Microsoft .Net Framework and C# Programming Language

STEP IT ACADEMY

Lesson 5

Operators Overloading

Contents

1. Introduction into operator overloading

Operator overloading is a mechanism of defining user-type standard operations. In relation to intrinsic C# types, standard operator overloading function is already implemented and cannot be changed. Thus, for example in `System.String` class, overloading of `==` and `!=` operators is used to verify content-wise lines equity and non-equity, while operator + is overloaded and realizes lines concatenation. It is to be noted that operator overloading is one of polymorphism methods, and thus, for example, applying + operation in relation to numeric types we will get their total amount, while applying it towards lines we will get lines concatenation.

Operator overloading is used to improve programs readability and should comply with certain requirements:

- Operator overloading should be done with public static class methods;
- Operator method type of return value or one of parameters should coincide with a type where operator overloading takes place; Parameters of operator method should not include out and ref modifiers.

Operator overloading can be applied pertaining to both classes and structures; meanwhile, some operator overloading-related restrictions should be noted:

- Overloading cannot change operator precedence;

- Number of operands, being handled by operator, cannot be changed when overloading;
- Not all operators can be overloaded.

Please find below operators that can be overloaded:

Operators	Operator category
–	Change of variable value sign
!	Logical negation operation
~	Bitwise complement operation, resulting in every bit inversion
++, – –	Increment and decrement
true, false	Truth criterion, being defined by class developer
+, –, *, /, %	Arithmetic operators
&, |, ^, <<, >>	Bit operations
==, !=, <, >, <=, >=	Comparison operators
&&, ||	Logical operators
[]	Array elements access operators, being simulated be means of indexers
()	Transformation operations

Operators that cannot be overloaded are specified in the following table:

Operator	Operators category
+=, –=, *=, /=, %=, &=, |=, ^=, <<=, >>=	Automatically overloaded with corresponding binary operation overloading
=	Assignment
.	Access to type members
?:	Conditional operators
new	Object creation
as, is, typeof	Used to get type-pertaining data
–>, sizeof,*,&	Accessible in insecure code only

Overloading syntax looks as follows:

```
public static return_type
              operator operation_symbol (parameters)
{
    //code
}
```

2. Overloading of unitary operators

Due to the fact that overloaded operators are static methods they are not assigned with this pointer, for that reason unitary operators should get a single operand. This operand should have a type of class where operator overloading takes place. So if unitary operator is overloaded in Point class, operand type should also be Point.

Let's review unitary operator overloading, using an example of increment, decrement and sign change (-) operators. In relation to ++ and — operators, a return value should be of the same type as that of operators or its derivative overloading.

Point class describes a point at a plane with x and y coordinates. Increment operator increases both coordinates by 1, decrement operator correspondingly decreases them by 1, operator changes coordinate sign into reverse one (Fig. 2.1).

```
using static System.Console;

namespace SimpleProject
{
    class Point
    {
        public int X { get; set; }
        public int Y { get; set; }
        //Increment overloading
        public static Point operator ++(Point s)
        {
```

```
            s.X++;
            s.Y++;
            return s;
        }

        //decrement overloading
        public static Point operator --(Point s)
        {
            s.X--;
            s.Y--;
            return s;
        }
         //operator overloading -
        public static Point operator -(Point s)
        {
            return new Point { X = -s.X, Y = -s.Y };
        }

        public override string ToString()
        {
            return $"Point: X = {X}, Y = {Y}";
        }
    }

    class Program
    {
        static void Main()
        {
            Point point = new Point { X = 10, Y = 10 };
            WriteLine($"Start point\n{point}");

            WriteLine("Pre- and post-increments
                      are executed in the same way ");

            WriteLine(++point); // x=11, y=11
            WriteLine(point++); // x=12, y=12

            WriteLine($"Pre-decrement\n
                       {--point}");
```

```
            WriteLine($"Operator execution -\n
                       {-point}");

            WriteLine($"start point is not changed\n
                      {point}");
        }
    }
}
```

Program outcome.

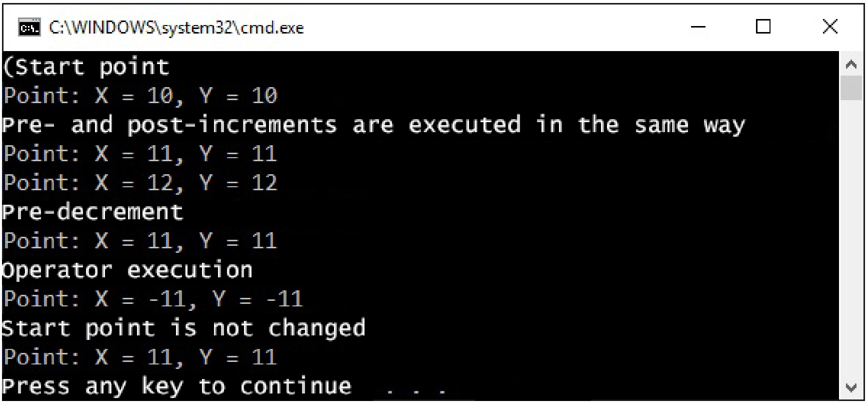


Fig. 2.1. Unitary operator overloading

In this example, `Point` is of reference type, for that reason change of x and y values, being realized in increment and decrement overloaded operators, change transferred object as well. Operator — (sign change) should not change transferred object status but should return a new object with reversed sign. In order to implement this method a new `Point` object is created, its coordinate sign is changed and this object is returned from the method.

It is interesting to note that C# has no functionalities to overload pre- and post-increments and decrements separately.

For that reason, when calling, post and pre forms operate in the same way as pre-form.

Operators are overloaded with creation of special class methods. Let's review the `Point` class with the help of ildasm (Fig. 2.2).

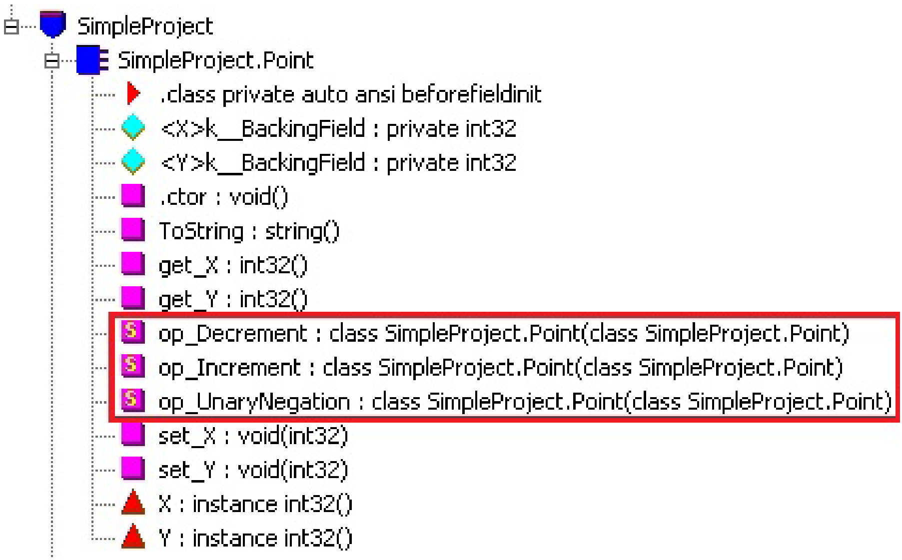


Fig. 2.2. Point class data in disassembler

It is easy to find out the following correspondence of overloaded operators to generated methods:

- operator -- op_Decrement
- operator ++ op_Increment
- operator - op_UnaryNegation

Metadata shows that these methods have a specialname flag (Pic 2.3).

```
.method public hidebysig specialname static
        class SimpleProject.Point  op_Increment(class SimpleProject.Point s) cil managed
{
  // Размер кода:       41 (0x29)
  .maxstack  3
  .locals init ([0] int32 V_0,
           [1] class SimpleProject.Point V_1)
  IL_0000:  nop
  IL_0001:  ldarg.0
```

Fig. 2.3. Metadata of increment overloaded operation

If we open CIL-code of `Main()` function, it is possible to see that ++p call corresponds to the call, specified at the figure 2.4.

```
.method private hidebysig static void  Main() cil managed
{
  .entrypoint
  // Размер кода:       111 (0x6f)
  .maxstack  3
  .locals init ([0] class SimpleProject.Point p,
           [1] class SimpleProject.Point p1)
  IL_0000:  nop
  IL_0001:  newobj     instance void SimpleProject.Point::.ctor()
  IL_0006:  dup
  IL_0007:  ldc.i4.s   10
  IL_0009:  callvirt   instance void SimpleProject.Point::set_X(int32)
  IL_000e:  nop
  IL_000f:  dup
  IL_0010:  ldc.i4.s   10
  IL_0012:  callvirt   instance void SimpleProject.Point::set_Y(int32)
  IL_0017:  nop
  IL_0018:  stloc.0
  IL_0019:  ldloc.0
  IL_001a:  call       class SimpleProject.Point SimpleProject.Point::op_Increment(class SimpleProject.Point)
  IL_001f:  dup
  IL_0020:  stloc.0
  IL_0021:  call       void [mscorlib]System.Console::WriteLine(object)
```

Figure 2.4. CIL-code of Main() function

If while code compilation a program text contains ++ operator, the compiler defines whether `op_Increment` method exists for this type and if it is so, it generates a code calling this method otherwise generates an exception case.

3. Overloading of binary operators

As it was noted before, overloaded operators are statistic methods, for that reason binary operators should be assigned with 2 parameters.

To review an example of binary operation overloading, let's refresh some knowledge gained in school — vectors.

So, vector is a directed segment having 2 points: initial point and terminal point. In order to get vector coordinates, it is necessary to subtract terminal point coordinates from initial point ones. In order to sum two vectors, corresponding coordinates are to be summed, difference is defined in the same way. In order to multiply vector by numeric, each vector coordinate is to be multiplied by this numeric.

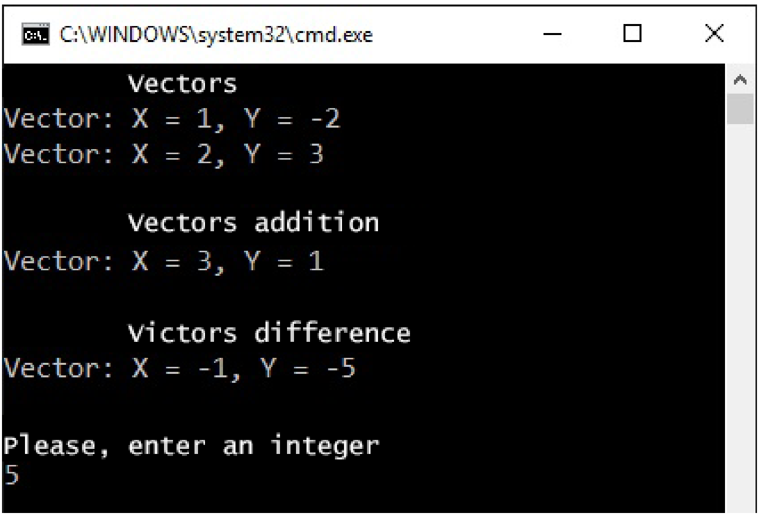
Let's create `Vector` class, making use of previously developed `Point` class.

```
using static System.Console;

namespace SimpleProject
{
    class Point
    {
        public int X { get; set; }
        public int Y { get; set; }
    }
    class Vector
    {
        public int X { get; set; }
```

```
        public int Y { get; set; }
        public Vector() { }
        public Vector(Point begin, Point end)
        {
            X = end.X - begin.X;
            Y = end.Y - begin.Y;
        }

        public static Vector operator +(Vector v1,
                                        Vector v2)
        {
            return new Vector { X = v1.X + v2.X,
                                Y = v1.Y + v2.Y };
        }

        public static Vector operator -(Vector v1,
                                        Vector v2)
        {
            return new Vector { X = v1.X - v2.X,
                                Y = v1.Y - v2.Y };
        }

        public static Vector operator *(Vector v, int n)
        {
            v.X *= n;
            v.Y *= n;
            return v;
        }

        public override string ToString()
        {
            return $"Vector: X = {X}, Y = {Y}";
        }
    }
    class Program
    {
        static void Main()
        {
```

```
            Point p1 = new Point { X = 2, Y = 3 };
            Point p2 = new Point { X = 3, Y = 1 };
            Vector v1 = new Vector(p1, p2);
            Vector v2 = new Vector { X = 2, Y = 3 };

            WriteLine($"\tVectors\n{v1}\n{v2}");
            WriteLine($"\n\tVectors summation\n{v1 +
                     v2}\n"); // x=3, y=1
            WriteLine($"\tVectors difference\n{v1 -
                     v2}\n"); // x=-1, y=-5
            WriteLine("Please enter an integer");
            int n = int.Parse(ReadLine());
            v1 *= n;
            WriteLine($"\n\tVector multiplication
                     by numeric {n}\n{v1}\n");
        }
    }
}
```

Probable program outcome.



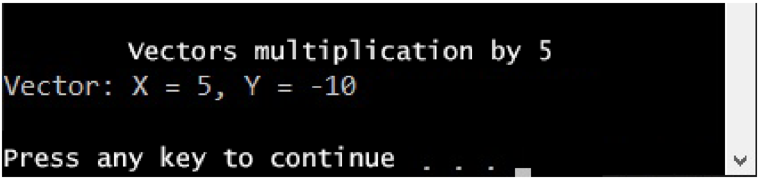


Fig. 3.1. Overloading of binary operators

Operators +=, *=, -= are overloaded automatically after overloading of corresponding binary operators, that's why application of *= operation will not lead to code error occurrence, because overloaded * operator will be used.

However, operators, overloaded as shown in the example will be used by compiler only if `Vector` type variable is located to the left of operand sign. It means that v1*10 formula will be compiled in a normal way, while multipliers relocation in 10*v1 formula will lead to error occurrence at compiling phase (Fig. 3.2).

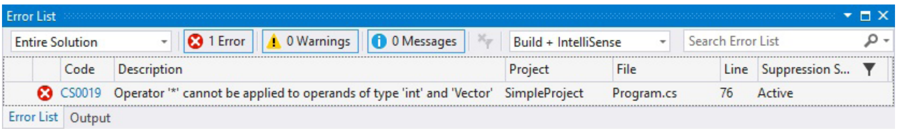


Fig. 3.2. Error: Operator * cannot be applied with these types

In order to debug this error, operator * should be overloaded, using another operand sequence:

```
public static Vector operator *(int n, Vector v)
{
    return v * n;
}
```

This overloaded version refers to operand `operator *` calling (`Vector` v, `int` n)

4. Overloading of relational operators

Comparison/relational operations are overloaded in pairs: if operation == overloads, so should operation!= do. There are the following relational operator pairs:

- == and !=
- < and >
- <= and >=.

While relational operators overloading it should be noted that there are two ways of equality verification:

- Links equality (identity);
- Values equality.

`Object` class defines following objects comparison methods:

- `public static bool ReferenceEquals(Object obj1, Object obj2)`
- `public bool virtual Equals(Object obj)`

These methods functioning differs depending on either value or reference type.

`ReferenceEquals()` method verifies whether two references point at one and the same class instance; or whether two references contain the same memory address, to be specific. This method cannot be overridden. In relation to value types, `ReferenceEquals()` always returns `false`, as far as while comparing casting to Object and packing is being done and then packed objects are distributed in address-wise manner.

Equals() method is virtual one. Its implementation in Object verifies equity of references, i.e. operates in the same manner as ReferenceEquals(). In relation to value types in basic System.ValueType, Equals() method is overridden and objects are compared via all fields comparison (bitwise comparison).

Please see below the example of ReferenceEquals() and Equals() methods application with both reference and value types:

```
using static System.Console;

namespace SimpleProject
{

    class CPoint
    {
        public int X { get; set; }
        public int Y { get; set; }
    }

    struct SPoint
    {
        public int X { get; set; }
        public int Y { get; set; }
    }

    class Program
    {

        static void Main()
        {
            //ReferenceEquals method functioning with
            //reference and value types

            //reference type
            CPoint cp = new CPoint { X = 10, Y = 10 };
```

```
CPoint cp1 = new CPoint { X = 10, Y = 10 };
CPoint cp2 = cp1;

//although p and p1 have the same values,
//they refer to various memory address
WriteLine($"ReferenceEquals(cp, cp1) =
{ReferenceEquals(cp, cp1)}"); // false

//p1 and p2 point at one and the same
//memory address
WriteLine($"ReferenceEquals(cp1, cp2) =
{ReferenceEquals(cp1, cp2)}"); // true

//value type
SPoint sp = new SPoint { X = 10, Y = 10 };

//packing takes place while data transfer
//into ReferenceEquals method,
//packed objects are distributed
//in addresswise manner

WriteLine($"ReferenceEquals(sp, sp) =
{ReferenceEquals(sp, sp)}"); // false

//Equals method functioning with
//reference and value types
//reference types addresses are being
//compared

WriteLine($"Equals(cp, cp1) =
          {Equals(cp, cp1)}"); // false

//value type
SPoint sp1 = new SPoint { X = 10, Y = 10 };
```

```
            //field values are being compared
            WriteLine($"Equals(sp, spl) =
            {Equals(sp, spl)}"); //true
        }
    }
}
```

Program outcome.

```
C:\WINDOWS\system32\cmd.exe
ReferenceEquals(cp, cp1) = False
ReferenceEquals(cp1, cp2) = True
ReferenceEquals(sp, sp) = False
Equals(cp, cp1) = False
Equals(sp, sp1) = True
Press any key to continue . . .
```

Fig. 4.1. Application of ReferenceEquals() and Equals() methods

Operator == is, as a rule, overloaded with `Equals()` method call, for that reason `Equals()` method is to be overridden in its own type.

As far as in `System.ValueType` the `Equals()` method deals with bitwise comparison, it means that it is possible not to override it in its own value types. However, in `System.ValueType`, field values obtaining for comparison in `Equals()` method is done with the help of reflexion, resulting in efficiency drop. For Thus, in order to speed up, it is recommended to override `Equals()` methodwhile developing value type .

While `Equals()` method overriding, `GetHashCode()` method should be overridden as well. This method is developed to get an integral value of object hash code, meanwhile different hash codes should correspond to different objects. If it is not possible to overload `GetHashCode()` method, the compiler will show the following warning message (Fig. 4.2).

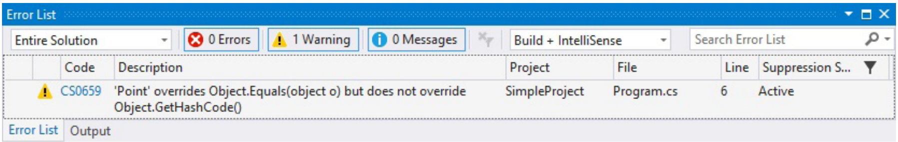


Fig. 4.2. Warning: overriding for GetHashCode() method is not available

While operator `!=` overloading, we used logical negation (!) and overloaded == operators. To compare two points during operators < and > overloading, use was made of the distance between set-up point and point with coordinates (0, 0) that is calculated using Pythagorean theorem.

Please, see below the example of relational operators overload for `Point` class (Fig. 4.3).

```
using System;
using static System.Console;

namespace SimpleProject
{
    class Point
    {
        public int X { get; set; }
        public int Y { get; set; }

        //Equals method overriding
        public override bool Equals(object obj)
        {
            return this.ToString() == obj.ToString();
        }
```

```
        //GetHashCode method should be overridden
        //as well
        public override int GetHashCode()
        {
            return this.ToString().GetHashCode();
        }

        public static bool operator ==(Point p1, Point p2)
        {
            return p1.Equals(p2);
        }

        public static bool operator !=(Point p1, Point p2)
        {
            return !(p1 == p2);
        }

        public static bool operator >(Point p1, Point p2)
        {
            return Math.Sqrt(p1.X * p1.X + p1.Y * p1.Y) >
                   Math.Sqrt(p2.X * p2.X + p2.Y * p2.Y);
        }

        public static bool operator <(Point p1, Point p2)
        {
            return Math.Sqrt(p1.X * p1.X + p1.Y * p1.Y) <
                  Math.Sqrt(p2.X * p2.X + p2.Y * p2.Y);
        }

        public override string ToString()
        {
            return $"Point: X = {X}, Y = {Y}.";
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
```

```
            Point point1 = new Point { X = 10, Y = 10 };
            Point point2 = new Point { X = 20, Y = 20 };

            WriteLine($"point1: {point1}");
            WriteLine($"point2: {point2}\n");

            WriteLine($"point1 == point2:
                {point1 == point2}"); // false
            WriteLine($"point1 != point2: {point1 !=
                 point2}\n"); // true

            WriteLine($"point1 > point2: {point1 >
                 point2}"); // false
            WriteLine($"point1 < point2: {point1 <
                 point2}"); // true
        }
    }
}
```

Program outcome.

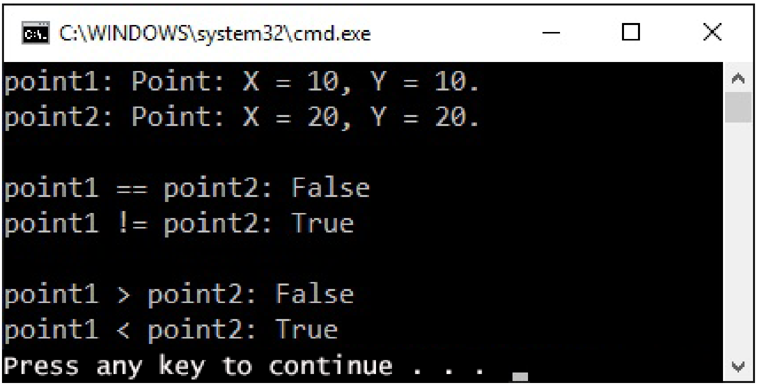


Fig. 4.3. Overloading of relational operators

5. Overloading of true and false operators

While overloading `true` and `false` operators, a developer sets up truth criterion for his/her data type. As soon as it is done, objects can be directly used as conditional statements in operator structure like `if`, `do`, `while`, `for`.

Overloading is done with application of the following rules:

- `true` operator should return `true` value, if objects state is true, otherwise it should be `false`;
- `false` operator should return `true` value, if objects state is false, otherwise it should be `false`;
- `true` and `false` operators should be overloaded in pair.

As truth criterion we have selected equity of all coordinates of a certain point to zero (Fig. 5.1).

```
using static System.Console;

namespace SimpleProject
{

    class Point
    {

        public int X { get; set; }
        public int Y { get; set; }

        public static bool operator true(Point p)
        {
            return p.X != 0 || p.Y != 0 ? true : false;
        }
```

```
        public static bool operator false(Point p)
        {
            return p.X == 0 && p.Y == 0 ? true : false;
        }

        public override string ToString()
        {
            return $"Point: X = {X}, Y = {Y}.";
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            WriteLine("Please enter coordinates
                      of a point in a plane");

            Point point = new Point { X =
                              int.Parse(ReadLine()), Y =
                              int.Parse(ReadLine()) };

            if (point)
            {
                WriteLine("Point is not
                          at coordinate origin");
            }
            else
            {
                WriteLine("Point is at coordinates
                          origin");
            }
        }
    }
}
```

Probable program outcome.

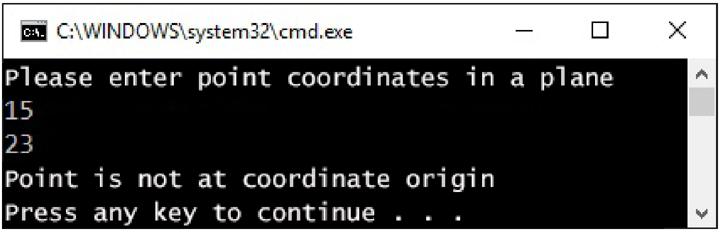


Fig. 5.1. Overload of true and false operators

6. Overloading of logical operators

Logical operators `&&` and `||` cannot be overloaded but they are simulated with the help of overloadable `&` and `|` operators.

In order to enable it the following requirements are to be met:

- `true` and `false` operators are to be overloaded in class;
- logical operators `&` and `|` are to be overloaded in class;
- operators `&` and `|` overload methodology should return type of class where overload takes place;
- references to class containing overload should be parameters in operators `&` and `|` overload methods.

Let's sum up all aforesaid in the following example (Fig. 6.1).

```
using static System.Console;

namespace SimpleProject
{
    class Point
    {
        public int X { get; set; }
        public int Y { get; set; }

        public static bool operator true(Point p)
        {
            return p.X != 0 || p.Y != 0 ? true : false;
        }
```

```
        public static bool operator false(Point p)
        {
            return p.X == 0 && p.Y == 0 ? true : false;
        }

        //logical operator | is overloaded
        public static Point operator |(Point p1,
                                       Point p2)
        {
            if ((p1.X != 0 || p1.Y != 0) || (p2.X != 0
                    || p2.Y != 0))
                return p2;

            return new Point();
        }

        //logical operator & is overloaded
        public static Point operator &(Point p1,
                                       Point p2)
        {
            if ((p1.X != 0 && p1.Y != 0) && (p2.X != 0
                    && p2.Y != 0))
                return p2;

            return new Point();
        }

        public override string ToString()
        {
            return $"Point: X = {X}, Y = {Y}.";
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
```

```
            Point point1 = new Point { X = 10, Y = 10 };
            Point point2 = new Point { X = 0, Y = 0 };

            WriteLine($"point1: {point1}");
            WriteLine($"point2: {point2}\n");
            Write("point1 && point2: ");

            if (point1 && point2)
            {
                WriteLine("true");
            }
            else
            {
                WriteLine("false");
            }

            Write("point1 || point2: ");

            if (point1 || point2)
            {
                WriteLine("true");
            }
            else
            {
                WriteLine("false");
            }
        }
    }
}
```

Program outcome.

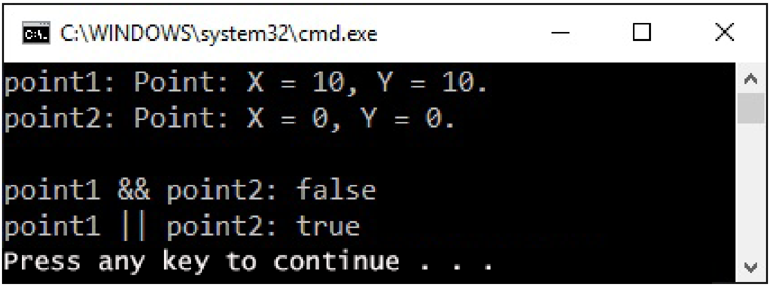


Fig. 6.1. Overloading of && and || operators

Operators `&&` and `||` are executed in the following manner.

First operand of `&&` operator is verified with the use of overloaded `false` operator and if the outcome is equal to `false` then further operands comparison is done with the use of overloaded operator &, this comparison is verified with overloaded `true` operator call as far as conditional operator is used. If the outcome of `false` operator for the first operand is equal to `true`, then & operator will not be executed, and the first operand will be considered as a parameter for `true` operator.

In relation to `||` operator the first operand is verified with overloaded operator `true`, if the result is equal to `false`, then further operands comparison will be done using overloaded | operator and the result of this comparison will be also checked by calling the overloaded operator `true` (conditional operator). If the outcome of `true` operator for the first operand is equal to true, then operator | will not be executed, and the first operand will be considered as a parameter for `true` operator.

Sequence of operations described above complies with functioning of short logical operators `&&` and `||` in C#.

7. Overloading of conversion operators

In native types there are operators to be further used for casting purposes.

Casting can be of 2 types:

- from arbitrary type to native one;
- from native type to arbitrary one.

Reference and value types casting is done in the same way.

As you know, casting can be done in explicit and implicit ways. Explicit type casting is required if there is a possibility of data loss during casting. For example:

- while converting `int` into `short`, because `short` size is not sufficient enough to store `int` values;
- while converting signed data types into unsigned, false result can be obtained, if signed variable contains negative value;
- while converting types with floating point into integers due to fractional part loss;
- while converting a type allowing `null`-value into a type not allowing `null`, if initial variable contains `null`, exception is generated.

If there is no data loss, observed as a result of casting, it can be executed as an implicit one.

Casting operation should be marked either as `implicit` one or as `explicit` one in order to define its further usage:

- `implicit` sets up an implicit conversion and can be used if a conversion is always safe, irrespectively of a variable value being converted;
- `explicit` sets up an explicit conversion and should be used if there is a possibility of data loss or exclusion occurrence.

Conversion operator declaration in class:

```
public static {implicit|explicit}
                    operator target_type (initial_type)
{
    //code
}
```

It is possible to cast elements of different native structures or classes. But the following limitations are to be considered:

- it is not possible to define an interclass casting if one of them is another one's descendant;

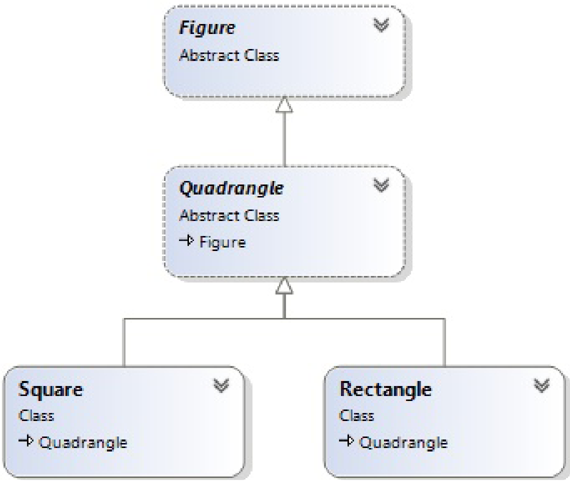


Fig. 7.1. Hierarchy of classes

- casting can be defined only in one of the types: either in initial type or in target one.

For example, there is the following class hierarchy (Fig. 7.1). The only possible type casting is the one between `Square` and `Rectangle` classes because there classes do not inherit each other. Meanwhile it should be considered that if conversion operator is defined inside a single class the same operator cannot be defined inside another class.

Let's review an example of applying the conversion operators with the use of class hierarchy, depicted in fig. 7.1. Let's overload an `implicit` operator from `Square` class into `Rectangle` class without data loss — rectangle width and height are to be got based on a square side. Let's also overload an `explicit` operator from class `Rectangle` into `Square` class with data loss — a square side is equal to rectangle height, width is not considered. Let's define an explicit and implicit conversion in relation to integer type for class `Square`.

In order to save space, `Draw()` method calls are commented (Fig. 7.2).

```
using static System.Console;

namespace SimpleProject
{
    abstract class Figure
    {
        public abstract void Draw();
    }

    abstract class Quadrangle : Figure { }

    class Rectangle : Quadrangle
    {
```

```
        public int Width { get; set; }
        public int Height { get; set; }

       public static implicit operator Rectangle(Square s)
        {
            return new Rectangle { Width = s.Length * 2,
                                   Height = s.Length };
        }

        public override void Draw()
        {
            for (int i = 0; i < Height; i++, WriteLine())
            {
                for (int j = 0; j < Width; j++)
                {
                    Write("*");
                }
            }
            WriteLine();
        }
        public override string ToString()
        {
            return $"Rectangle: Width = {Width},
                                  Height = {Height}";
        }
    }

    class Square : Quadrangle
    {
        public int Length { get; set; }
        public static explicit operator
                Square(Rectangle rect)
        {
            return new Square { Length = rect.Height };
        }

        public static explicit operator int(Square s)
        {
```

```
            return s.Length;
        }

        public static implicit operator Square(int number)
        {
            return new Square { Length = number };
        }

        public override void Draw()
        {
            for (int i = 0; i < Length; i++, WriteLine())
            {
                for (int j = 0; j < Length; j++)
                {
                    Write("*");
                }
            }
            WriteLine();
        }
        public override string ToString()
        {
            return $"Square: Length = {Length}";
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Rectangle rectangle = new Rectangle {
                          Width = 5, Height = 10 };
            Square square = new Square { Length = 7 };

            Rectangle rectSquare = square;
            WriteLine($"Implicit {square}
                          conversion  to rectangle.\n
                          {rectSquare}\n");
            //rectSquare.Draw();
```

```
            Square squareRect = (Square)rectangle;
            WriteLine($"Explicit {rectangle}
                      conversion to square.
                      \n {squareRect}\n");
            //squareRect.Draw();

            WriteLine("Please enter an integrer.");
            int number = int.Parse(ReadLine());
            Square squareInt = number;
            WriteLine($"Implicit integer
                      ({number})conversion to square.
                      \n{squareInt}\n");
            //squareInt.Draw();

            number = (int)square;
            WriteLine($"Explicit {square} conversion
                      to integer.\n{number}");
        }
    }
}
```

Possible program outcome.

```
C:\WINDOWS\system32\cmd.exe
Implicit square conversion (Square: Length=7) to Rectangle
Rectangle: Width = 14, Height = 7
Explicit rectangle conversion (Rectangle: Width = 5, Height = 10) to square
Square: Length = 10
Please enter an integer.
8
Implicit integer (8) conversion to square.
Square: Length = 8
Explicit square conversion (Square: Length = 7) to integer.
7
Press any key to continue . . .
```

Fig. 7.2. Overloading of conversion operators

8. Indexers

The concept of indexer

There is another interesting tool of C# language, being both a tool of operator [] overloading (but without involvement of the key word `operator`) and a kind of property at the same time (or in other words, parameters-containing property). Indexers are used in order to facilitate handling of special classes that implement user collection via array indexing syntax application.

Indexer declaration resembles a property but the difference is that indexers are of anonymous type (reference `this` is used instead of a name) and contain indexing parameters.

Indexer declaration syntax is as follows:

```
data_type this[argument_type] {get; set;}
```

Data_type is a type of collection objects, where `this` is a reference to an object wherein an indexer is declared. The fact that `this` containing syntax is used for indexers highlights that they can be applied at instance level only. `Argument_type` is an object index in a collection, being not necessarily an integer but of any type. Each indexer should have one parameter at least, but their number can be more (multidimensional indexers).

Creation of one-dimensional indexers

Let's review at the example the process of indexer creation and application. Suppose there is some kind of shop (`Shop` class), dealing with laptop sales (`Laptop` class). In order not to congest the example with extra data, let's assign only two properties to Laptop class: Vendor — manufacturer's name and Price — laptop price. Let's also override ToString() to view information as per certain product item. Reference to array of Laptop objects is the only Shop class field. Single-parameter builder is used to set up number of array elements and allocate memory for their storing. After that we have to enable access to the array elements via Shop class instance, using array syntax in such a way as if Shop class is Laptop type elements array. For that purpose we add an indexer to Shop class.

```
public Laptop this[int index]
{
    get
    {
        if (index >= 0 && index < laptopArr.Length)
        {
            return laptopArr[index];
        }
        throw new IndexOutOfRangeException();
        }
    set
    {
        laptopArr[index] = value;
    }
}
```

Here, in accessor get, index location within the array is verified and in case of attempt to transfer the index being outside the array to the indexer, then exception case IndexOutOfRangeException will be generated (exception cases will be studied in the next lesson).

There is another peculiarity of this program — Length property in Shop class that as soon as added enables getting size of array `laptopArr` of `Shop` class just like `Length` property of standard array.

```
    public int Length
    {
        get { return laptopArr.Length; }
    }
```

Let's review a program code as a whole.

```
using System;
using static System.Console;

namespace SimpleProject
{
    public class Laptop
    {
        public string Vendor { get; set; }
        public double Price { get; set; }

        public override string ToString()
        {
            return $"{Vendor} {Price}";
        }
    }

    public class Shop
    {
        Laptop[] laptopArr;

        public Shop(int size)
        {
            laptopArr = new Laptop[size];
        }
        public int Length
        {
            get { return laptopArr.Length; }
        }
```

```
        public Laptop this[int index]
        {
            get
            {
                if (index >= 0 && index <
                                laptopArr.Length)
                {
                    return laptopArr[index];
                }
                throw new IndexOutOfRangeException();
            }
            set
            {
                laptopArr[index] = value;
            }
        }
    }

    public class Program
    {

        public static void Main()
        {
            Shop laptops = new Shop(3);
            laptops[0] = new Laptop { Vendor =
                         "Samsung", Price = 5200 };
            laptops[1] = new Laptop { Vendor =
                         "Asus", Price = 4700 };
            laptops[2] = new Laptop { Vendor = "LG",
                                      Price = 4300 };

            try
            {
                for (int i = 0; i < laptops.Length; i++)
                {
                    WriteLine(laptops[i]);
                }
            }
```

```
            catch (Exception ex)
            {
                WriteLine(ex.Message);
            }
        }
    }
}
```

Program outcome (Fig. 8.1).

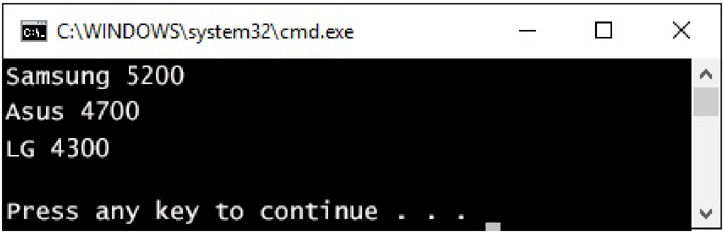


Fig. 8.1. One-dimensional indexers application

Creation of multidimensional indexers

C# enables creation of not only one-dimensional but multidimensional indexers. It is possible under condition that container class contains a multidimensional array as a field. As an example let's see review schematic application of two-dimensional indexer, not overloaded with extra checkups (Fig. 8.2).

```
using static System.Console;

namespace SimpleProject
{
    public class MultArray
    {
```

```
        private int[,] array;

        public int Rows { get; private set; }
        public int Cols { get; private set; }
        public MultArray(int rows, int cols)
        {
            Rows = rows;
            Cols = cols;
            array = new int[rows, cols];
        }

        public int this[int r, int c]
        {
            get { return array[r, c]; }
            set { array[r, c] = value; }
        }
    }

    public class Program
    {
        static void Main()
        {
            MultArray multArray = new MultArray(2, 3);
            for (int i = 0; i < multArray.Rows; i++)
            {
                for (int j = 0; j < multArray.Cols; j++)
                {
                    multArray[i, j] = i + j;
                    Write($"{multArray[i, j]} ");
                }
                WriteLine();
            }
        }
    }
}
```

Program outcome.

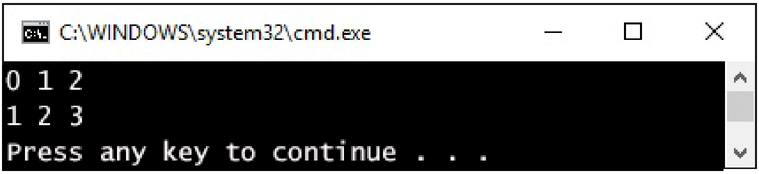


Fig. 8.2. Application of two-dimensional indexer

Indexer overloading

As it was noted before, type can support different indexer overloads under condition they differ in signatures. It means that a type of indexer parameter may be not only integral but any value. We for example one may add an indexer to our class to search by vendor name. For that we have created Vendors list and when entering a vendor matches will be searched among values of this list. We have also added an indexer to our class in order to enable price search. To search for element index by specified price extra method FindByPrice() was created. In accessors set, our code does not react to wrong value entering it is just ignored.

Now we have three indexer overloads, not clashing with each other, as far as their signatures do not coincide in terms of data type. Let's see what we came up with at the end (Fig. 8.3).

```
using System;
using static System.Console;

namespace SimpleProject
{
    public class Laptop
    {
```

```
        public string Vendor { get; set; }

        public double Price { get; set; }

        public override string ToString()
        {
            return $"{Vendor} {Price}";
        }
    }

    enum Vendors { Samsung, Asus, LG };

    public class Shop
    {
        private Laptop[] laptopArr;

        public Shop(int size)
        {
            laptopArr = new Laptop[size];
        }

        public int Length
        {
            get { return laptopArr.Length; }
        }

        public Laptop this[int index]
        {
            get
            {
                if (index >= 0 && index <
                    laptopArr.Length)
                {
                    return laptopArr[index];
                }
                throw new IndexOutOfRangeException();
            }
```

```
        set
        {
            laptopArr[index] = value;
        }
    }

    public Laptop this[string name]
    {
        get
        {
            if (Enum.IsDefined(typeof(Vendors), name))
            {
                return laptopArr[(int)Enum.
                Parse(typeof(Vendors), name)];
            }
            else
            {
                return new Laptop();
            }
        }

        set
        {
            if (Enum.IsDefined(typeof(Vendors), name))
            {
                laptopArr[(int)Enum.
                Parse(typeof(Vendors), name)] =
                value;
            }
        }
    }

    public int FindByPrice(double price)
    {
        for (int i = 0; i < laptopArr.Length; i++)
        {
```

```
                if (laptopArr[i].Price == price)
                {
                    return i;
                }
            }
            return -1;
        }

        public Laptop this[double price]
        {
            get
            {
                if (FindByPrice(price) >= 0)
                {
                    return this[FindByPrice(price)];
                }
                throw new Exception("Wrong price.");
            }
            set
            {
                if (FindByPrice(price) >= 0)
                {
                    this[FindByPrice(price)] = value;
                }
            }
        }
    }

    public class Program
    {
        public static void Main()
        {
            Shop laptops = new Shop(3);
            laptops[0] = new Laptop { Vendor = "Samsung",
                                  Price = 5200 };
            laptops[1] = new Laptop { Vendor = "Asus",
                                  Price = 4700 };
```

```
            laptops[2] = new Laptop { Vendor = "LG",
                               Price = 4300 };
            try
            {
                for (int i = 0; i < laptops.Length; i++)
                {
                    WriteLine(laptops[i]);
                }
                WriteLine();

                WriteLine($"Vendor Asus:
                          {laptops["Asus"]}.");

                WriteLine($"Vendor HP:
                          {laptops["HP"]}.");

                //ignore
                laptops["HP"] = new Laptop();

                WriteLine($"Price 4300:
                          {laptops[4300.0]}.");
                //wrong price
                WriteLine($"Price 4300:
                          {laptops[10500.0]}.");

                //ignore
                laptops[10500.0] = new Laptop();
            }

            catch (Exception ex)
            {
                WriteLine(ex.Message);
            }
        }
    }
}
```

Program outcome.

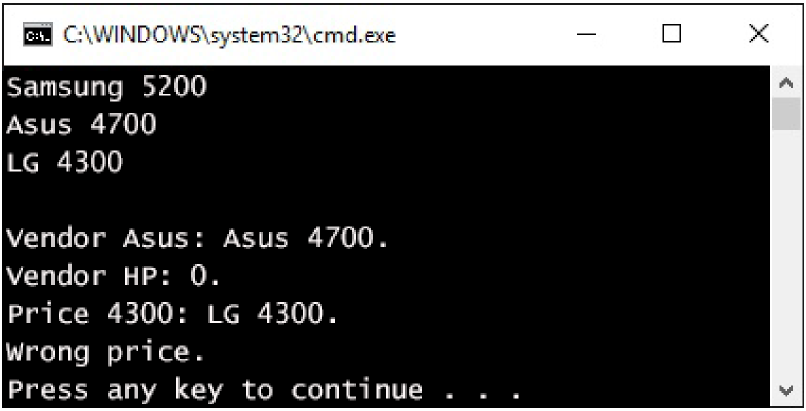


Fig. 8.3. Indexer overloading

Home task

1. Develop your own structural data type to store integer A and B coefficients of linear equation $A \times X + B \times Y = 0$. Apply static method Parse(), accepting the line with coefficients separated with coma or spacing.
2. Develop a method for resolving 2 linear equation system:
   $A1 \times X + B1 \times Y = 0$
   $A2 \times X + B2 \times Y = 0$
   Using output parameters the method should return computed solution or error if there is no solution.
3. Implement a class for complex number storing. Overload all necessary operators to compile the following code fragment successfully:

```
Complex z = new Complex(1,1);
Complex z1;
z1 = z - (z * z * z - 1) / (3 * z * z);
Console.WriteLine("z1 =  {0}", z1);
```

Summary on complex numbers (taken from Wikipedia):

- Any complex number can be represented as formal sum $x + iy$, where x and y are real numbers, i is an imaginary unit, i.e. a number satisfying the equation $i2 = -1$.

Operations on complex numbers:

- *Comparison*
  $a+bi = c+di$ means that $a=c$ and $b=d$ (two complex numbers are equal inbetween only when their real and imaginary

numbers are equal as well)

- *Addition*
  (a+bi) + (c+di) = (a+c) + (b+d)i
- *Subtraction*
  (a+bi) - (c+di) = (a-c) + (b-d)i
- *Multiplication*
  (a+bi) (c+di) = ac + bci + adi + bdi2= (ac-bd) + (bc +ad)i
- *Division*

$$\frac{(a+bi)}{(c+di)} = \left(\frac{ac+bd}{c^2+d^2}\right) + \left(\frac{bc-ad}{c^2+d^2}\right)i$$

4. Develop a Fraction class, represented with a common fraction. Two fields are to be provided in the class: fraction numerator and denominator. The following operators are to be overloaded: +,-,*,/,==,!=,<,>, true и false.
   Arithmetic and comparison are executed as per fraction rules. True operator returns true if the fraction is proper one (numerator is less than denominator), false operator returns true if the fraction is improper one (numerator is more than denominator).
   Overload the operators, needed for successful compilation of following code fragment:

```
Fraction f = new Fraction(3, 4);
int a = 10;
Fraction f1 = f * a;
Fraction f2 = a * f;
double d = 1.5;
Fraction f3 = f + d;
```

Lesson 5

Operators Overloading