Operators

What we will learn:

8 Types of Operator(s):

1. **Mathematical**
2. **Unary**
3. **Relational**
4. **Logical**
5. **Assignment**
6. **Ternary**
7. **Null Coalescing**
8. **Null Conditional**

Var keyword

Var keyword

```
int x = 10;
```

Var keyword

```
var x = 10;
```

Var keyword

What's the difference?

```
int x = 10;

var x = 10;
```

Var keyword

They are the same thing!

```
int x = 10;

var x = 10;
```

Var keyword

```
int x = 10; //explicit typing

var x = 10;  //inferred typing
```

C# is Statically Typed

- Every variable must have a type

Var keyword

```
int x = 10; //explicit typing

var x = 10;  //inferred typing
```

Var keyword

```
var x = 10;
```

Var keyword

```
var x = 10;
```

var is a “placeholder” for int

Var keyword

var x = “hello”;

What is the the type of x?

Var keyword

var x = “hello”;

String!

Var keyword

```
var x = 4.5;
```

What is the the type of x?

Var keyword

```
var x = 4.5;
```

Double!

Definitions

Operand - the data that is being operated on

Operator - perform an operation on operand(s)

Operands

- **Operand** - the data that is being operated on

$$x + y = \ ?$$

Operands

- **Operand** - the data that is being operated on

x and y are the operands

$$x + y = ?$$

Operator

- **Operator** - perform an operation on operand(s)

$$x + y = ?$$

Operator

- **Operator** - perform an operation on operand(s)

+ is the operator

x + y = ?

Categories

1. Binary
2. Unary
3. Ternary

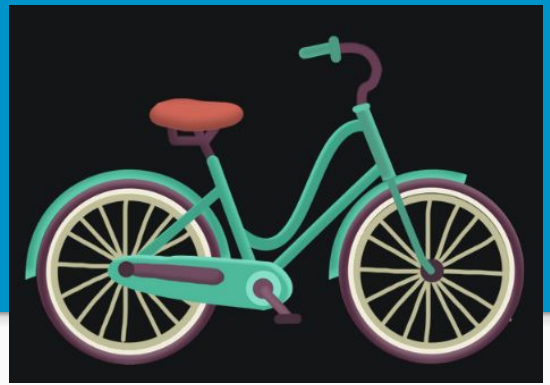
Categories

1. Binary - requires at least two operands
2. Unary
3. Ternary

Categories

1. Binary - requires at least two operands
2. Unary - only needs one operand
3. Ternary

Categories



1. Binary - requires at least two operands
2. Unary - only needs one operand
3. Ternary - requires three operands

1. Mathematical

These are used to perform mathematical operations on operands.

- Addition: **x + y**
- Subtraction: **x - y**
- Multiplication: **x * y**
- Division: **x / y**
- Modulus: **x % y**

1. Mathematical

These are used to perform mathematical operations on operands.

- Addition: **x + y**
- Subtraction: **x - y**
- Multiplication: **x * y**
- Division: **x / y**
- Modulus: **x % y**

Binary Operators

1. Mathematical

These are used to perform arithmetic/mathematical operations on operands.

- Addition: **x + y**
- Subtraction: **x - y**
- Multiplication: **x * y**
- Division: **x / y**
- Modulus: **x % y**

Mathematical - *Modulus*

Modulus means finding the remainder!

Mathematical - *Modulus*

100 % 3

Mathematical - *Modulus*

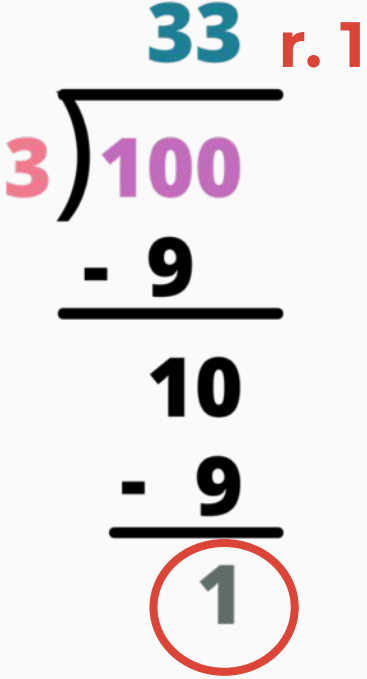
100 % 3

Means: what is the remainder of 100/3?

Mathematical - *Modulus*

$$
\begin{array}{r}
33 \\
3\overline{)100} \\
-\ 9 \\
\hline
10 \\
-\ 9 \\
\hline
1
\end{array}
$$

Mathematical - *Modulus*



Mathematical - *Modulus*

$$100 \ \% \ 3 = 1$$

2. Unary

Two kinds:

- Increment: The **++ operator** is used to increment the value of an integer.
- Decrement: The **-- operator** is used to decrement the value of an integer.

Unary

- **It only requires <u>one</u> operand!**

x++ → **increments by 1**

x-- → **decrements by 1**

Unary

- **It only requires <u>one</u> operand!**

x++ → **increments by 1**

x is the operand

x-- → **decrements by 1**

Unary

Placement matters!

x++ → **increments by 1**

++x → **also increments by 1**

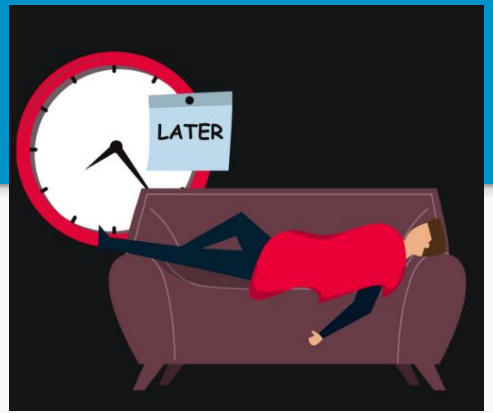
Unary

Both increment by 1 but at at different times

x++ → **post-increment**

++x → **pre-increment**

Unary



Both increment by 1 but at at different times

x++ → **procrastinator**

++x → **get it done right away**

3. Relational

Relational operators are used to compare two values.

- **> (Greater Than) operator**
- **< (Less Than) operator**
- **>= (Greater Than Equal To) operator**
- **<= (Less Than Equal To) operator**
- **== (Equal To) operator**
- **!= (Not Equal To) operator**

3. Relational

Relational operators are used to compare two values.

- **> (Greater Than) operator**
- **< (Less Than) operator**
- **>= (Greater Than Equal To) operator**
- **<= (Less Than Equal To) operator**
- **== (Equal To) operator**
- **!= (Not Equal To) operator**

3. Relational

Each operation will return a True or False value

- **> (Greater Than) operator**
- **< (Less Than) operator**
- **>= (Greater Than Equal To) operator**
- **<= (Less Than Equal To) operator**
- **== (Equal To) operator**
- **!= (Not Equal To) operator**

Relational

Each operation will return a True or False value

- **10 > 5**
- **2 < 3**
- **5 >= 5**
- **7 <= 2**
- **10 == 10**
- **7 != 2**

Relational

Each operation will return a True or False value

- **10 > 5 → true**
- **2 < 3 → true**
- **5 >= 5 → true**
- **7 <= 2 → false**
- **10 == 10 → true**
- **7 != 2 → true**

Relational

The numbers could be represented by variables!

- **number1 > number2**
- **numberGuessed == actualNumber**

4. Logical

Used to evaluate boolean expressions

- **&&** **The AND operator**
- **||** **The OR operator**
- **!** **The NOT operator**

Logical

Both sides must be true for entire evaluation to be true

- **5 >= 5 && 4 == 4** **The AND operator**

Logical

Both sides must be true for entire evaluation to be true

true true

- **5 >= 5 && 4 == 4**

The AND operator

Logical

Only one side has to be true for entire evaluation to be true

- **5 >= 5 || 4 != 4** **The OR operator**

Logical

Only one side has to be true for entire evaluation to be true

true false

- **5 >= 5 || 4 != 4**

The OR operator

Logical

Reverses the boolean value of its operand

AKA if the operand has a value of true, the Logical NOT operator will change it to false, and vice versa.

- **!** **The NOT operator**

Logical

Reverses evaluation results!

- **!** **The NOT operator**

Logical

Reverses evaluation results!

```
true
```

```
false
```

Logical

Reverses evaluation results!

```
bool isTrue= true;

isFalse = !isTrue;      // will set isFalse to false
```

Logical

The Logical NOT operator is often used in loops or conditionals to check for the opposite of a Boolean condition.

Logical

scope will execute

```
bool isBlue = false;

if (!isBlue)

{

        Console.WriteLine("The sky is NOT blue");

}
```

Assignment

- Used to assign a value to a variable.

5. Assignment

1) **Simple**
2) **Add**
3) **Subtract**

etc...

Assignment - *Simple*

- = (Simple Assignment)

```
a = 10;

b = 20;

ch = 'y';
```

Assignment - *Add*

- += (Add Assignment)

```
x += 4;
```

Assignment - *Add*

- += (Add Assignment)

```
int x = 10;

x += 4;
```

Assignment - *Add*

- **+= (Add Assignment)**

```
int x = 10;

x += 4;
```

Value of x is 14

Assignment - *Add*

```
x += 4;
```

Is shorthand for:

```
x = x + 4
```

Assignment - *Subtract*

- **-= (Subtract Assignment)**

```
x -= 4;
```

Assignment - *Subtract*

- **-= (Subtract Assignment)**

```
int x = 10;

x -= 4;
```

Assignment - *Subtract*

- **-= (Subtract Assignment)**

```
int x = 10;

x -= 4;
```

Value of x is 6

Assignment - *Subtract*

- **-= (Subtract Assignment)**

```
x -= 4;
```

```
x = x - 4
```

Assignment - *Subtract*

```
x -= 4;
```

Is shorthand for:

```
x = x - 4
```

6. Ternary

- Three operands

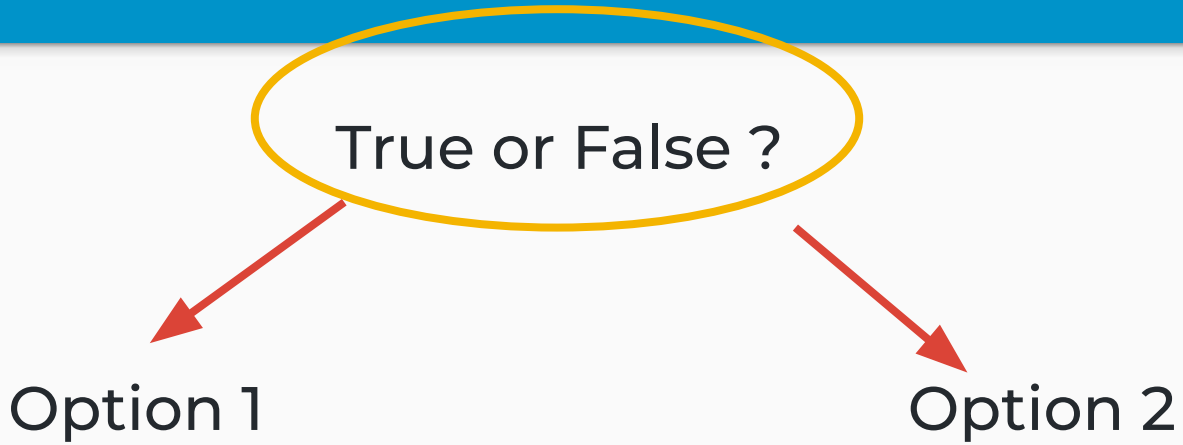
6. Ternary

- Shorthand version of an if-else statement.

6. Ternary

- It will return a value depending on the outcome of a Boolean expression.

6. Ternary



Ternary - *Syntax*

```
condition ? trueScope : falseScope;
```

If - Else Statement

```
if (condition == true)
{
   //code will execute in this scope if condition is true
}
else
{
  //code will execute in this scope if condition is false
}
```

Ternary

- Inline if-else statement

```
condition ? trueScope : falseScope;
```

If - Else Statement

```
int x = 1;

string message = “”;


if (x > 0)
{
   message = “This is a positive number.”;
}
else
{
  message = “This is a negative number.”;
}
```

Ternary

- Syntax

```
condition ? trueScope : falseScope;
```

Ternary

- Inline if-else

```
int x = 1;

string message = (x > 0) ? "You are positive!" : "You are
negative.”;
```

Ternary

- Inline if-else

```
int x = -1;

string message = (x > 0) ? "You are positive!" : "You are
negative.”;
```

7. Null Coalescing

7. Null Coalescing

- Backup!

7. Null Coalescing

7. Null Coalescing

- **Null** is the representation of the lack of value.

7. Null Coalescing

- **Null** is the representation of the lack of value.

```
string fruit = null;
```

7. Null Coalescing

- **Coalescing** means “combine” or “come together”

7. Null Coalescing

- **Null Coalescing** - Provides a fallback value in the event that the expression is null.

7. Null Coalescing

- **Is the value null?**

7. Null Coalescing

- **if (value == null) → use backup value**

Null Coalescing - *Syntax*

```
var exampleValue = possibleNullValue ?? someDefaultValue;
```

Null Coalescing - *Syntax*

```
var exampleValue = possibleNullValue ?? someDefaultValue;
```

Backup!

Null Coalescing - *Syntax*

```
var exampleValue = possibleNullValue ?? someDefaultValue;
```

Is it null?

Null Coalescing - *Syntax*

```
var exampleValue = possibleNullValue ?? someDefaultValue;
```

if (possibleNullValue == null)

Null Coalescing - *Syntax*

```
var exampleValue = possibleNullValue ?? someDefaultValue;
```

if (possibleNullValue != null)

Null Coalescing - *Syntax*

```
var exampleValue = possibleNullValue ?? someDefaultValue;
```

if (possibleNullValue != null)

Null Coalescing - *Syntax*

```
var exampleValue = possibleNullValue ?? someDefaultValue;
```

Backup!

Null Coalescing

- Example

```
string fruit = null;
string favoriteFruit = fruit ?? “apple”;
Console.WriteLine(favoriteFruit);
```

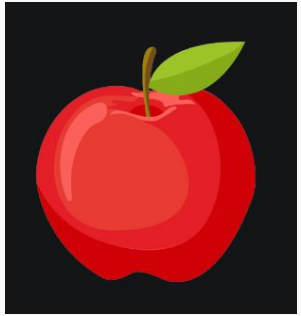
What is the value of favoriteFruit?

Null Coalescing

- Example

```
string fruit = null;
string favoriteFruit = fruit ?? “apple”;
Console.WriteLine(favoriteFruit);
```

What is the value of favoriteFruit?



Null Coalescing

- Example

```
int? a = null;
int? x = a ?? 100;
Console.WriteLine(x);
```

What is the value of x?

Null Coalescing

- Example

```
int? a = null;
int? x = a ?? 100;
Console.WriteLine(x);
```

The value of x is

Null Coalescing

- Example

```
int? a = null;
int? x = a ?? 100;
Console.WriteLine(x);
```

a is null so we assign x the fallback value of 100

Last one

8. Null Conditional

- Applies an operation to its operand **only if that operand is non-null.** Otherwise, the result of applying the operator is null.

Null Conditional - *Syntax*

- Only does something if something is **NOT** Null

Null Conditional - *Syntax*

- if (value != null) → do action

Null Conditional - *Syntax*

```
List<string> myList = new List<string>();
```

Null Conditional - *Syntax*

```
List<string> myList = new List<string>();

myList.Add("myString");
```

Null Conditional - *Syntax*

```
List<string> myList = null;

myList.Add("myString");
```

Null Conditional

```
List<string> myList = null;

myList.Add("myString");
```

- **Run-time Error!** The list has not been instantiated!

Null Conditional - *Syntax*

```
List<string> myList = null;

myList?.Add("myString");
```

Null Conditional Operator

Null Conditional - *Syntax*

```
List<string> myList = null;

myList?.Add("myString");
```

myString will not be added to the list

Null Conditional - *Syntax*

```
List<string> myList = null;

myList?.Add("myString");
```

Why?

Null Conditional - *Syntax*

```
List<string> myList = null;

myList?.Add("myString");
```

Because myList is null

Null Conditional - *Syntax*

```
List<string> myList = new List<string>();

myList?.Add("myString");
```

Null Conditional - *Syntax*

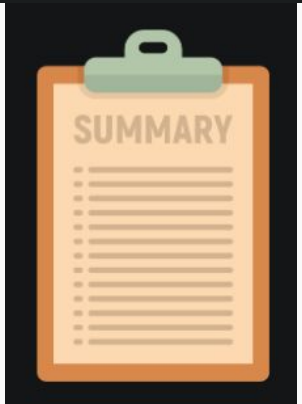
```
List<string> myList = new List<string>();

myList?.Add("myString");
```

myList is non-null, so myString is added

Null Conditional

In summary:

The null conditional operator is used as a verification process. It will only execute an operation if the code is properly set up. Essentially, it acts as a null check at run-time.




QUIZ
TIME!!!

What is this? And what does it do?

What is its return type?

String!

What if i need a number?

Cannot assign a string value to an int container

For example, you **cannot do this: int number = “hello”;**

What if i need a number?

Parse Definition

parse

[pärs]

VERB

1. analyze (a sentence) into its parts and describe their syntactic roles:
   "I asked a couple of students to parse these sentences for me"

NOUN COMPUTING

1. an act of or the result obtained by parsing a string or a text:
   "a failed parse was retried"

For Portal:

- Applies an operation to its operand only if that operand is non-null. Otherwise, the result of applying the operator is null.

Example using conditional operator and non-null value:

```
List<string> myList = new List<string>();

myList?.Add("myString");
```

Explanation:

myList has been instantiated properly, and is not null. Because it is not null, the operation will be successful and “myString” will be added to the myList.

For Portal:

- Applies an operation to its operand only if that operand is non-null. Otherwise, the result of applying the operator is null.

Example using conditional operator and null value:

```
List<string> myList = null;

myList?.Add("myString");
```

Explanation:

myList has a null value, so the operation will not execute. “myString” will **not** be added to myList.

For Portal:

Without the null conditional operator and null value:

```
List<string> myList = null;

myList.Add("myString");
```

Explanation:

Here, we are not using the null conditional operator. Not using it would cause a compile error. You cannot add to a list that has not been instantiated.

For Portal:

Why the null conditional operator is important:

The null conditional operator is used as a verification process. It will only execute an operation if the code is properly set up. It skips over an operation, if it will not be able to successfully operate. Essentially, it acts as a null check at compile time.

Operators

- Use walkthrough documentation for exercises

Null Coalescing Demo

Null Conditional Operator

Assignment Operator Demo

Relational Operators Demo

Unary Operator Demo

Mathematical Operator Demo

Ternary Demo

Logical Operators Demo