

- *passing parameters to functions in C#.*

- **Call (Value Type by Value).**

- ✓ *By default, parameters in C# are passed by value.*
- ✓ *When passing by value, a copy of the value is passed to the function.*
- ✓ *inside the function do not affect the original value in the calling code.*

- **Call (Value Type by Reference).**

- ✓ *To pass parameters by reference, you can use the `ref` Keyword.*
- ✓ *When passing by reference, the memory address (reference) of the variable.*
- ✓ *Allowing changes to affect the original value.*

- **Call (Reference Type by Value).**

- ✓ *When you pass a reference type by value to a method:*
- ✓ *'copy of the reference (memory address) is passed, not the actual object'.*
- ✓ *The method receives a copy of the reference, allowing access to the same object in memory.*
- ✓ *Can be Access Data in This Reference(Applied Some Operation), Such '++';*

- **Call (Reference Type by Reference).**

- ✓ *When you pass a reference type by reference to a method.*
- ✓ *'you are passing a reference to the original reference variable, not just a copy of the reference.'*
- ✓ *modifications made to the reference inside the method will affect the original reference.*
- ✓ *Such As 'Swap 2 Array'.*

- **Call by 'out'.**

- ✓ *The out parameter modifier is similar to the ref modifier.*
- ✓ *The out modifier is used when a method needs to return multiple values.*
- ✓ *must be assigned a value inside the method before it returns.*
- ✓ *useful when a method needs to modify the value of a parameter and return it as an output.*

- Note: Differences between 'out' and 'ref':

- *In out must be 'assigned a value inside the method before it return'*

```
void CalculateSumAndDifference(int a, int b, out int sum, out int difference)
{
    sum = a + b;
    difference = a - b;
}

bool TryDivide(int dividend, int divisor, out int result)
{
    if (divisor != 0)
    {
        result = dividend / divisor;
        return true;
    }
    else
    {
        result = 0;
        return false;
    }
}
```

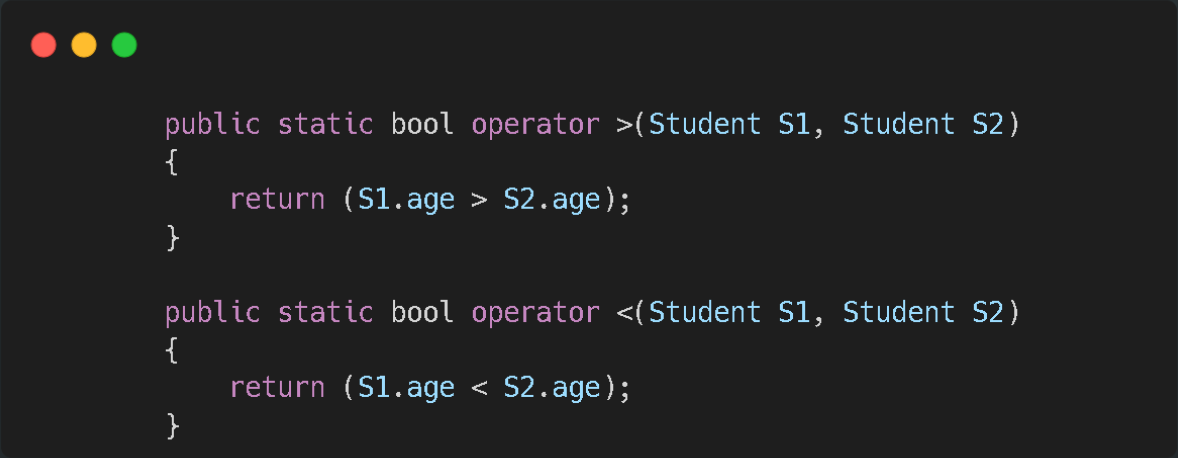
- *Operator overloading*

- *in C# allows you to redefine the behavior of operators such as +, -, *, /, ==, !=, <, >, etc.*

Or (custom behaviors for operators).

- *The overloaded operator method must be declared as `public` and `static`.*
- *Must defined inside a class using the `operator` keyword followed by the operator being overloaded.*

Example:



```
public static bool operator >(Student S1, Student S2)
{
    return (S1.age > S2.age);
}

public static bool operator <(Student S1, Student S2)
{
    return (S1.age < S2.age);
}
```

- **User Define Casting (custom type conversion).**
- ✓ Allows you to define how objects of a user-defined type are converted to other types.
- ✓ you can define explicit and implicit conversion operators for your classes.
- ✓ **Implicit Conversion:** Implicit conversion allows automatic type conversion from one type to another without explicit casting syntax.
- ✓ **Explicit Conversion:** Explicit conversion requires explicit casting syntax.
- ✓ **Must Be Define Implicit Or Explicit (Not Both).**

```

public static implicit operator int(Student S1)
{
    return S1.id;
}

////Must Be Define Implicit Or Explicit (Not Both).

public static explicit operator int(Student S1)
{
    return S1.id;
}

```

- **Access Modifier:**

Access Modifier	Accessibility
public	➤ Everywhere
private	➤ Only within the declaring class or struct
protected	➤ Within the declaring class or struct and its <u>subclasses</u>
internal	➤ Within the <u>assembly</u> that declares it and other assemblies in the same .NET Framework version
protected internal	➤ Within the declaring class or struct, its subclasses, and other assemblies in the same .NET Framework version.