard logical operators, such as the **and**, **or**, and **exclusive or**, are denoted with a single symbol.

For example, **and** will evaluate to true if both operands are true, **or** will return true if one of the operands are true and **exclusive or** will return true if one operand is true while the other is false.

Logical operators always evaluate both operands, even when doing so would be unnecessary, while conditional operators, which are denoted with double ampersands and pipe symbols, for conditional and and conditional or, will only evaluate the right hand operand if the left hand operand is true, or in the case of the conditional or, false.

This means that, generally speaking, there are very few cases where you'd use the standard **logical and** since, if the first operand isn't true, there's really no reason to check the second.

However, while rare, there may still be times when you'd want to do this. For example, it might be that the operands are functions that ultimately return true or false, but still execute other code. In which case you may want both operands to still be executed, in a logical and, for example, even if the condition ends up returning false.

Ternary Operator

1. int num = condition ? 5 : 2; // Sets the number to 5 if the condition is true, 2 if it's false.

Generally, many of the available operators can be used in place of each other. The only difference is that some operators are simpler to write and take up less space, which can be extremely useful in a large script.

One operator that's extremely good at this is the **Ternary Conditional Operator**.

The Ternary Operator is basically shorthand for assigning one of two values depending on a condition.

It works by assigning a condition to the variable you want to set, followed by a question mark, followed by two possible values that can be assigned to the variable separated by a colon.

If the condition is true, the first value is assigned, if it's not, the second is instead.

This is functionally the same as checking the condition in an if statement, and assigning one of the two values based on the result, except that it takes up much less space and, when you're used to using it, can be much easier to read.

Bitwise and Shift Operators

<< Left shift / >> Right Shift

1. int layerMask = 1<<9; // Shifts the bit value for 1, 9 bits to the left. Passing this value in as a layer mask will select the 9th layer.

~ Bitwise complement

1. int layerMask = ~(1<<9); // Shifts the bit value for 1, 9 bits to the left and flips all of the bits (i.e. 0 becomes 1). Passing this value in as a layer mask will select everything except the 9th layer.

When you're first getting started in Unity, you're less likely to use the bitwise and shift operators, but they can be helpful in a couple of specific scenarios.

When dealing with layer masks, for example, which use the binary representation of integer values to flag specific layers, it's possible to use left shift operator to identify a specific layer in code, without using a layermask variable.

This works by shifting the bits left by a certain number which, when performed on the number one, moves the active flagged bit to the layer you want.

While the Bitwise complement operator can be used to flip all of the bits which, in a layermask, allows you to exclude a particular layer instead.