

Agenda

- Operator Overloading
- Operator Overloading Implementation



Operator Overloading

- Operator overloading permits user-defined operator implementations to be specified for operations where one or both of the operands are of a user-defined class or struct type.
- User-defined operator implementations take precedence over predefined operator implementations.
 - When no applicable user-defined operator implementations exist, the predefined operator implementations will be considered.

Overloadable Operators

Operators	Overloadability
+, -, !, ~, ++, --, true, false	These unary operators can be overloaded.
+, -, *, /, %, &, , ^, <<, >>	These binary operators can be overloaded.
==, !=, <, >, <=, >=	The comparison operators can be overloaded.
&&,	The conditional logical operators cannot be overloaded, but they are evaluated using & and , which can be overloaded.
[]	The array indexing operator cannot be overloaded, but you can define indexers.
()	The cast operator cannot be overloaded, but you can define new conversion operators.
+=, -=, *=, /=, %=, &=, =, ^=, <<=, >>=	Assignment operators cannot be overloaded, but +=, for example, is evaluated using +, which can be overloaded.
=, ., ?:, ->, new, is, sizeof, typeof	These operators cannot be overloaded.

Note: The comparison operators, if overloaded, must be overloaded in pairs; that is, if == is overloaded, != must also be overloaded. The reverse is also true, and similar for < and >, and for <= and >=.

Operator Overloading Guidelines

- Define operators on value types that are logical built-in language types, such as the `System.Decimal` structure.
- Provide operator-overloading methods only in the class in which the methods are defined. The C# compiler enforces this guideline.
- Use the names and signature conventions described in the Common Language Specification (CLS). The C# compiler does this for you automatically.
- Use operator overloading in cases where it is immediately obvious what the result of the operation will be.
- Overload operators in a symmetrical manner. For example, if you overload the equality operator (`==`), you should also overload the not equal operator (`!=`).

Operator Overloading Guidelines (cont.)

- Provide alternate signatures:
 - Most languages do not support operator overloading. For this reason, it is a CLS requirement for all types that overload operators include a secondary method with an appropriate domain-specific name that provides the equivalent functionality.

Introduction to Indexers

- C# introduces a new concept known as indexers which are used for treating an object as an array.
- Defining an indexer allows you to create classes that act like virtual arrays.
- Instances of that class can be accessed using the [] Array Access operator.

Example for Indexer

```
namespace Indexers
{
    class ParentClass
    {
        private string[] range = new string[5];
        public string this[int indexrange]
        {
            get
            {
                return range[indexrange];
            }
            set
            {
                range[indexrange] = value;
            }
        }
    }
}
```

Example for Indexer (cont.)

```
class childclass
{
    public static void Main()
    {
        ParentClass obj = new ParentClass();

        /* The Above Class ParentClass create one object name is obj */

        obj[0] = "ONE";
        obj[1] = "TWO";
        obj[2] = "THREE";
        obj[3] = "FOUR ";
        obj[4] = "FIVE";
        Console.WriteLine("{0}\n,{1}\n,{2}\n,{3}\n,{4}\n", obj[0], obj[1], obj[2], obj[3], obj[4]);
        Console.WriteLine("\n");
        Console.ReadLine();
    }
}
```


Example for Operator Overloading

```
using System;
class Complex
{
    private int x;
    private int y;
    public Complex()
    {
    }
    public Complex(int i, int j)
    {
        x = i;
        y = j;
    }
    public void ShowXY()
    {
        Console.WriteLine("{0} {1}",x,y);
    }
    public static Complex operator +(Complex c1,Complex c2)
    {
        Complex temp = new Complex();
        temp.x = c1.x+c2.x;
        temp.y = c1.y+c2.y;
        return temp;
    }
}
```

Example for Operator Overloading (cont.)

```
class MyClient
{
    public static void Main()
    {
        Complex c1 = new Complex(10,20);
        c1.ShowXY(); // displays 10 & 20
        Complex c2 = new Complex(20,30);
        c2.ShowXY(); // displays 20 & 30
        Complex c3 = new Complex();
        c3 = c1 + c2;
        c3.ShowXY(); // displays 30 & 50
    }
}
```

Operator Overloading Implementation (1 of 3)

```
using System;

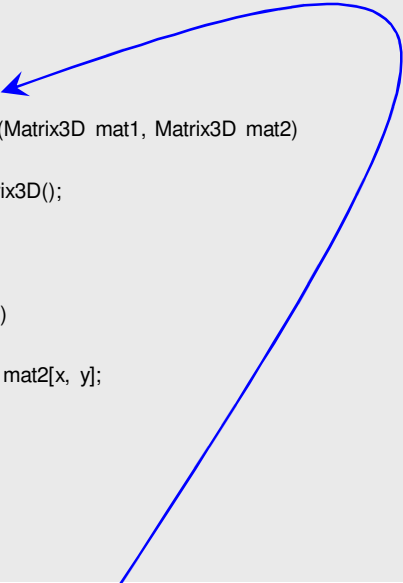
namespace TestOperatorOverloadApps
{
    class Matrix3D
    {
        // its a 3D matrix
        public const int DIMSIZE = 3;
        private double[,] matrix = new double[DIMSIZE, DIMSIZE];

        // allow callers to initialize
        public double this[int x, int y]
        {
            get { return matrix[x, y]; }
            set { matrix[x, y] = value; }
        }

        // let user add matrices
        public static Matrix3D operator +(Matrix3D mat1, Matrix3D mat2)
        {
            Matrix3D newMatrix = new Matrix3D();

            for (int x = 0; x < DIMSIZE; x++)
            {
                for (int y = 0; y < DIMSIZE; y++)
                {
                    newMatrix[x, y] = mat1[x, y] + mat2[x, y];
                }
            }

            return newMatrix;
        }
    }
}
```



Overloading the + operator

```
class TestOperatorOverloading
{
    // used in the InitMatrix method.
    public static Random rand = new Random();

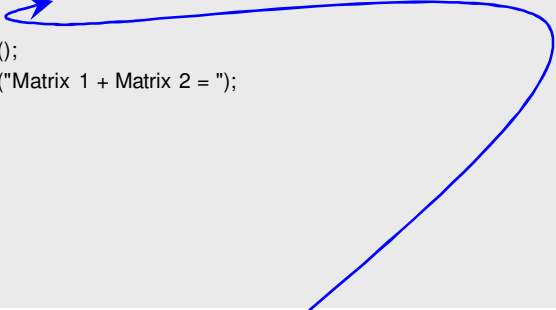
    // test Matrix3D
    static void Main()
    {
        Matrix3D mat1 = new Matrix3D();
        Matrix3D mat2 = new Matrix3D();

        // init matrices with random values
        InitMatrix(mat1);
        InitMatrix(mat2);

        // print out matrices
        Console.WriteLine("Matrix 1: ");
        PrintMatrix(mat1);
        Console.WriteLine("Matrix 2: ");
        PrintMatrix(mat2);

        // perform operation and print out results
        Matrix3D mat3 = mat1 + mat2;

        Console.WriteLine();
        Console.WriteLine("Matrix 1 + Matrix 2 = ");
        PrintMatrix(mat3);
    }
}
```



Invoking the overloaded '+' operator

Operator Overloading Implementation (2 of 3)

```
// initialize matrix with random values
public static void InitMatrix(Matrix3D mat)
{
    for (int x = 0; x < Matrix3D.DIMSIZE; x++)
    {
        for (int y = 0; y < Matrix3D.DIMSIZE; y++)
        {
            mat[x, y] = rand.NextDouble();
        }
    }
}

// print matrix to console
public static void PrintMatrix(Matrix3D mat)
{
    Console.WriteLine();
    for (int x = 0; x < Matrix3D.DIMSIZE; x++)
    {
        Console.Write("[ ");
        for (int y = 0; y < Matrix3D.DIMSIZE; y++)
        {
            // format the output
            Console.Write("{0,8:#.000000}", mat[x, y]);

            if ((y + 1 % 2) < 3)
            {
                Console.Write(", ");
            }
        }
        Console.WriteLine(" ]");
    }
    Console.WriteLine();
}
}
```

Operator Overloading Implementation (3 of 3)

Output*

```
Matrix 1:
[ .215683, .520822, .417234 ]
[ .450960, .066141, .456521 ]
[ .205327, .188658, .115642 ]

Matrix 2:
[ .097595, .674218, .975474 ]
[ .292551, .463416, .156411 ]
[ .382715, .364382, .269819 ]

Matrix 1 + Matrix 2 =
[ .313279, 1.195040, 1.392708 ]
[ .743511, .529557, .612932 ]
[ .588042, .553040, .385462 ]
```

***Note:** The example constructs Matrix 1 and Matrix 2 with random values. Hence output would differ for every execution.