Chapter 3: Operators

Parameter	Details
operatorSymbol	The operator being overloaded, e.g. +, -, /, *
OperandType	The type that will be returned by the overloaded operator.
operand1	The first operand to be used in performing the operation.
operand2	The second operand to be used in performing the operation, when doing binary operations.
statements	Optional code needed to perform the operation before returning the result.

In C#, an operator is a program element that is applied to one or more operands in an expression or statement. Operators that take one operand, such as the increment operator (++) or new, are referred to as unary operators. Operators that take two operands, such as arithmetic operators (+,-,*,/), are referred to as binary operators. One operator, the conditional operator (?:), takes three operands and is the sole ternary operator in C#.

Section 3.1: Overloadable Operators

C# allows user-defined types to overload operators by defining static member functions using the `operator` keyword.
The following example illustrates an implementation of the `+` operator.

If we have a `Complex` class which represents a complex number:

```
public struct Complex
{
    public double Real { get; set; }
    public double Imaginary { get; set; }
}
```

And we want to add the option to use the `+` operator for this class. i.e.:

```
Complex a = new Complex() { Real = 1, Imaginary = 2 };
Complex b = new Complex() { Real = 4, Imaginary = 8 };
Complex c = a + b;
```

We will need to overload the `+` operator for the class. This is done using a static function and the `operator` keyword:

```
public static Complex operator +(Complex c1, Complex c2)
{
   return new Complex
   {
       Real = c1.Real + c2.Real,
       Imaginary = c1.Imaginary + c2.Imaginary
   };
}
```

Operators such as `+`, `-`, `*`, `/` can all be overloaded. This also includes Operators that don't return the same type (for example, `==` and `!=` can be overloaded, despite returning booleans) The rule below relating to pairs is also enforced here.

Comparison operators have to be overloaded in pairs (e.g. if `<` is overloaded, `>` also needs to be overloaded).

A full list of overloadable operators (as well as non-overloadable operators and the restrictions placed on some overloadable operators) can be seen at MSDN - Overloadable Operators (C# Programming Guide).

Version ≥ 7.0

overloading of `operator is` was introduced with the pattern matching mechanism of C# 7.0. For details see Pattern Matching

Given a type `Cartesian` defined as follows

```
public class Cartesian
{
    public int X { get; }
    public int Y { get; }
}
```

An overloadable `operator is` could e.g. be defined for `Polar` coordinates

```
public static class Polar
{
    public static bool operator is(Cartesian c, out double R, out double Theta)
    {
        R = Math.Sqrt(c.X*c.X + c.Y*c.Y);
        Theta = Math.Atan2(c.Y, c.X);
        return c.X != 0 || c.Y != 0;
    }
}
```

which can be used like this

```
var c = Cartesian(3, 4);
if (c is Polar(var R, *))
{
    Console.WriteLine(R);
}
```

(The example is taken from the Roslyn Pattern Matching Documentation)

Section 3.2: Overloading equality operators

Overloading just equality operators is not enough. Under different circumstances, all of the following can be called:

1. `object.Equals` and `object.GetHashCode`
2. `IEquatable<T>.Equals` (optional, allows avoiding boxing)
3. `operator ==` and `operator !=` (optional, allows using operators)

When overriding `Equals`, `GetHashCode` must also be overridden. When implementing `Equals`, there are many special cases: comparing to objects of a different type, comparing to self etc.

When NOT overridden `Equals` method and `==` operator behave differently for classes and structs. For classes just references are compared, and for structs values of properties are compared via reflection what can negatively affect performance. `==` can not be used for comparing structs unless it is overridden.

Generally equality operation must obey the following rules:

- Must not *throw exceptions*.
- Reflexivity: `A` always equals `A` (may not be true for `NULL` values in some systems).
- Transitvity: if `A` equals `B`, and `B` equals `C`, then `A` equals `C`.
- If `A` equals `B`, then `A` and `B` have equal hash codes.

- Inheritance tree independence: if B and C are instances of Class2 inherited from Class1: Class1.Equals(A,B) must always return the same value as the call to Class2.Equals(A,B).

```
class Student : IEquatable<Student>
{
    public string Name { get; set; } = "";

    public bool Equals(Student other)
    {
        if (ReferenceEquals(other, null)) return false;
        if (ReferenceEquals(other, this)) return true;
        return string.Equals(Name, other.Name);
    }

    public override bool Equals(object obj)
    {
        if (ReferenceEquals(null, obj)) return false;
        if (ReferenceEquals(this, obj)) return true;

        return Equals(obj as Student);
    }

    public override int GetHashCode()
    {
        return Name?.GetHashCode() ?? 0;
    }

    public static bool operator ==(Student left, Student right)
    {
        return Equals(left, right);
    }

    public static bool operator !=(Student left, Student right)
    {
        return !Equals(left, right);
    }
}
```

Section 3.3: Relational Operators

Equals

Checks whether the supplied operands (arguments) are equal

```
"a" == "b"     // Returns false.
"a" == "a"     // Returns true.
1 == 0         // Returns false.
1 == 1         // Returns true.
false == true  // Returns false.
false == false // Returns true.
```

Unlike Java, the equality comparison operator works natively with strings.

The equality comparison operator will work with operands of differing types if an implicit cast exists from one to the other. If no suitable implicit cast exists, you may call an explicit cast or use a method to convert to a compatible type.

```
1 == 1.0              // Returns true because there is an implicit cast from int to double.
new Object() == 1.0   // Will not compile.
```

```
MyStruct.AsInt() == 1 // Calls AsInt() on MyStruct and compares the resulting int with 1.
```

Unlike Visual Basic.NET, the equality comparison operator is not the same as the equality assignment operator.

```
var x = new Object();
var y = new Object();
x == y // Returns false, the operands (objects in this case) have different references.
x == x // Returns true, both operands have the same reference.
```

Not to be confused with the assignment operator (=).

For value types, the operator returns `true` if both operands are equal in value.
For reference types, the operator returns `true` if both operands are equal in *reference* (not value). An exception is that string objects will be compared with value equality.

Not Equals

Checks whether the supplied operands are *not* equal.

```
"a" != "b"     // Returns true.
"a" != "a"     // Returns false.
1 != 0         // Returns true.
1 != 1         // Returns false.
false != true  // Returns true.
false != false // Returns false.

var x = new Object();
var y = new Object();
x != y // Returns true, the operands have different references.
x != x // Returns false, both operands have the same reference.
```

This operator effectively returns the opposite result to that of the equals (`==`) operator

Greater Than

Checks whether the first operand is greater than the second operand.

```
3 > 5    //Returns false.
1 > 0    //Returns true.
2 > 2    //Return false.

var x = 10;
var y = 15;
x > y    //Returns false.
y > x    //Returns true.
```

Less Than

Checks whether the first operand is less than the second operand.

```
2 < 4     //Returns true.
1 < -3    //Returns false.
2 < 2     //Return false.

var x = 12;
var y = 22;
x < y    //Returns true.
y < x    //Returns false.
```

Greater Than Equal To

Checks whether the first operand is greater than equal to the second operand.

```
7 >= 8    //Returns false.
0 >= 0    //Returns true.
```

Less Than Equal To

Checks whether the first operand is less than equal to the second operand.

```
2 <= 4    //Returns true.
1 <= -3    //Returns false.
1 <= 1     //Returns true.
```

Section 3.4: Implicit Cast and Explicit Cast Operators

C# allows user-defined types to control assignment and casting through the use of the `explicit` and `implicit` keywords. The signature of the method takes the form:

```
public static <implicit/explicit> operator <ResultingType>(<SourceType> myType)
```

The method cannot take any more arguments, nor can it be an instance method. It can, however, access any private members of type it is defined within.

An example of both an `implicit` and `explicit` cast:

```
public class BinaryImage
{
    private bool[] _pixels;

    public static implicit operator ColorImage(BinaryImage im)
    {
        return new ColorImage(im);
    }

    public static explicit operator bool[](BinaryImage im)
    {
        return im._pixels;
    }
}
```

Allowing the following cast syntax:

```
var binaryImage = new BinaryImage();
ColorImage colorImage = binaryImage; // implicit cast, note the lack of type
bool[] pixels = (bool[])binaryImage; // explicit cast, defining the type
```

The cast operators can work both ways, going *from* your type and going *to* your type:

```
public class BinaryImage
{
    public static explicit operator ColorImage(BinaryImage im)
    {
        return new ColorImage(im);
    }
```

```
    public static explicit operator BinaryImage(ColorImage cm)
    {
        return new BinaryImage(cm);
    }
}
```

Finally, the `as` keyword, which can be involved in casting within a type hierarchy, is **not** valid in this situation. Even after defining either an `explicit` or `implicit` cast, you cannot do:

```
ColorImage cm = myBinaryImage as ColorImage;
```

It will generate a compilation error.

Section 3.5: Short-circuiting Operators

By definition, the short-circuiting boolean operators will only evaluate the second operand if the first operand can not determine the overall result of the expression.

It means that, if you are using && operator as *firstCondition && secondCondition* it will evaluate *secondCondition* only when *firstCondition* is true and ofcource the overall result will be true only if both of *firstOperand* and *secondOperand* are evaluated to true. This is useful in many scenarios, for example imagine that you want to check whereas your list has more than three elements but you also have to check if list has been initialized to not run into *NullReferenceException*. You can achieve this as below:

```
bool hasMoreThanThreeElements = myList != null && mList.Count > 3;
```

mList.Count > *3* will not be checked untill myList != null is met.

Logical AND

`&&` is the short-circuiting counterpart of the standard boolean AND (`&`) operator.

```
var x = true;
var y = false;

x && x // Returns true.
x && y // Returns false (y is evaluated).
y && x // Returns false (x is not evaluated).
y && y // Returns false (right y is not evaluated).
```

Logical OR

`||` is the short-circuiting counterpart of the standard boolean OR (`|`) operator.

```
var x = true;
var y = false;

x || x // Returns true (right x is not evaluated).
x || y // Returns true (y is not evaluated).
y || x // Returns true (x and y are evaluated).
y || y // Returns false (y and y are evaluated).
```

Example usage

```
if(object != null && object.Property)
// object.Property is never accessed if object is null, because of the short circuit.
```

```
    Action1();
else
    Action2();
```

Section 3.6: ? : Ternary Operator

Returns one of two values depending on the value of a Boolean expression.

Syntax:

```
condition ? expression_if_true : expression_if_false;
```

Example:

```
string name = "Frank";
Console.WriteLine(name == "Frank" ? "The name is Frank" : "The name is not Frank");
```

The ternary operator is right-associative which allows for compound ternary expressions to be used. This is done by adding additional ternary equations in either the true or false position of a parent ternary equation. Care should be taken to ensure readability, but this can be useful shorthand in some circumstances.

In this example, a compound ternary operation evaluates a `clamp` function and returns the current value if it's within the range, the `min` value if it's below the range, or the `max` value if it's above the range.

```
light.intensity = Clamp(light.intensity, minLight, maxLight);

public static float Clamp(float val, float min, float max)
{
    return (val < min) ? min : (val > max) ? max : val;
}
```

Ternary operators can also be nested, such as:

```
a ? b ? "a is true, b is true" : "a is true, b is false" : "a is false"

// This is evaluated from left to right and can be more easily seen with parenthesis:

a ? (b ? x : y) : z

// Where the result is x if a && b, y if a && !b, and z if !a
```

When writing compound ternary statements, it's common to use parenthesis or indentation to improve readability.

The types of *expression_if_true* and *expression_if_false* must be identical or there must be an implicit conversion from one to the other.

```
condition ? 3 : "Not three"; // Doesn't compile because `int` and `string` lack an implicit
conversion.

condition ? 3.ToString() : "Not three"; // OK because both possible outputs are strings.

condition ? 3 : 3.5; // OK because there is an implicit conversion from `int` to `double`. The
ternary operator will return a `double`.

condition ? 3.5 : 3; // OK because there is an implicit conversion from `int` to `double`. The
ternary operator will return a `double`.
```

The type and conversion requirements apply to your own classes too.

```
public class Car
{}

public class SportsCar : Car
{}

public class SUV : Car
{}

condition ? new SportsCar() : new Car(); // OK because there is an implicit conversion from
`SportsCar` to `Car`. The ternary operator will return a reference of type `Car`.

condition ? new Car() : new SportsCar(); // OK because there is an implicit conversion from
`SportsCar` to `Car`. The ternary operator will return a reference of type `Car`.

condition ? new SportsCar() : new SUV(); // Doesn't compile because there is no implicit conversion
from `SportsCar` to SUV or `SUV` to `SportsCar`. The compiler is not smart enough to realize that
both of them have an implicit conversion to `Car`.

condition ? new SportsCar() as Car : new SUV() as Car; // OK because both expressions evaluate to a
reference of type `Car`. The ternary operator will return a reference of type `Car`.
```

Section 3.7: ?. (Null Conditional Operator)

Version ≥ 6.0

Introduced in C# 6.0, the Null Conditional Operator `?.` will immediately return `null` if the expression on its left-hand side evaluates to `null`, instead of throwing a `NullReferenceException`. If its left-hand side evaluates to a non-`null` value, it is treated just like a normal `.` operator. Note that because it might return `null`, its return type is always a nullable type. That means that for a struct or primitive type, it is wrapped into a `Nullable<T>`.

```
var bar = Foo.GetBar()?.Value; // will return null if GetBar() returns null
var baz = Foo.GetBar()?.IntegerValue; // baz will be of type Nullable<int>, i.e. int?
```

This comes handy when firing events. Normally you would have to wrap the event call in an if statement checking for `null` and raise the event afterwards, which introduces the possibility of a race condition. Using the Null conditional operator this can be fixed in the following way:

```
event EventHandler<string> RaiseMe;
RaiseMe?.Invoke("Event raised");
```

Section 3.8: "Exclusive or" Operator

The operator for an "exclusive or" (for short XOR) is: ^

This operator returns true when one, but only one, of the supplied bools are true.

```
true ^ false   // Returns true
false ^ true   // Returns true
false ^ false  // Returns false
true ^ true    // Returns false
```

Section 3.9: default Operator

Value Type (where T : struct)

The built-in primitive data types, such as char, int, and float, as well as user-defined types declared with struct, or enum. Their default value is new T() :

```
default(int)            // 0
default(DateTime)       // 0001-01-01 12:00:00 AM
default(char)           // '\0' This is the "null character", not a zero or a line break.
default(Guid)           // 00000000-0000-0000-0000-000000000000
default(MyStruct)       // new MyStruct()

// Note: default of an enum is 0, and not the first *key* in that enum
// so it could potentially fail the Enum.IsDefined test
default(MyEnum)         // (MyEnum)0
```

Reference Type (where T : class)

Any class, interface, array or delegate type. Their default value is null :

```
default(object)         // null
default(string)         // null
default(MyClass)        // null
default(IDisposable)    // null
default(dynamic)        // null
```

Section 3.10: Assignment operator '='

The assignment operator = sets thr left hand operand's value to the value of right hand operand, and return that value:

```
int a = 3;     // assigns value 3 to variable a
int b = a = 5; // first assigns value 5 to variable a, then does the same for variable b
Console.WriteLine(a = 3 + 4); // prints 7
```

Section 3.11: sizeof

Returns an int holding the size of a type* in bytes.

```
sizeof(bool)    // Returns 1.
sizeof(byte)    // Returns 1.
sizeof(sbyte)   // Returns 1.
sizeof(char)    // Returns 2.
sizeof(short)   // Returns 2.
sizeof(ushort)  // Returns 2.
sizeof(int)     // Returns 4.
sizeof(uint)    // Returns 4.
sizeof(float)   // Returns 4.
sizeof(long)    // Returns 8.
sizeof(ulong)   // Returns 8.
sizeof(double)  // Returns 8.
sizeof(decimal) // Returns 16.
```

**Only supports certain primitive types in safe context.*

In an unsafe context, `sizeof` can be used to return the size of other primitive types and structs.

```
public struct CustomType
{
    public int value;
}

static void Main()
{
    unsafe
    {
        Console.WriteLine(sizeof(CustomType)); // outputs: 4
    }
}
```

Section 3.12: ?? Null-Coalescing Operator

The Null-Coalescing operator `??` will return the left-hand side when not null. If it is null, it will return the right-hand side.

```
object foo = null;
object bar = new object();

var c = foo ?? bar;
//c will be bar since foo was null
```

The `??` operator can be chained which allows the removal of `if` checks.

```
//config will be the first non-null returned.
var config = RetrieveConfigOnMachine() ??
             RetrieveConfigFromService() ??
             new DefaultConfiguration();
```

Section 3.13: Bit-Shifting Operators

The shift operators allow programmers to adjust an integer by shifting all of its bits to the left or the right. The following diagram shows the affect of shifting a value to the left by one digit.

Left-Shift

```
uint value = 15;              // 00001111

uint doubled = value << 1;    // Result = 00011110 = 30
uint shiftFour = value << 4;  // Result = 11110000 = 240
```

Right-Shift

```
uint value = 240;             // 11110000

uint halved = value >> 1;     // Result = 01111000 = 120
uint shiftFour = value >> 4;  // Result = 00001111 = 15
```

Section 3.14: => Lambda operator

Version ≥ 3.0

The `=>` operator has the same precedence as the assignment operator `=` and is right-associative.

It is used to declare lambda expressions and also it is widely used with LINQ Queries:

```
string[] words = { "cherry", "apple", "blueberry" };

int shortestWordLength = words.Min((string w) => w.Length); //5
```

When used in LINQ extensions or queries the type of the objects can usually be skipped as it is inferred by the compiler:

```
int shortestWordLength = words.Min(w => w.Length); //also compiles with the same result
```

The general form of lambda operator is the following:

```
(input parameters) => expression
```

The parameters of the lambda expression are specified before `=>` operator, and the actual expression/statement/block to be executed is to the right of the operator:

```
// expression
(int x, string s) => s.Length > x

// expression
(int x, int y) => x + y

// statement
(string x) => Console.WriteLine(x)

// block
(string x) => {
        x += " says Hello!";
        Console.WriteLine(x);
    }
```

This operator can be used to easily define delegates, without writing an explicit method:

```
delegate void TestDelegate(string s);

TestDelegate myDelegate = s => Console.WriteLine(s + " World");

myDelegate("Hello");
```

instead of

```
void MyMethod(string s)
{
    Console.WriteLine(s + " World");
}

delegate void TestDelegate(string s);

TestDelegate myDelegate = MyMethod;

myDelegate("Hello");
```

Section 3.15: Class Member Operators: Null Conditional Member Access

```
var zipcode = myEmployee?.Address?.ZipCode;
//returns null if the left operand is null.
//the above is the equivalent of:
var zipcode = (string)null;
if (myEmployee != null && myEmployee.Address != null)
    zipcode = myEmployee.Address.ZipCode;
```

Section 3.16: Class Member Operators: Null Conditional Indexing

```
var letters = null;
char? letter = letters?[1];
Console.WriteLine("Second Letter is {0}",letter);
//in the above example  rather than throwing an error because letters is null
//letter is assigned the value null
```

Section 3.17: Postfix and Prefix increment and decrement

Postfix increment `X++` will add `1` to `x`

```
var x = 42;
x++;
Console.WriteLine(x); // 43
```

Postfix decrement

`X--`

will subtract one

```
var x = 42
x--;
Console.WriteLine(x); // 41
```

`++x` is called prefix increment it increments the value of x and then returns x while `x++` returns the value of x and then increments

```
var x = 42;
Console.WriteLine(++x); // 43
System.out.println(x); // 43
```

while

```
var x = 42;
Console.WriteLine(x++); // 42
System.out.println(x); // 43
```

both are commonly used in for loop

```
for(int i = 0; i < 10; i++)
{
}
```

Section 3.18: typeof

Gets `System.Type` object for a type.

```
System.Type type = typeof(Point)        //System.Drawing.Point
System.Type type = typeof(IDisposable)  //System.IDisposable
System.Type type = typeof(Colors)       //System.Drawing.Color
System.Type type = typeof(List<>)       //System.Collections.Generic.List`1[T]
```

To get the run-time type, use `GetType` method to obtain the `System.Type` of the current instance.

Operator `typeof` takes a type name as parameter, which is specified at compile time.

```
public class Animal {}
public class Dog : Animal {}

var animal = new Dog();

Assert.IsTrue(animal.GetType() == typeof(Animal)); // fail, animal is typeof(Dog)
Assert.IsTrue(animal.GetType() == typeof(Dog));    // pass, animal is typeof(Dog)
Assert.IsTrue(animal is Animal);                   // pass, animal implements Animal
```

Section 3.19: Binary operators with assignment

C# has several operators that can be combined with an `=` sign to evaluate the result of the operator and then assign the result to the original variable.

Example:

```
x += y
```

is the same as

```
x = x + y
```

Assignment operators:

- `+=`
- `-=`
- `*=`
- `/=`
- `%=`
- `&=`
- `|=`
- `^=`
- `<<=`
- `>>=`

Section 3.20: nameof Operator

Returns a string that represents the unqualified name of a `variable`, `type`, or `member`.

```
int counter = 10;
nameof(counter); // Returns "counter"
Client client = new Client();
```

```
nameof(client.Address.PostalCode)); // Returns "PostalCode"
```

The `nameof` operator was introduced in C# 6.0. It is evaluated at compile-time and the returned string value is inserted inline by the compiler, so it can be used in most cases where the constant string can be used (e.g., the `case` labels in a `switch` statement, attributes, etc...). It can be useful in cases like raising & logging exceptions, attributes, MVC Action links, etc...

Section 3.21: Class Member Operators: Member Access

```
var now = DateTime.UtcNow;
//accesses member of a class.  In this case the UtcNow property.
```

Section 3.22: Class Member Operators: Function Invocation

```
var age = GetAge(dateOfBirth);
//the above calls the function GetAge passing parameter dateOfBirth.
```

Section 3.23: Class Member Operators: Aggregate Object Indexing

```
var letters = "letters".ToCharArray();
char letter = letters[1];
Console.WriteLine("Second Letter is {0}",letter);
//in the above example we take the second character from the array
//by calling letters[1]
//NB: Array Indexing starts at 0; i.e. the first letter would be given by letters[0].
```