ЛЕКЦИЯ 5

- Модуль 2
- 15.11.2023
- Опечатанные классы
- Методы расширения, перегрузка операций

ЦЕЛИ ЛЕКЦИИ

- Изучить средства «запрета» наследования
- Познакомиться с методами расширения
- Разобраться с перегрузкой операций

ОПЕЧАТАННЫЕ КЛАССЫ И МЕТОДЫ РАСШИРЕНИЯ

ОПЕЧАТАННЫЕ (SEALED) КЛАССЫ

Модификатор **sealed** запрещает другим классам наследовать его

```
public class A   {  }

public sealed class B : A {  }
public class C : B {  } // compilation error!!!
```

Чтобы определить, нужно ли запечатывать класс, метод или свойство, имейте в виду следующее:

- потенциальные преимущества, которые производные классы могут получить от возможности настраивать ваш класс
- вероятность того, что производные классы могут корректировать ваши классы, препятствуя их нормальной работе

SEALED ЧЛЕНЫ КЛАССА

`sealed` можно использовать для методов и свойств, которые переопределяют в наследнике виртуальный член

Класс при этом можно наследовать, но нельзя переопределить виртуальный метод или свойство

```
class X {
    protected virtual void F() { Console.WriteLine("X.F"); }
    protected virtual void F2() { Console.WriteLine("X.F2"); }
}

class Y : X {
    sealed protected override void F() { Console.WriteLine("Y.F"); }
    protected override void F2() { Console.WriteLine("Y.F2"); }
}

class Z : Y {
    // Attempting to override F causes compiler error CS0239.
    // protected override void F() { Console.WriteLine("Z.F"); }

    // Overriding F2 is allowed.
    protected override void F2() { Console.WriteLine("Z.F2"); }
}
```

МЕТОДЫ РАСШИРЕНИЯ

- **Метод расширения** позволяет «условно» добавлять методы в существующие типы без создания нового производного типа
  - *не требуется изменение типа и перекомпиляция*

Особенности методов расширения:

- Методы расширения определяются как статические методы, но вызываются с помощью синтаксиса обращения к методу экземпляра
- Первый параметр метода расширения показывает каким типом оперирует метод, при этом ему предшествует модификатор `this`
- Метод расширения вводится в область действия только если пространство имен импортировано в код с помощью директивы `using`
  - Метод расширения можно ввести в область действия с помощью следующей директивы `using`: `using ИдентификаторМетодаРасширения;`

ПРИМЕР. РАСШИРЕНИЕ КЛАССА STRING

```
public static class StringExtensions
{
    public static int WordCount(this String str)
    {
        return str.Split(new[] { ' ', '.', '?' }, StringSplitOptions.RemoveEmptyEntries).Length;
    }
}
```

```
string s = "Hello, how are you?";
int wordCount = s.WordCount();
Console.WriteLine($"The string \"{s}\" has {wordCount} words.");
```

ПРИМЕР. ДЛЯ СОБСТВЕННОГО КЛАССА

```
// Внешний класс, который мы не можем изменить
sealed class MyData
{
private double D1;       // поля
private double D2;
private double D3;
// конструктор
public MyData(double d1, double d2, double d3)
{ D1 = d1; D2 = d2; D3 = d3; }
// метод
public double Sum()
{ return D1 + D2 + D3; }
}
```

Желательно иметь такое обращение: **md.Average();**

```
static class ExtendMyData
{
    public static double Average(MyData md) =>
                                    md.Sum() / 3;

}
class Program
{
    static void Main()
    {

        MyData md = new MyData(3, 4, 5);
        Console.WriteLine("Average: {0}",
                ExtendMyData.Average(md));
    }
}
```

Ссылка на объект типа MyData

Используем ссылку на объект типа MyData

Объект типа MyData

Вызов статического метода

МЕТОД РАСШИРЕНИЯ

Обязательно static!

```
static class ExtendMyData
{
    public static double Average( this MyData md )
    {   ...   }
}
```

Обязательно public и static!

Ключевое слово

Использование метода расширения:

```
MyData md = new MyData(3, 4, 5);
Console.WriteLine("Average: {0}", md.Average());
Console.Write("Average: {0}", ExtendMyData.Average(md));
```

ПЕРЕГРУЗКА ОПЕРАЦИЙ

ПЕРЕГРУЗКА ОПЕРАЦИЙ

Перегрузка операций – исключительно синтаксическое удобство, являющееся альтернативой методам

Перегрузка операций допустима для:

- классов
- структур и записей

Перегрузки операций стоит определять только в случаях, когда выполнение соответствующих операций является интуитивно понятным (логичным) для данного типа (например, вектора)

Сравните синтаксис:

- С использованием перегрузок операций:

```
C = (A + B) * D;
```

- Без перегрузки операций, с использованием методов:

```
C = D.Multiply(A.Add(B));
```

УМЕСТНОЕ ИСПОЛЬЗОВАНИЕ ПЕРЕГРУЗКИ ОПЕРАЦИЙ

Ниже представлен перевод **C# Language Specification** ECMA-334 5th Edition / December 2017: *(стр. 116)*

Хотя определяемые перегрузки операций могут выполнять любые вычисления, **настоятельно не рекомендуется** предоставлять реализации кроме тех случаев, когда определяемое поведение интуитивно понятно

Так, например, реализация **перегрузки операции ==** должна сравнивать операнды на равенство и возвращать соответствующий результат типа **bool**

ПРАВИЛА ПЕРЕГРУЗКИ ОПЕРАЦИЙ

При определении перегрузки операций для типов существует ряд правил:

- Операция должна быть открытым статическим методом (**`public static`**)
- Хотя бы один из параметров метода-операции должен иметь тот тип, в котором определяется
- Параметры операций могут передаваться либо по значению, либо с использованием модификатора **`in`** (запрет на **`ref`** или **`out`**)
- Допускается определять несколько перегрузок операции с разными параметрами
- Операции могут, но идеологически не должны изменять значение передаваемых им аргументов (актуально для ссылочных типов)

СИНТАКСИС ПЕРЕГРУЗКИ ОПЕРАЦИЙ

```
5 * obj – 45.3
12 – obj * 4.3
```

Для перегрузки **унарных** операций используется общий синтаксис:

```
public static <тип возвр. знач.> operator<Символ>(<Параметр>) { [тело…] }
```

Для перегрузки **бинарных** операций используется общий синтаксис:

```
public static <тип возвр. знач.> operator<Символ>
        (<Параметр 1>, <Параметр 2>) { [тело…] }
```

Не допускается добавлять параметры операций, менять их приоритет, синтаксис или ассоциативность. Кроме того, нельзя определять новые операции

Тернарная условная операция не перегружается

ПРИМЕР ПЕРЕГРУЗКИ ОПЕРАЦИЙ

```
public class Vector
{
    public double x;
    public double y;
    public double z;

    public Vector(double vectorX, double vectorY, double vectorZ)
        => (x, y, z) = (vectorX, vectorY, vectorZ);

    public static Vector operator + (Vector left, Vector right)
        => new Vector(left.x + right.x, left.y + right.y, left.z + right.z);

    public static Vector operator * (Vector left, Vector right)
        => new Vector(left.x * right.x, left.y * right.y, left.z * right.z);

    // Перегрузка для умножения. Обратите внимание, что она не коммутативна!
    public static Vector operator * (Vector left, double right)
        => new Vector(left.x * right, left.y * right, left.z * right);
}
```

Сокращённый синтаксис инициализации.

ЯВНО ПЕРЕГРУЖАЕМЫЕ ОПЕРАЦИИ

Операции	Комментарии
Унарные: **+**, **–**, **!**, **~**, **++**, **– –** , **true**, **false**	true и false – операции, позволяющие использовать тип в условных выражениях, должны перегружаться одновременно. При перегрузке инкремента и декремента явно перегружается префиксный (и автоматически неявно перегружается постфиксный).
Бинарные: *****, **/**, **%**, **+**, **–**, **>>**, **<<**, **&**, **|** , **^**, **<**, **<=**, **>**, **>=**, **==**, **!=**	Операции сравнения обязательно перегружаются попарно: > и <, <= и >=, == и !=.

Операции true, false не допускают явного вызова

НЕЯВНО ПЕРЕГРУЖАЕМЫЕ ОПЕРАЦИИ

Часть операций в C# **перегружаются только неявно** – компилятор добавляет их реализации при определении перегрузок других операций:

- Операции составного присваивания (**+=**, **–=**, **/=** и т. д.) перегружаются неявно в случае добавления перегрузки соответствующей бинарной операции
- Логические условные операции **||**, **&&** неявно перегружаются при определении перегрузок для **true** и **false** и операций **&** или **|** соответственно

Операция [ ] в C# фактически не перегружается, для организации ожидаемого от неё поведения используются индексаторы

ПРИМЕР: КЛАСС «РАЦИОНАЛЬНАЯ ДРОБЬ»

```
public class Fraction
{
    int num;     // числитель
    int den;     // знаменатель

    public Fraction(int n, int d) { //  Конструктор
        if (d > 0) { num = n; den = d; return; }
        if (d < 0) { num = -n; den = -d; return; }
        throw new ArgumentException(
       $"Нулевой знаменатель: {n}/{d}", "d");
    }

    public void Print() => Console.WriteLine(this.ToString());
    // ...
}
```

ПЕРЕГРУЗКА ОПЕРАЦИЙ + И –

```
// Унарный минус.
static public Fraction operator-(Fraction f)
{
    return new Fraction(-f.num, f.den);
}

static public Fraction operator+(Fraction f1, Fraction f2)
{
    int n = f1.num * f2.den + f1.den * f2.num;
    int d = f1.den * f2.den;
    return new Fraction(n, d);
}
```

ПЕРЕГРУЗКА ОПЕРАЦИЙ < И >

```
// Данные операции обязательно перегружаются парно.
static public bool operator < (Fraction f1, Fraction f2) {
    return f1.num * f2.den < f1.den * f2.num;
}

static public bool operator > (Fraction f1, Fraction f2) {
    // Операцию > можно выразить через <.
    return f2 < f1;
}
```

ПРИМЕНЕНИЕ ПЕРЕГРУЗОК ОПЕРАЦИЙ

```
Fraction A = new Fraction(1, 4);
A.Print();              // 1/4
(-A).Print();           // -1/4

A.Print();             // 1/4
Fraction B = new Fraction(3, 5);
(A + B).Print();       // 17/20
Fraction C;
if (A > B) {
    C = A;
}
else {
    C = B;
}
C.Print();             // 3/5
```

Вывод:
1/4
-1/4
1/4
17/20
3/5

ВОЗМОЖНЫЕ ОШИБКИ

```
// Ошибка компиляции – у операции «.» приоритет выше:
// Error - Operator '-' cannot be applied to operand of type 'void'
-A.Print();

// Ошибка компиляции – такой синтаксис недопустим в C#.
Fraction D = Fraction.operator+(A, B);
```

CS1003	Syntax error, ',' expected
CS8179	Predefined type 'System.ValueTuple`2' is not defined or imported
CS0023	Operator '+' cannot be applied to operand of type '(Demo_OperOverload.Fraction, Demo_OperOverload.Fraction)'
CS0201	Only assignment, call, increment, decrement, await, and new object expressions can be used as a statement
CS1001	Identifier expected
CS1002	; expected
CS0117	'Fraction' does not contain a definition for ''

ОПЕРАЦИИ ПОЛЬЗОВАТЕЛЬСКИХ ПРИВЕДЕНИЙ ТИПОВ

C# допускает определение собственных **явных** (*explicit*) и **неявных** (*implicit*) **операций приведения типа**, для этого всегда используется метод с заголовком вида:

```
public static explicit/implicit operator <Тип результата>
                    (<Приводимый параметр>) { [тело…] }
```

1) Для одного типа нельзя одновременно определить операции явного и неявного приведения типов (возникнет ошибка компиляции *CS0557*: Duplicate user-defined conversion in type <T>);

2) Операции **is** и **as** игнорируют пользовательские приведения типов

ОПЕРАЦИИ ПРИВЕДЕНИЯ ТИПОВ В КЛАССЕ FRACTION

В случае с классом Fraction операцию приведения к int стоит определить явной – в результате целочисленного деления теряется точность вычислений:

```
// Неявные приведения к Fraction и double – точность не теряется.
public static implicit operator Fraction(int x) => new Fraction(x, 1);

public static implicit operator double(Fraction f) =>
    (double)f.num / f.den;

// Явное приведение к int – может теряться точность.
public static explicit operator int(Fraction f) => f.num / f.den;
```

ИСПОЛЬЗОВАНИЕ ПРИВЕДЕНИЙ ТИПОВ В КЛАССЕ FRACTION

```
Fraction B = new Fraction(3, 5);
Fraction C = B + 3;

Console.Write("C = ");
C.Print();                          // 18/5
C = B + (-3);                       // C = B + (Fraction)(-3);

Console.Write("C = ");
C.Print();                          // -12/5
double res = 3.0 * C;                  // Неявное приведение к double.
int res = 3 * (int)C;                  // Явное приведение к int.
Console.WriteLine("res = " + res);  // -6
```

ПЕРЕГРУЗКА TRUE И FALSE

```
static public bool operator true(Fraction f) => f.num > f.den;

static public bool operator false(Fraction f) => f.num <= f.den;

static public Fraction operator-(Fraction f1, Fraction f2) {
    int n = f1.num * f2.den - f1.den * f2.num;
    int d = f1.den * f2.den;
    return new Fraction(n, d);
}

// Имеет приоритет над operator true (if или while) !
public static implicit operator bool(Fraction f) => f.num > f.den;
```

ПРИМЕНЕНИЕ ПЕРЕГРУЗКИ TRUE И FALSE

```
Fraction A = new Fraction(13, 3), B = new Fraction(1, 1);
while (A) {
    Console.Write("Неправильная дробь: ");
    A.Print();
    A -= B;
}
Console.Write("Результат: ");
A.Print();
```

Вывод:

```
Неправильная дробь: 13/3
Неправильная дробь: 10/3
Неправильная дробь: 7/3
Неправильная дробь: 4/3
Результат: 1/3
```

ПРАВИЛА ПЕРЕГРУЗКИ ОПЕРАЦИЙ ++ И --

Синтаксис C++ :

- `public T operator++();  // префикс`
- `public T operator++(int); // постфикс`

В C# **единая перегрузка** для обоих случаев. Принципы работы:

- Возвращать из метода необходимо **новое** значение;
- Если вызывается постфиксная операция, то старое (сохраненное) значение/ссылка используется в выражении;
- Если вызывается префиксная операция, то новое значение/ссылка используется в выражении;
- Компилятор самостоятельно обрабатывает эти различия!

Для получения ожидаемого поведения необходимо создавать новый объект и возвращать именно его из метода в качестве результата операции.

Если просто изменить переданный объект по ссылке – будет “сюрприз” при постфиксном использовании операции (увлекательная отладка в качестве бонуса)

ПРИМЕР ПЕРЕГРУЗКИ ОПЕРАЦИЙ ++ И --

```
class MyComplex  {
    public double re, im;
    public MyComplex(double xre, double xim) {
           re = xre;
           im = xim;
    }

    public static MyComplex operator -- (MyComplex mc) {
         return new MyComplex(mc.re - 1, mc.im - 1);
    }
    // неправильная реализация:
    public static MyComplex operator ++ (MyComplex mc) {
           mc.re++; mc.im++;
           return mc;
    }
}
```

ПРИМЕР ПЕРЕГРУЗКИ ОПЕРАЦИЙ ++ И --

```
MyComplex c1 = new MyComplex(11, 22);
Console.Write($"  c1 => {FormatComplex(c1)}");
Console.Write($"++c1 => {FormatComplex(++c1)}");
Console.Write($"c1++ => {FormatComplex(c1++)}");
Console.Write($"  c1 => {FormatComplex(c1)}");
Console.Write($"--c1 => {FormatComplex(--c1)}");
Console.Write($"c1-- => {FormatComplex(c1--)}");
Console.Write($"  c1 => {FormatComplex(c1)}");

static string FormatComplex(MyComplex cs)
    => $"real= {cs.re}, image= {cs.im}\n";
```

Вывод (неправильный):

```
 c1 => real= 11, image= 22
++c1 => real= 12, image= 23
c1++ => real= 13, image= 24
  c1 => real= 13, image= 24
--c1 => real= 12, image= 23
c1-- => real= 12, image= 23
  c1 => real= 11, image= 22
```

ИЗМЕНЕНИЕ РЕАЛИЗАЦИИ ОПЕРАЦИИ ++

```
// Правильная реализация:
public static MyComplex operator ++ (MyComplex mc)
{
    return new MyComplex(mc.re + 1, mc.im + 1);
}
```

Вывод (до исправлений):

```
 c1 => real= 11, image= 22
++c1 => real= 12, image= 23
c1++ => real= 13, image= 24
  c1 => real= 13, image= 24
--c1 => real= 12, image= 23
c1-- => real= 12, image= 23
  c1 => real= 11, image= 22
```

Вывод (после исправлений):

```
 c1 => real= 11, image= 22
++c1 => real= 12, image= 23
c1++ => real= 12, image= 23
  c1 => real= 13, image= 24
--c1 => real= 12, image= 23
c1-- => real= 12, image= 23
  c1 => real= 11, image= 22
```

ПЕРЕГРУЗКА ОПЕРАЦИЙ == И !=

Перегрузка == и != возможна только в паре (как и в случае с другими операциями сравнения)

- При перегрузке == рекомендуется перегрузить Equals():
  - Некоторые .Net-языки не поддерживают перегрузку операций
  - Equals() и == должны вести себя одинаково
  - Если нет парной перегрузки Equals() и == , то возможны сюрпризы…
- При перегрузке Equals() также настоятельно рекомендуется перегрузить GetHashCode(). Важно при использовании в словарях в качестве ключа. Логика:
  - Если Equals() == true, то и GetHashCode() обязан совпадать
  - Если GetHashCode() совпадает, то Equals() может отличаться (если при совпадении хеш-кода Equals(…) != true - это коллизия)

ПЕРЕГРУЗКИ ОПЕРАЦИЙ == И != ДЛЯ FRACTION

```
public static bool operator == (Fraction lhs, Fraction rhs) { return (double)lhs == rhs; }

public static bool operator != (Fraction lhs, Fraction rhs) { return !(lhs == rhs); }

public override bool Equals(object o) { return this == o as Fraction; }


public override int GetHashCode()
{
    Reduce();   // ВАЖНО! Считаем, что дробь несократима!
    return num ^ den;
}
```

УСЛОВНЫЕ ОПЕРАЦИИ && И | |

Операция & или операция | вызываются косвенно, если используется && или | | соответственно:

- Операция & вызывается только если первый операнд true
- Операция | вызывается только если первый операнд false

- При использовании && или | | левый операнд оценивается с использованием операции false или операции true (соответственно):
  - Помните, что нельзя использовать && и | | пока не перегружены true и false

- Почему бы не использовать неявное приведение типа к bool (implicit operator bool)?
  - Компилятор так не делает (просто факт)…

УСЛОВНЫЕ ОПЕРАЦИИ && И || ДЛЯ FRACTION

```
// Работает по короткой схеме для &&.
public static Fraction operator & (Fraction lhs, Fraction rhs)
                                        => new Fraction(lhs.num, lhs.den);
// НЕ работает по короткой схеме ||, т. к.
// тип возвращаемого значения не Fraction.
public static bool operator | (Fraction lhs, bool rhs) => (lhs.num > lhs.den) | rhs;

// Работает по короткой схеме для ||.
public static Fraction operator | (Fraction lhs, Fraction rhs)
        => new Fraction(rhs.num, rhs.den);
```

ИСПОЛЬЗОВАННАЯ ЛИТЕРАТУРА

- sealed (Справочник по C#) (https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/keywords/sealed)
- https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/operators/operator-overloading

  https://www.ecma-international.org/publications/standards/Ecma-334.htm