CHAPTER 4

Operators

Operators are special symbols used to operate on values. They can be grouped into five types: arithmetic, assignment, comparison, logical, and bitwise operators.

Arithmetic Operators

The arithmetic operators include the four basic arithmetic operations, as well as the modulus operator (%), which is used to obtain the division remainder.

```
float x = 3 + 2; // 5 // addition
      x = 3 - 2; // 1 // subtraction
      x = 3 * 2; // 6 // multiplication
      x = 3 / 2; // 1 // division
      x = 3 % 2; // 1 // modulus (division remainder)
```

Notice that the division sign gives an incorrect result. This is because it operates on two integer values and will therefore round the result and return an integer. To get the correct value, one of the numbers needs to be converted into a floating-point number.

```
x = 3 / (float)2; // 1.5
```


M. Olsson, *C# 8 Quick Syntax Reference*, https://doi.org/10.1007/978-1-4842-5577-3_4

Assignment Operators

The next group is the assignment operators. Most importantly is the assignment operator (=) itself, which assigns a value to a variable.

Combined Assignment Operators

A common use of the assignment and arithmetic operators is to operate on a variable and then to save the result back into that same variable. These operations can be shortened with the combined assignment operators.

```
int x = 0;
    x += 5; // x = x+5;
    x -= 5; // x = x-5;
    x *= 5; // x = x*5;
    x /= 5; // x = x/5;
    x %= 5; // x = x%5;
```

Increment and Decrement Operators

Another common operation is to increment or decrement a variable by one. This can be simplified with the increment (++) and decrement (--) operators.

```
x++; // x = x+1;
x--; // x = x-1;
```

Both of these operators can be used before or after a variable.

```
x++; // post-increment
x--; // post-decrement
++x; // pre-increment
--x; // pre-decrement
```

The result on the variable is the same whichever is used. The difference is that the post-operator returns the original value before it changes the variable, while the pre-operator changes the variable first and then returns the value.

```
int x, y;
x = 5; y = x++; // y=5, x=6
x = 5; y = ++x; // y=6, x=6
```

Comparison Operators

The comparison operators compare two values and return `true` or `false`. They are mainly used to specify conditions, which are expressions that evaluate to `true` or `false`.

```
bool b = (2 == 3); // equal to (false)
     b = (2 != 3); // not equal to (true)
     b = (2 > 3);  // greater than (false)
     b = (2 < 3);  // less than (true)
     b = (2 >= 3); // greater than or equal to (false)
     b = (2 <= 3); // less than or equal to (true)
```

Logical Operators

The logical operators are often used together with the comparison operators. Logical and (&&) evaluates to `true` if both the left and right side are true, and logical or (||) evaluates to `true` if either the left or right side is true. The logical not (!) operator is used for inverting a Boolean result. Note that for both "logical and" and "logical or," the right side of the operator will not be evaluated if the result is already determined by the left side.

```
bool b = (true && false); // logical and (false)
     b = (true || false); // logical or (true)
     b = !(true);         // logical not (false)
```

Bitwise Operators

The bitwise operators can manipulate individual bits inside an integer. For example, the bitwise and (&) operator makes the resulting bit 1 if the corresponding bits on both sides of the operator are set.

```
int x = 5 & 4;  // and (0b101 & 0b100 = 0b100 = 4)
    x = 5 | 4;  // or (0b101 | 0b100 = 0b101 = 5)
    x = 5 ^ 4;  // xor (0b101 ^ 0b100 = 0b001 = 1)
    x = 4 << 1; // left shift (0b100 << 1 = 0b1000 = 8)
    x = 4 >> 1; // right shift (0b100 >> 1 = 0b10 = 2)
    x = ~4;     // invert (~0b00000100 = 0b11111011 = -5)
```

These bitwise operators have shorthand assignment operators, just like the arithmetic operators.

```
int x=5; x &= 4;  // and (0b101 & 0b100 = 0b100 = 4)
    x=5; x |= 4;  // or (0b101 | 0b100 = 0b101 = 5)
    x=5; x ^= 4;  // xor (0b101 ^ 0b100 = 0b001 = 1)
    x=5; x <<= 1; // left shift (0b101 << 1 = 0b1010 = 10)
    x=5; x >>= 1; // right shift (0b101 >> 1 = 0b10 = 2)
```

Operator Precedents

In C#, expressions are normally evaluated from left to right. However, when an expression contains multiple operators, the precedence of those operators decides the order in which they are evaluated. The order of precedence can be seen in the following table, where the operator with the lower precedence will be evaluated first.

Pre	Operator	Pre	Operator
1	++ -- ! ~	7	&
2	* / %	8	^
3	+ -	9	|
4	<< >>	10	&&
5	< <= > >=	11	||
6	== !=	12	= op=

For example, logical and (&&) binds weaker than relational operators, which in turn bind weaker than arithmetic operators.

```
bool x = 2+3 > 1*4 && 5/5 == 1; // true
```

To make things clearer, parentheses can be used to specify which part of the expression will be evaluated first. Parentheses have the greatest precedence of all operators.

```
bool x = ((2+3) > (1*4)) && ((5/5) == 1); // true
```