

**top**

КОМПЬЮТЕРНАЯ  
АКАДЕМИЯ

# ПЛАТФОРМА MICROSOFT.NET И ЯЗЫК ПРОГРАММИРОВАНИЯ

# C#

# Урок №5

## Перегрузка операторов

### Содержание

1. Введение в перегрузку операторов .....	3
2. Перегрузка унарных операторов .....	6
3. Перегрузка бинарных операторов .....	11
4. Перегрузка операторов отношений .....	16
5. Перегрузка операторов true и false .....	23
6. Перегрузка логических операторов .....	26
7. Перегрузка операторов преобразования .....	31
8. Индексаторы .....	38
Понятие индексатора .....	38
Создание одномерных индексаторов .....	39
Создание многомерных индексаторов .....	42
Перегрузка индексаторов .....	44
Домашнее задание .....	50

# 1. Введение в перегрузку операторов

---

Перегрузка операторов представляет собой механизм определения стандартных операций для пользовательских типов. Для встроенных типов языка C# перегрузка стандартных операторов уже реализована, и ее изменить невозможно. Например, в классе `System.String` перегрузка операторов `==` и `!=` используется для проверки равенства и неравенства строк по их содержимому, а оператор `+` перегружен и выполняет конкатенацию строк. Хочется обратить Ваше внимание, что перегрузка операторов является одним из способов полиморфизма, например, применяя операцию `+` к числовым типам, мы получим значение их суммы, а применяя ее же со строками, получим конкатенацию строк.

Перегрузка операторов используется для улучшения читабельности программ и должна соответствовать определенным требованиям:

- перегрузка операторов должна выполняться открытыми статическими методами класса;
- у метода-оператора тип возвращаемого значения или одного из параметров должен совпадать с типом, в котором выполняется перегрузка оператора;
- параметры метода-оператора не должны включать модификатор `out` и `ref`.

Перегрузку операторов можно использовать как в классах, так и в структурах, при этом следует учитывать некоторые ограничения на перегрузку операторов:

- перегрузка не может изменить приоритет операторов;
- при перегрузке невозможно изменить число операндов, с которыми работает оператор;
- не все операторы можно перегружать.

Ниже представлены операторы, допускающие перегрузку:

Операторы	Категория операторов
-	Изменение знака переменной
!	Операция логического отрицания
~	Операция побитового дополнения, которая приводит к инверсии каждого бита
++, --	Инкремент и декремент
true, false	Критерий истинности объекта, определяется разработчиком класса
+, -, *, /, %	Арифметические операторы
&,  , ^, <<, >>	Битовые операции
==, !=, <, >, <=, >=	Операторы сравнения
&&,	Логические операторы
[]	Операции доступа к элементам массивов моделируются за счет индексаторов
()	Операции преобразования

Операторы, не допускающие перегрузку, приведены в следующей таблице:

Операторы	Категория операторов
<code>+=, -=, *=, /=, %=, &amp;=,  =, ^=, &lt;&lt;=, &gt;&gt;=</code>	Перегружаются автоматически при перегрузке соответствующих бинарных операций
<code>=</code>	Присвоение
<code>.</code>	Доступ к членам типа
<code>?:</code>	Оператора условия
<code>new</code>	Создание объекта
<code>as, is, typeof</code>	Используются для получения информации о типе
<code>→, sizeof, *, &amp;</code>	Доступны только в небезопасном коде

Синтаксис перегрузки выглядит следующим образом:

```
public static тип_возврата
                operator символ_операции (параметры)
{
    // код
}
```

## 2. Перегрузка унарных операторов

Поскольку перегруженные операторы являются статическими методами, они не получают указателя `this`, поэтому унарные операторы должны получать единственный операнд. Этот операнд должен иметь тип класса, в котором выполняется перегрузка оператора. То есть, если выполняется перегрузка унарного оператора в классе `Point`, то и тип операнда должен быть `Point`.

Рассмотрим перегрузку унарных операторов на примере операторов инкремента, декремента и изменения знака `-`. Для операторов `++` и `--` возвращаемое значение должно быть того же типа, в котором выполняется перегрузка оператора или производного от него.

Класс `Point` описывает точку на плоскости с координатами `x` и `y`. Оператор инкремента увеличивает обе координаты на 1, оператор декремента соответственно уменьшает на 1, оператор `-` — изменяет знак координат на противоположный (Рисунок 2.1).

```
using static System.Console;

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

```

//перегрузка инкремента
public static Point operator ++(Point s)
{
    s.X++;
    s.Y++;
    return s;
}

//перегрузка декремента
public static Point operator --(Point s)
{
    s.X--;
    s.Y--;
    return s;
}

//перегрузка оператора -
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($"Исходная точка\n{point}");

        WriteLine("Префиксная и постфиксная формы
                    инкремента выполняются одинаково");

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

```

```

        WriteLine($"Префиксная форма декремента\n
                    {--point}");

        WriteLine($"Выполнение оператора -\n
                    {-point}");

        WriteLine($"не изменило исходную точку\
                    n{point}");
    }
}

```

Результат работы программы.

```

C:\WINDOWS\system32\cmd.exe
Исходная точка
Point: X = 10, Y = 10
Префиксная и постфиксная формы инкремента выполняются одинаково
Point: X = 11, Y = 11
Point: X = 12, Y = 12
Префиксная форма декремента
Point: X = 11, Y = 11
Выполнение оператора -
Point: X = -11, Y = -11
не изменило исходную точку
Point: X = 11, Y = 11
Для продолжения нажмите любую клавишу . . .

```

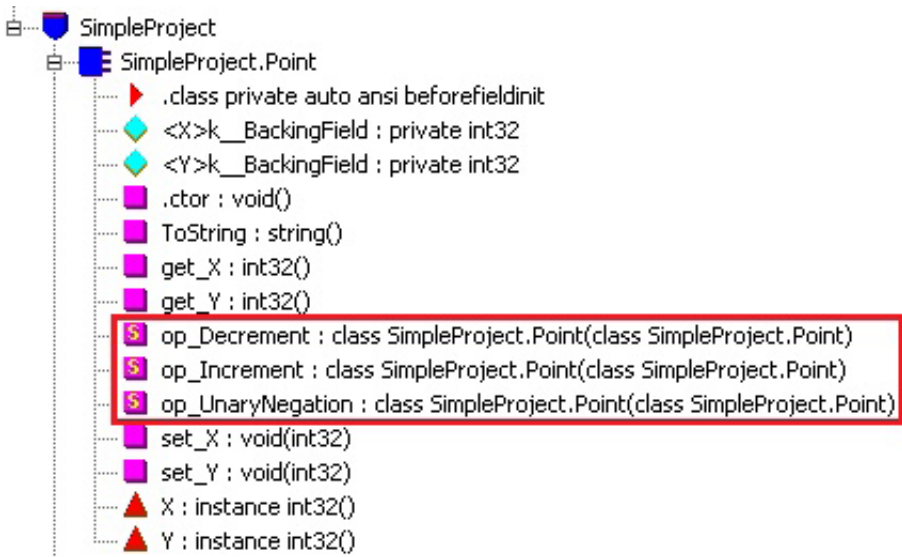
**Рисунок 2.1.** Перегрузка унарных операторов

В данном примере `Point` является ссылочным типом, поэтому изменения значений `x` и `y`, которые выполняются в перегруженных операторах инкремента и декремента, изменяют переданный в них объект. Оператор — (изменение знака) не должен изменять состояние переданного объекта, а должен возвращать новый объект с измененным знаком. Для этого в реализации этого метода создается новый объект `Point`, изменяется знак его координат и этот объект возвращается из метода.



Интересно отметить, что с С# нет возможности выполнить отдельно перегрузку постфиксной и префиксной форм операторов инкремента и декремента. Поэтому при вызове постфиксная и префиксная форма работают одинаково, как префиксная форма.

Перегрузка операторов выполняется путем создание специальных методов класса. Рассмотрим класс `Point` с помощью `ildasm` (Рисунок 2.2).



**Рисунок 2.2.** Информация о классе `Point` в дисассемблере

Легко установить следующее соответствие перегруженных операторов сгенерированным методам:

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

Из метаданных видно, что эти методы имеют флаг `specialname` (Рисунок 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
```

**Рисунок 2.3.** Метаданные перегруженной операции инкремента

Если открыть CIL-код функции `Main()`, можно увидеть, что вызову `++p` соответствует вызову на рисунке 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)
```

**Рисунок 2.4.** CIL-код функции `Main()`

Когда при компиляции кода в тексте программы встречается оператор `++`, компилятор определяет, существует ли метод `op_Increment` в данном типе, и если да — генерирует код, вызывающий этот метод, иначе генерирует исключительную ситуацию.

## 3. Перегрузка бинарных операторов

Как отмечалось ранее, перегруженные операторы являются статическими методами, поэтому бинарные операторы должны получать два параметра.

Для примера перегрузки бинарных операций освежим некоторые знания за 8 класс общеобразовательной школы — векторы.

Итак, вектор — направленный отрезок, имеющий начало и конец, то есть две точки. Для того чтобы получить координаты вектора необходимо из координат конечной точки вычесть соответствующие координаты начальной точки. Для того чтобы сложить два вектора нужно сложить их соответствующие координаты, разность — аналогично. Для того чтобы умножить вектор на число, необходимо каждую координату вектора умножить на это число.

Создадим класс `Vector`, используя разработанный ранее класс `Point`.

```
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($"{tВектора\n{v1}\n{v2}");

            WriteLine($"{n\tСложение векторов\n{v1 +
                v2}\n"); // x=3, y=1

            WriteLine($"{tРазность векторов\n{v1 -
                v2}\n"); // x=-1, y=-5

            WriteLine("Введите целое число");

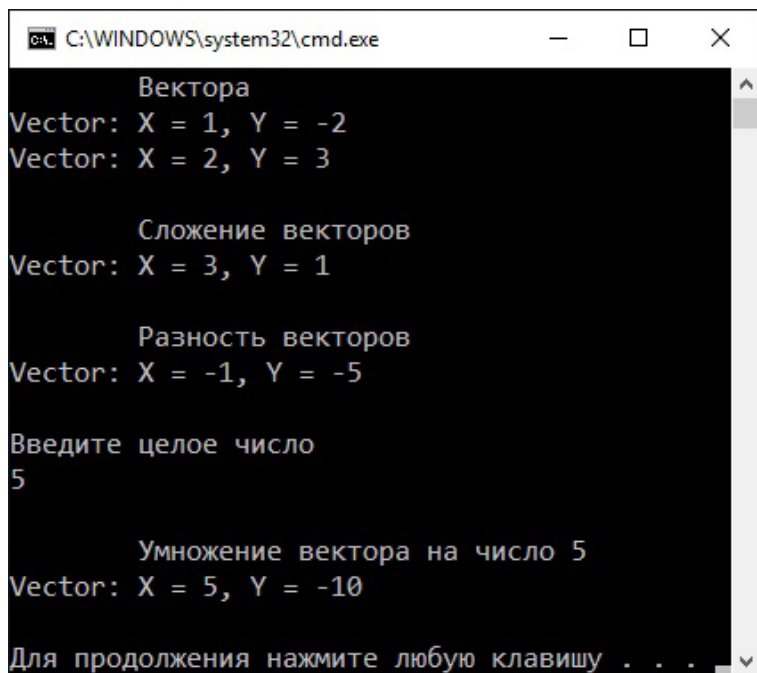
            int n = int.Parse(ReadLine());

            v1 *= n;

            WriteLine($"{n\tУмножение вектора на
                число {n}\n{v1}\n");
        }
    }
}

```

Возможный результат работы программы.



```
C:\WINDOWS\system32\cmd.exe

    Вектора
Vector: X = 1, Y = -2
Vector: X = 2, Y = 3

    Сложение векторов
Vector: X = 3, Y = 1

    Разность векторов
Vector: X = -1, Y = -5

Введите целое число
5

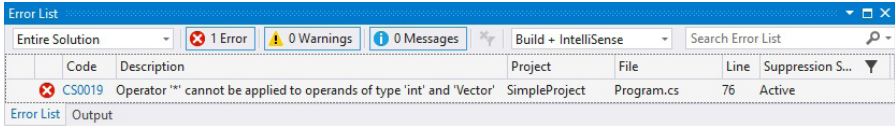
    Умножение вектора на число 5
Vector: X = 5, Y = -10

Для продолжения нажмите любую клавишу . . .
```

**Рисунок 3.1.** Перегрузка бинарных операторов

Перегрузка операторов `+=`, `*=`, `-=` выполняется автоматически после перегрузки соответствующих бинарных операторов, поэтому применение операции `*=` в коде ошибки не вызовет, а будет использован перегруженный оператор `*`.

Однако, перегруженные в примере, операторы будут использоваться компилятором, только если переменная типа `Vector` находится слева от знака операнда. То есть выражение `v1*10` откомпилируется нормально, а при перестановке сомножителей — в выражении `10*v1` произойдет ошибка на этапе компиляции (Рисунок 3.2).



**Рисунок 3.2.** Ошибка: нельзя применить оператор \* с данными типами

Для исправления этой ошибки следует перегрузить оператор \* с другим порядком операндов:

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

Эта перегруженная версия сводится к вызову оператора `operator*(Vector v, int n)`

## 4. Перегрузка операторов отношений

Операции сравнения перегружаются парами: если перегружается операция `==`, также должна перегружаться операция `!=`. Существуют следующие пары операторов сравнения:

- `==` и `!=`
- `<` и `>`
- `<=` и `>=`.

При перегрузке операторов отношения надо учитывать, что есть два способа проверки равенства:

- равенство ссылок (тождество);
- равенство значений.

В классе `Object` определены следующие методы сравнения объектов:

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

Есть отличия в работе этих методов со значимыми и ссылочными типами.

Метод `ReferenceEquals()` проверяет, указывают ли две ссылки на один и тот же экземпляр класса; точнее — содержат ли две ссылки одинаковый адрес памяти. Этот метод невозможно переопределить. Со значимыми типами `ReferenceEquals()` всегда возвращает `false`, т.к. при



сравнении выполняется приведение к `Object` и упаковка, упакованные объекты располагаются по разным адресам.

Метод `Equals()` является виртуальным. Его реализация в `Object` выполняет проверку равенства ссылок, т.е. работает, так же как и `ReferenceEquals()`. Для значимых типов в базовом типе `System.ValueType` выполнено переопределение метода `Equals()`, в котором выполняется сравнение объектов путем сравнения всех полей (побитовое сравнение).

Пример использования методов `ReferenceEquals()` и `Equals()` со ссылочными и значимыми типами:

```
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()
        {
            // работа метода ReferenceEquals со
            // ссылочными и значимыми типами

            //ссылочный тип
            CPoint cp = new CPoint { X = 10, Y = 10 };
        }
    }
}
```

```

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

// хотя p и p1 содержат одинаковые
// значения, они указывают на разные
// адреса памяти
WriteLine($"ReferenceEquals(cp, cp1) =
{ReferenceEquals(cp, cp1)}"); // false

// p1 и p2 указывают на один и тот же
// адрес памяти
WriteLine($"ReferenceEquals(cp1, cp2) =
{ReferenceEquals(cp1, cp2)}"); // true

// значимый тип
SPoint sp = new SPoint { X = 10, Y = 10 };

// при передаче в метод ReferenceEquals
// выполняется упаковка, упакованные
// объекты располагаются
// по разным адресам
WriteLine($"ReferenceEquals(sp, sp) =
{ReferenceEquals(sp, sp)}"); // false

// работа метода Equals со ссылочными
// и значимыми типами

// выполняется сравнение адресов
// ссылочных типов
WriteLine($"Equals(cp, cp1) =
{Equals(cp, cp1)}"); // false

// значимый тип
SPoint sp1 = new SPoint { X = 10, Y = 10 };

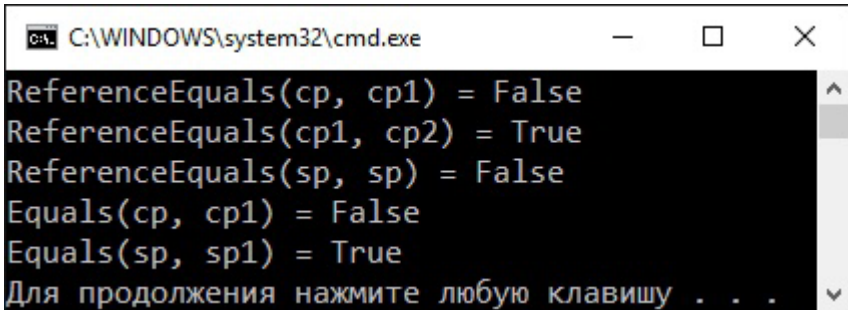
```

```

        // выполняется сравнение значений полей
        WriteLine($"Equals(sp, sp1) =
        {Equals(sp, sp1)}"); // true
    }
}

```

Результат работы программы.



```

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
Для продолжения нажмите любую клавишу . . .

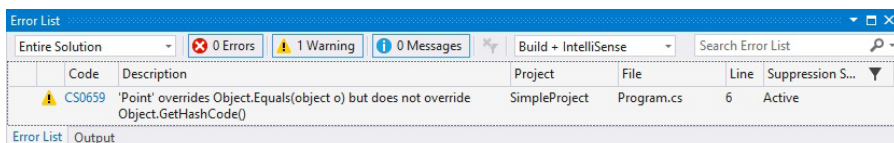
```

**Рисунок 4.1.** Использование методов `ReferenceEquals()` и `Equals()`

Перегрузка оператора `==` обычно выполняется путем вызова метода `Equals()`, поэтому необходимо переопределить метод `Equals()` в собственном типе.

Поскольку в `System.ValueType` метод `Equals()` выполняет побитовое сравнение, то в собственных значимых типах его можно и не переопределять. Однако, в `System.ValueType` получение значений полей для сравнения в методе `Equals()` выполняется с помощью рефлексии, что приводит к снижению производительности. Поэтому при разработке значимого типа для увеличения быстродействия все-таки рекомендуется выполнить переопределение метода `Equals()`.

При переопределении метода `Equals()` следует также переопределять и метод `GetHashCode()`. Этот метод предназначен для получения целочисленного значения хеш-кода объекта, при этом различным объектам должны соответствовать различные хеш-коды. Если перегрузку метода `GetHashCode()` не выполнить, то компилятор выдаст предупреждение (Рисунок 4.2).



**Рисунок 4.2.** Предупреждение: нет переопределения для метода `GetHashCode()`

При перегрузке оператора `!=` мы воспользовались оператором логического отрицания (`!`) и перегруженным оператором `==`. в качестве сравнения двух точек при перегрузке операторов `<i>` было взято расстояние между заданной точкой и точкой с координатами `(0, 0)`, которое вычисляется с использованием теоремы Пифагора.

Пример перегрузки операторов отношений для класса `Point` (Рисунок 4.3).

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

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

```

// переопределение метода Equals
public override bool Equals(object obj)
{
    return this.ToString() == obj.ToString();
}

// необходимо также переопределить метод
// GetHashCode
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 };

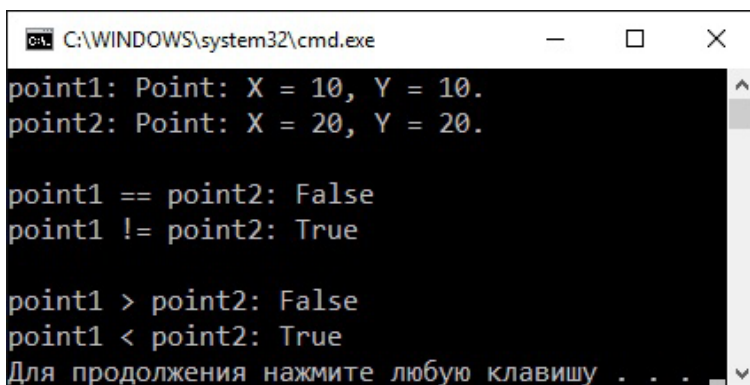
        WriteLine($"point1: {point1}");
        WriteLine($"point2: {point2}\n");

        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
    }
}

```

Результат работы программы.



```

C:\WINDOWS\system32\cmd.exe
point1: Point: X = 10, Y = 10.
point2: Point: X = 20, Y = 20.

point1 == point2: False
point1 != point2: True

point1 > point2: False
point1 < point2: True
Для продолжения нажмите любую клавишу . . .

```

**Рисунок 4.3.** Перегрузка операторов отношений

## 5. Перегрузка операторов true и false

При перегрузке операторов `true` и `false` разработчик задает критерий истинности для своего типа данных. После этого, объекты типа напрямую можно использовать в структуре операторов `if`, `do`, `while`, `for` в качестве условных выражений.

Перегрузка выполняется по следующим правилам:

- оператор `true` должен возвращать значение `true`, если состояние объекта истинно и `false` в противном случае;
- оператор `false` должен возвращать значение `true`, если состояние объекта ложно и `false` в противном случае;
- операторы `true` и `false` надо перегружать в паре.

В качестве критерия истинности мы взяли равенство всех координат конкретной точки нулевому значению (Рисунок 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;
}

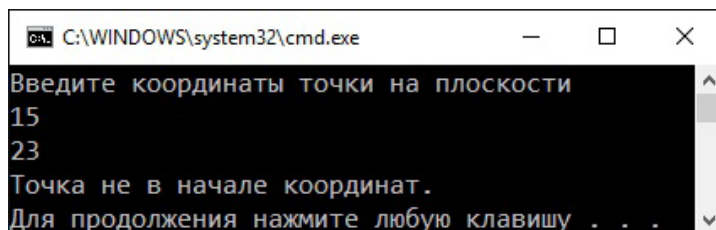
public override string ToString()
{
    return $"Point: X = {X}, Y = {Y}.";
}
}
class Program
{
    static void Main(string[] args)
    {
        WriteLine("Введите координаты точки  
на плоскости");

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

        if (point)
        {
            WriteLine("Точка не в начале  
координат.");
        }
        else
        {
            WriteLine("Точка в начале  
координат.");
        }
    }
}
```



Возможный результат работы программы.



A screenshot of a Windows command prompt window. The title bar shows the path 'C:\WINDOWS\system32\cmd.exe'. The window has standard minimize, maximize, and close buttons. The command prompt displays the following text: 'Введите координаты точки на плоскости' (Enter the coordinates of a point on the plane), followed by the input '15' on the next line and '23' on the line after. Then, it displays 'Точка не в начале координат.' (Point is not at the origin.) and 'Для продолжения нажмите любую клавишу . . .' (Press any key to continue . . .). A vertical scrollbar is visible on the right side of the text area.

**Рисунок 5.1.** Перегрузка операторов true и false

## 6. Перегрузка логических операторов

Логические операторы `&&` и `||` перегрузить нельзя, но они моделируются с помощью операторов `&` и `|`, допускающих перегрузку.

Для того чтобы это стало возможным, необходимо выполнить ряд требований:

- в классе должна быть выполнена перегрузка операторов `true` и `false`;
- в классе необходимо перегрузить логические операторы `&` и `|`;
- методы перегрузки операторов `&` и `|` должны возвращать тип класса, в котором осуществляется перегрузка;
- параметрами в методах перегрузки операторов `&` и `|` должны быть ссылки на класс, который содержит перегрузку.

Реализуем все вышеперечисленное в следующем примере (Рисунок 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;
}

// перегружаем логический оператор |
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();
}

// перегружаем логический оператор &
public static Point operator &(amp;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");

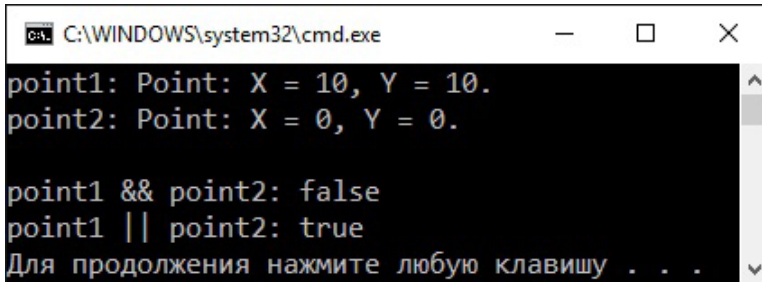
        Write("point1 && point2: ");

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

        Write("point1 || point2: ");

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

Результат работы программы.



```
C:\WINDOWS\system32\cmd.exe
point1: Point: X = 10, Y = 10.
point2: Point: X = 0, Y = 0.

point1 && point2: false
point1 || point2: true
Для продолжения нажмите любую клавишу . . .
```

**Рисунок 6.1.** Перегрузка операторов `&&` и `||`

Выполнение операторов `&&` и `||` происходит следующим образом.

Для оператора `&&` первый операнд проверяется с использованием перегруженного оператора `false`, если результат равен `false`, тогда дальнейшее сравнение операндов осуществляется с использованием перегруженного оператора `&`, результат этого сравнения проверяется вызовом перегруженного оператора `true`, так как используется условный оператор. Если результат оператора `false` для первого операнда будет равен `true`, тогда оператор `&` выполняться не будет, а параметром для оператора `true` будет являться первый операнд.

Для оператора `||` первый операнд проверяется с использованием перегруженного оператора `true`, если результат равен `false`, тогда дальнейшее сравнение операндов осуществляется с использованием перегруженного оператора `|`, результат этого сравнения также проверяется вызовом перегруженного оператора `true` (условный оператор). Если результат оператора `true` для первого

операнда будет равен `true`, тогда оператор `|` выполняться не будет, а параметром для оператора `true` будет являться первый операнд.

Описанные выше последовательности действий соответствуют работе укороченных логических операторов `&&` и `||` в языке C#.

## 7. Перегрузка операторов преобразования

---

В собственных типах можно определить операторы, которые будут использоваться для выполнения приведения.

Приведение может быть двух типов:

- из произвольного типа в собственный тип;
- из собственного типа в произвольный тип.

Для ссылочных и значимых типов приведение выполняется одинаково.

Как Вам известно, приведение может выполняться явным и неявным образом. Явное приведение типов требуется, если возможна потеря данных в результате приведения. Например:

- при преобразовании `int` в `short`, потому что размер `short` недостаточен для сохранения значения `int`;
- при преобразовании типов данных со знаком в беззнаковые может быть получен неверный результат, если переменная со знаком содержит отрицательное значение;
- при конвертировании типов с плавающей точкой в целые, так как дробная часть теряется;
- при конвертировании типа, допускающего `null`-значения, в тип, не допускающий `null`, если исходная переменная содержит `null`, генерируется исключение.

Если потери данных в результате приведения не происходит, приведение можно выполнять как неявное.

Операция приведения должна быть помечена либо как `implicit`, либо как `explicit`, чтобы указать, как ее предполагается использовать:

- `implicit` задает неявное преобразование, его можно использовать, если преобразование всегда безопасно независимо от значения переменной, которая преобразуется;
- `explicit` задает явное преобразование, его следует использовать, если возможна потеря данных или возникновение исключения.

Объявление оператора преобразования в классе:

```
public static {implicit|explicit}  
               operator целевой_тип (исходный_тип)  
{  
    // код  
}
```

Имеется возможность выполнять приведение между экземплярами разных собственных структур или классов. Однако при этом существуют следующие ограничения:

- нельзя определить приведение между классами, если один из них является наследником другого;
- приведение может быть определено только в одном из типов: либо в исходном типе, либо в типе назначения.

Например, имеется следующая иерархия классов (Рисунок 7.1):



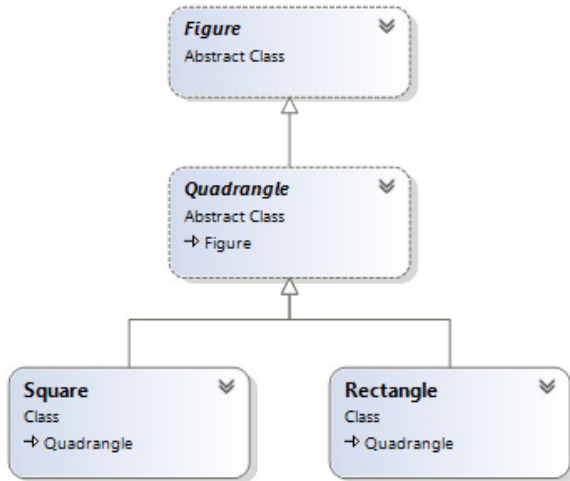


Рисунок 7.1. Иерархия классов

Единственное допустимое приведение типов — это приведения между классами `Square` и `Rectangle`, потому что эти классы не наследуют друг друга. При этом следует помнить, что если оператор преобразования определен внутри одного класса, то нельзя определить такой же оператор внутри другого класса.

Приведем пример использования операторов преобразования, взяв за основу иерархию классов на рисунке 7.1. Перегрузим, оператор неявного преобразования (`implicit`), из класса `Square` в класс `Rectangle` без потери данных — ширину и высоту прямоугольника получаем на основании стороны квадрата. Также перегрузим, оператор явного преобразования (`explicit`), из класса `Rectangle` в класс `Square` с потерей данных — сторона квадрата равна высоте прямоугольника, ширина не учитывается. Для класса `Square` определим явное и неявное преобразование к целому типу.

В целях экономии места вызовы метода Draw() закомментированы (Рисунок 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($"Неявное преобразование
                    квадрата ({square}) к
                    прямоугольнику.\n{rectSquare}\n");
        //rectSquare.Draw();

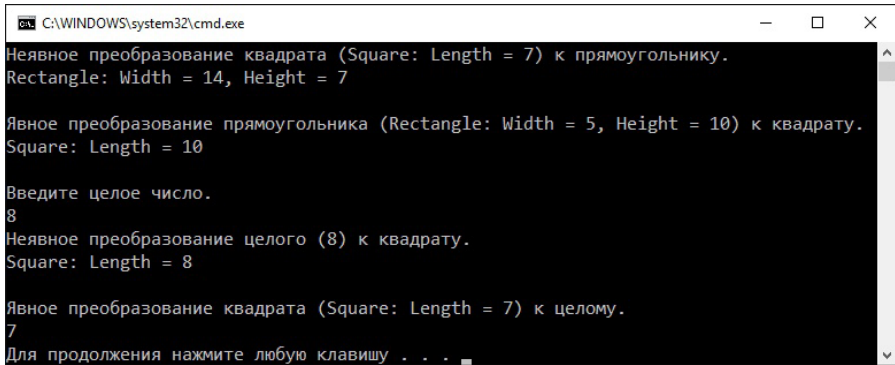
        Square squareRect = (Square)rectangle;
        WriteLine($"Явное преобразование
                    прямоугольника ({rectangle}) к
                    квадрату.\n{squareRect}\n");
        //squareRect.Draw();

        WriteLine("Введите целое число.");
        int number = int.Parse(ReadLine());
        Square squareInt = number;
        WriteLine($"Неявное преобразование целого
                    ({number}) к квадрату.\n{squareInt}\n");
        //squareInt.Draw();

        number = (int)square;
        WriteLine($"Явное преобразование квадрата
                    ({square}) к целому.\n{number}");
    }
}

```

Возможный результат работы программы.



```
C:\WINDOWS\system32\cmd.exe
Неявное преобразование квадрата (Square: Length = 7) к прямоугольнику.
Rectangle: Width = 14, Height = 7

Явное преобразование прямоугольника (Rectangle: Width = 5, Height = 10) к квадрату.
Square: Length = 10

Введите целое число.
8
Неявное преобразование целого (8) к квадрату.
Square: Length = 8

Явное преобразование квадрата (Square: Length = 7) к целому.
7
Для продолжения нажмите любую клавишу . . .
```

**Рисунок 7.2.** Перегрузка операторов преобразования

## 8. Индексаторы

### Понятие индексатора

А теперь ещё одно любопытное средство языка C#, представляющее собой одновременно способ перегрузки оператора `[]` (но без участия ключевого слова `operator`) и разновидность свойства (или как его ещё называют, свойством с параметрами). Индексаторы применяются для облегчения работы со специальными классами, реализующими пользовательскую коллекцию, используя синтаксис индексирования массива.

Объявление индексатора подобно свойству, но с той разницей, что индексаторы безымянные (вместо имени используется ссылка `this`) и что индексаторы включают параметры индексирования.

Синтаксис объявления индексатора следующий:

```
тип_данных this[тип_аргумента] {get; set;}
```

Тип\_данных — это тип объектов коллекции, `this` — это ссылка на объект, в котором определен индексатор. То, что для индексаторов используется синтаксис со ссылкой `this`, подчёркивает, что их можно использовать только на экземплярном уровне и никак иначе. Тип\_аргумента представляет индекс объекта в коллекции, причём этот индекс необязательно целочисленный, он может быть любого типа. У каждого индексатора должен быть минимум один параметр, но их может быть и больше (многомерные индексаторы).

## Создание одномерных индексаторов

Рассмотрим пример создания и применения индексатора. Предположим, есть некий магазин (класс `Shop`), занимающийся реализацией ноутбуков (класс `Laptop`). Дабы не перегружать пример лишней информацией, снабдим класс `Laptop` только двумя свойствами: `Vendor` — имя фирмы-производителя и `Price` — цена ноутбука. Также переопределим метод `ToString()` для отображения информации по конкретной единице товара. в качестве единственного поля класса `Shop` выступает ссылка на массив объектов `Laptop`. в конструкторе с одним параметром задаётся количество элементов массива и выделяется память для их хранения. Далее нам нужно сделать возможным обращение к элементам этого массива через экземпляр класса `Shop`, пользуясь синтаксисом массива так, словно класс `Shop` и есть массив элементов типа `Laptop`. Для этого мы добавляем в класс `Shop` индексатор.

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

Здесь в аксессоре `get` осуществляется проверка нахождения индекса в пределах массива и в случае если в индексатор попробуют передать индекс, который находится за пределами массива, тогда будет сгенерирована исключительная ситуация `IndexOutOfRangeException` (исключительные ситуации Вы изучите в последующем уроке).

Еще одна особенность данной программы — свойство `Length` в классе `Shop`, добавление которого позволяет получать размер массива `laptopArr` класса `Shop` подобно свойству `Length` стандартного массива.

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

Теперь рассмотрим код программы целиком.

```
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);
        }
    }
}

```

Результат работы программы (Рисунок 8.1).

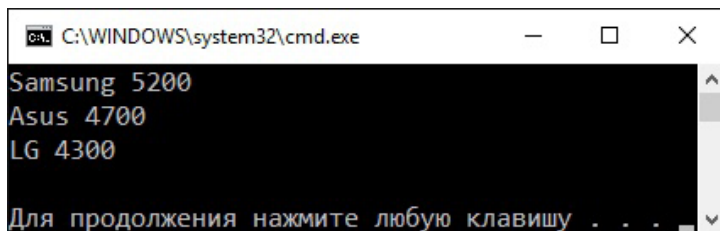


Рисунок 8.1. Использование одномерных индексаторов

## Создание многомерных индексаторов

В C# есть возможность создавать не только одномерные, но и многомерные индексаторы. Это возможно, если класс-контейнер содержит в качестве поля массив с более чем одним измерением. Для демонстрации такой

возможности приведём схематичный пример использования двумерного индексатора, не перегруженный дополнительными проверками (Рисунок 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; }
        }
    }

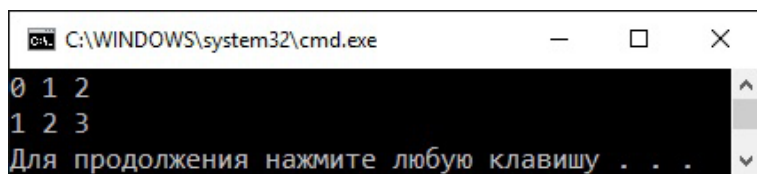
    public class Program
    {
        static void Main()
        {
            MultArray multArray = new MultArray(2, 3);
            for (int i = 0; i < multArray.Rows; i++)
            {
```

```

        for (int j = 0; j < multArray.Cols; j++)
        {
            multArray[i, j] = i + j;
            Write($"{multArray[i, j]} ");
        }
        WriteLine();
    }
}
}
}

```

Результат работы программы.



**Рисунок 8.2.** Использование двумерного индексатора

## Перегрузка индексаторов

Как было отмечено ранее, тип может поддерживать различные перегрузки индексаторов при условии, что они отличаются сигнатурой. Это значит, что тип параметра индексатора может быть не только целочисленным, но и вообще любым. Например, можно добавить в наш класс индексатор для поиска по имени производителя. Для этого мы создали перечисление [Vendors](#), при вводе производителя поиск совпадения будет осуществляться среди значений этого перечисления. Также мы добавили в наш класс индексатор для поиска по цене. Для поиска индекса элемента по заданной цене был создан

дополнительный метод `FindByPrice()`. в аксессорах `set` наш код никак не реагирует на неправильный ввод значений, он просто игнорируется.

Теперь у нас есть три перегрузки индексатора, но они между собой не конфликтуют, поскольку их сигнатуры не совпадают по типу данных. Вот что получилось в конечном счёте (Рисунок 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("Недопустимая
                                стоимость.");
        }
        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($"Производитель Asus:
                       {laptops["Asus"]}");

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

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

            WriteLine($"Стоимость 4300:
                       {laptops[4300.0]}");
        }
    }
}

```



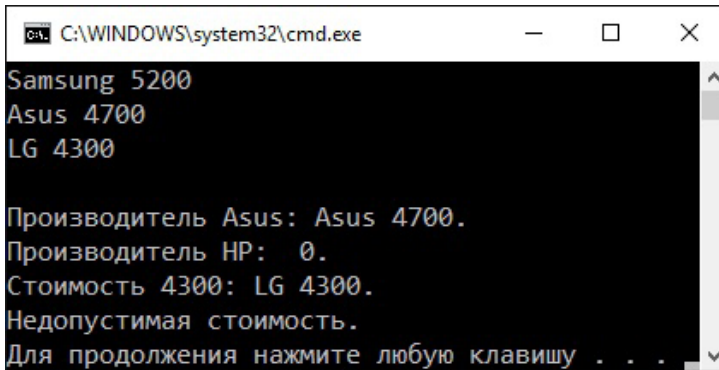
```

        // недопустимая стоимость
        WriteLine($"Стоимость 10500:
                    {laptops[10500.0]}.");

        // игнорирование
        laptops[10500.0] = new Laptop();
    }
    catch (Exception ex)
    {
        WriteLine(ex.Message);
    }
}
}
}

```

Результат работы программы.



```

C:\WINDOWS\system32\cmd.exe
Samsung 5200
Asus 4700
LG 4300

Производитель Asus: Asus 4700.
Производитель HP: 0.
Стоимость 4300: LG 4300.
Недопустимая стоимость.
Для продолжения нажмите любую клавишу . . .

```

Рисунок 8.3. Перегрузка индексаторов

# Домашнее задание

1. Разработать собственный структурный тип данных для хранения целочисленных коэффициентов  $A$  и  $B$  линейного уравнения  $A \times X + B \times Y = 0$ . в классе реализовать статический метод `Parse()`, которые принимает строку со значениями коэффициентов, разделенных запятой или пробелом.
2. Разработать метод для решения системы 2 линейных уравнений:  
 $A1 \times X + B1 \times Y = 0$   
 $A2 \times X + B2 \times Y = 0$   
Метод с помощью выходных параметров должен возвращать найденное решение или ошибку, если решения не существует.
3. Реализовать класс для хранения комплексного числа. Выполнить в нем перегрузку всех необходимых операторов для успешной компиляции следующего фрагмента кода:

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

*Краткая справка по комплексным числам  
(из Википедии):*

- Любое комплексное число может быть представлено как формальная сумма  $x + iy$ , где  $x$  и  $y$  — вещественные

числа,  $i$  — мнимая единица, то есть число, удовлетворяющее уравнению  $i^2 = -1$

### Действия над комплексными числами

- Сравнение

$a + bi = c + di$  означает, что  $a = c$  и  $b = d$  (два комплексных числа равны между собой тогда и только тогда, когда равны их действительные и мнимые части).

- Сложение  $(a + bi) + (c + di) = (a + c) + (b + d)i$

- Вычитание  $(a + bi) - (c + di) = (a - c) + (b - d)i$

- Умножение  $(a + bi)(c + di) = ac + bci + adi + bdi^2 = (ac - bd) + (bc + ad)i$

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

4. Разработать класс `Fraction`, представляющий простую дробь. в классе предусмотреть два поля: числитель и знаменатель дроби. Выполнить перегрузку следующих операторов: `+`, `-`, `*`, `/`, `=`, `!=`, `<`, `>`, `true` и `false`.

Арифметические действия и сравнение выполняется в соответствии с правилами работы с дробями. Оператор `true` возвращает `true` если дробь правильная (числитель меньше знаменателя), оператор `false` возвращает `true` если дробь неправильная (числитель больше знаменателя).

Выполнить перегрузку операторов, необходимых для успешной компиляции следующего фрагмента кода:

```
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;
```