

## 34. What is operator overloading?

**Brief summary:** Operator overloading is a mechanism that allows us to provide custom behavior when objects of the type we defined are used as operands for some operators. For example, we can define what will "obj1+obj2" do.

C# provides many operators, for example +, -, ++, ?: etc. The important thing to understand about operators is that their behavior differs depending on what types they are used with. For example, adding two numbers with the + operator will simply calculate the sum of numbers, while adding two strings with the same operator will concatenate those two strings.

When defining our own types, we would often like to provide a custom implementation for some of the operators. Let's consider a simple Point type:

```
record struct Point
{
    4 references
    public float X { get; }
    4 references
    public float Y { get; }

    6 references
    public Point(float x, float y)
    {
        X = x;
        Y = y;
    }
}
```

We would like to define the operation of adding two points - it should work by adding their X and Y coordinates, for example adding (10,5) point to (3, -2) shall give a new Point with coordinates (13, 3). We can achieve it by defining the Add method in the Point record struct:

```
public Point Add(Point other)
{
    return new Point(X + other.X, Y + other.Y);
}
```

This is correct, but it's a bit awkward to use. To add two Points we will need to write something like this:

```
var point1 = new Point(10, 5);
var point2 = new Point(-3, 4);
var result = point1.Add(point2);
```

It would be more natural to perform the addition with the + operator: it is, after all, the addition operator. Unfortunately, this doesn't work:

```
var point1 = new Point(10, 5);
var point2 = new Point(-3, 4);
var result = point1 + point2;
```

(local variable) Point point2  
CS0019: Operator '+' cannot be applied to operands of type 'Point' and 'Point'

The compiler doesn't know how to add two points yet. To enable the addition of two objects of this type we must overload the addition operator:

```
public static Point operator +(Point point1, Point point2) =>
    new Point(point1.X + point2.X, point1.Y + point2.Y);
```

As you can see to overload the operator we must define a **static** method using the "**operator**" keyword. We must define the parameters and the return type just like in regular methods. In the case of an addition, there are two operands, so we have two parameters. Remember - the operand is the thing to which the operator is applied, for example when adding 3+5 we have two operands: 3 and 5.

Please note that they are operators taking less or more operands. For example, the ++ operator that increases the number by one only takes one operand.

```
int a = 5;
++a;
```

On the other hand, the ternary conditional operator takes three operands: the condition, value if true, and value if false:

```
int a = 5;
var text = a > 1000 ? "big number" : "small number";
```

All right. We overloaded the addition operator for Point type, and now we can safely write this:

```
var point1 = new Point(10, 5);
var point2 = new Point(-3, 4);
var result = point1 + point2;
```

We can overload most of the C# operators, but not all of them. For example, we can't overload lambda operator =>, member access operator (a dot, like in obj.Property), or "new" operator. You can find the full list of overloadable operators here, and at the bottom of the table all non-overloadable operators are listed:


<https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/operators/operator-overloading>

I don't want to show you the overloads for all operators because the code would mostly be the same. But let me show you two interesting and commonly used operators: the **explicit and implicit conversion operators**. First, let me show you some examples of their usage for built-in types.

```
int a = 5;
double b = a;
```

This code looks innocent, but there is more going on here than it seems. After all, we assign an integer to a double. They are two different types, so how does it work? Well, it works because implicit conversion happens. The "a" integer is implicitly converted to a double. Now, let's see the opposite assignment:

```
double c = 5.5d;  
int d = c;
```

 (local variable) double c

CS0266: Cannot implicitly convert type 'double' to 'int'. An explicit conversion exists (are you missing a cast?)

As you can see, this doesn't work. You might be asking, why did it work when assigning an int to a double, but it doesn't work for the opposite operation? The reason for that is simple: the conversion of an int to a double is lossless. The double type can represent the value of 5 that was stored in an int variable. On the other hand, the integer can't represent the number 5.5. When converting 5.5 to int, we will lose some accuracy of the data. The result will be trimmed to a full 5. That's why we must perform such conversion explicitly, so there is no chance we will do it by accident. When using explicit cast we say "I know what I'm doing and I'm aware that the value might actually change during the conversion - I'm ready to take this risk and handle it".


```
double c = 5.5d;  
int d = (int)c;
```

By adding "(int)" I performed the explicit conversion from 5.5 double to int. The result will be 5.

It is quite a common use case that we want to overload the conversion operators. Let's go back to the Point type example. Let's say that our application is getting the points data from some external source and that the points are delivered to us as tuples. We would like to be able to simply assign a tuple of two floats to a variable of Point type, thus performing implicit conversion:

```
Point fromTuple = (5, 7);
```



 (field) int (int, int).Item1

Gets the value of the current (T1, T2) instance's first element.

CS0029: Cannot implicitly convert type '(int, int)' to 'Point'

Show potential fixes (Ctrl+.)

Well, it doesn't work. The compiler doesn't know how to cast a tuple of two numbers to the Point type. We must implement our own implicit conversion operator:

```
public static implicit operator Point((float, float) externalPoint) =>
    new Point(externalPoint.Item1, externalPoint.Item2);
```

We can also overload the explicit conversion operator. If only the explicit conversion operator is implemented, we will have to cast the tuple to Point explicitly:

```
Point fromTuple = (Point)(5, 7);
```

To overload the explicit cast operator we must write this:

```
public static explicit operator Point((float, float) externalPoint) =>
    new Point(externalPoint.Item1, externalPoint.Item2);
```

As you can see, for conversion operators overloading the "explicit" or "implicit" keyword is needed.

Before we wrap up, let's think about when we should use the implicit, and when the explicit casting operator overloading. We can safely use implicit casting when the cast is lossless, so it won't change the underlying data - for example, it won't change its precision. In all other cases, we should use explicit casting, so no data-losing operations are executed behind the scenes, without the programmer's intention.

Let's summarize. We can provide custom behavior for operators use in our own types by using operators overloading. We can overload most, but not all C# operators. We can also overload the implicit and explicit conversion operators.

### Bonus questions:

- **"What is the purpose of the "operator" keyword?"**  
*It is used when overloading an operator for a type.*
- **"What is the difference between explicit and implicit conversion?"**  
*Implicit conversion happens when we assign a value of one type to a variable of another type, without specifying the target type in the parenthesis. For example, it happens when assigning an int to a double. Explicit conversion requires specifying the type in parenthesis, for example when assigning a double to an int.*