В.В. Подбельский

Использованы иллюстрации пособия Daniel Solis, Illustrated C#

Иллюстрации к курсу лекций по дисциплине «Программирование на C#»

08. Часть 2

Перегрузка Операций

Благодарность Д.Д. Сагалову за помощь в переработке презентаций

Перегрузка Операций

Перегрузка операций – исключительно синтаксическое удобство, являющееся альтернативой методам. Допускается перегрузка операций для классов, структур и записей.

Важно: 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));
```

https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/operators/operator-overloading

Уместное Использование Перегрузки Операций

Ниже представлен перевод **C# Language Specification** ECMA-334 5th Edition / December 2017:

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**.

While it is possible for a user-defined operator to perform any computation it pleases, implementations that produce results other than those that are intuitively expected are **strongly discouraged**. For example, an implementation of operator == should compare the two operands for equality and return an appropriate bool result.

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

(стр. 116)

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…] }
```

Обратите внимание: Вы не можете 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

Список явно перегружаемых операций в C# ограничен и в него входят:

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: *****, ***I***, **%**, **+**, **–**, **>>**, **<<**, **&**, **|**, **^**, **<**, **<=**, **>**, **>=**, **==**, **!=**	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…] }
```

Важно:

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

```
// 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;
```

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

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!

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

Если просто измените переданный объект по ссылке – получите “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 | |

(косвенно перегружаемые)

- **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);
```

Перегрузка Операции и Наследование

Warning: *данный слайд выходит за рамки темы лекции и в первую очередь предназначен для более продвинутой аудитории.*

```
class Base {
    public int Num { get; set; }
    public override string ToString() => Num.ToString(CultureInfo.InvariantCulture);
    public Base(int n) => Num = n;

    public static Base operator +(Base a, Base b) => new Base(a.Num + b.Num);

    // public static explicit operator Derived(Base b) => new Derived(b.Num);
    // user-defined conversions to or from a derived class are not allowed Demo_OperOverload.
}

class Derived : Base {
    public Derived(int n) : base(n) { }
    public static Derived operator +(Derived a, Derived b) =>
        new Derived(a.Num + b.Num);
}
```

```
Base resB = new Derived(5) + new Derived(2);       // 7 типа Derived.
```