**№ 4 Перегрузка операций, методы расширениями вложенные типы**

|  |  |
| --- | --- |
| **Вариант 8** | Класс ─ множество **Set**. Дополнительно перегрузить следующие операции: > ─ проверка на принадлежность (типа операции in множественного типа);\* ─ пересечение множеств; < ─ проверка на подмножество; явный Date() ─ приведение типа  Методы расширения:  1) Выделение первого числа, содержащегося в строке  2) Удаление положительных элементов из множества |

**Задание**

1) Создать заданный в варианте класс. Определить в классе необходимые методы, конструкторы, индексаторы и заданные перегруженные операции. Написать программу тестирования, в которой проверяется использование перегруженных операций.

2) Добавьте в свой класс вложенный объект Owner, который содержит Id, имя и организацию создателя. Проинициализируйте его

3) Добавьте в свой класс вложенный класс Date (дата создания). Проинициализируйте

4) Создайте статический класс StatisticOperation, содержащий 3 метода для работы с вашим классом (по варианту п.1): сумма, разница между максимальным и минимальным, подсчет количества элементов.

5) Добавьте к классу StatisticOperation методы расширения для типа string и вашего типа из задания№1. См. задание по вариантам.

**► Перегрузка операторов**

Наряду с методами мы можем также перегружать операторы. Например, пусть у нас есть следующий класс Counter:

class Counter

{

    public int Value { get; set; }

}

Данный класс представляет некоторый счетчик, значение которого хранится в свойстве Value.

И допустим, у нас есть два объекта класса Counter - два счетчика, которые мы хотим сравнивать или складывать на основании их свойства Value, используя стандартные операции сравнения и сложения:

Counter c1 = new Counter { Value = 23 };

Counter c2 = new Counter { Value = 45 };

bool result = c1 > c2;

Counter c3 = c1 + c2;

Но на данный момент ни операция сравнения, ни операция сложения для объектов Counter не доступны. Эти операции могут использоваться для ряда примитивных типов. Например, по умолчанию мы можем складывать числовые значения, но как складывать объекты комплексных типов - классов и структур компилятор не знает. *И для этого нам надо выполнить перегрузку нужных нам операторов.*

Перегрузка операторов заключается в определении в классе, для объектов которого мы хотим определить оператор, специального метода:

public static возвращаемый\_тип operator оператор(параметры)

{ }

Этот **метод должен иметь модификаторы** **public static**, так как перегружаемый оператор будет использоваться для всех объектов данного класса. Далее идет название возвращаемого типа. К примеру, в результате сложения двух объектов Counter мы ожидаем получить новый объект Counter. А в результате сравнения двух мы хотим получить объект типа bool, который указывает истинно ли условное выражение или ложно. Но в зависимости от задачи возвращаемые типы могут быть любыми.

Затем вместо названия метода идет ключевое слово **operator** и собственно сам оператор. И далее в скобках перечисляются параметры. **Бинарные операторы принимают два параметра, унарные - один параметр. И в любом случае один из параметров должен представлять тот тип - класс или структуру, в котором определяется оператор.**

**И, конечно, метод должен возвращать значение.**

Например, перегрузим ряд операторов для класса Counter:

class Counter

{

public int Value { get; set; }

public static Counter operator +(Counter c1, Counter c2)

{

return new Counter { Value = c1.Value + c2.Value };

}

public static bool operator >(Counter c1, Counter c2)

{

return c1.Value > c2.Value;

}

public static bool operator <(Counter c1, Counter c2)

{

return c1.Value < c2.Value;

}

}

Так как в случае с операцией сложения мы хотим сложить два объекта класса Counter, то оператор принимает два объекта этого класса. И так как мы хотим в результате сложения получить новый объект Counter, то данный класс также используется в качестве возвращаемого типа. Все действия этого оператора сводятся к созданию, нового объекта, свойство Value которого объединяет значения свойства Value обоих параметров.

**Операторы сравнения должны перегружаться парами**. То есть при перегрузке оператора из пары другой оператор тоже должен перегружаться. Ниже приведены эти пары:

* Операторы == и !=
* Операторы < и >
* Операторы <= и >=

Теперь используем перегруженные операторы в программе:

static void Main(string[] args)

{

Counter c1 = new Counter { Value = 23 };

Counter c2 = new Counter { Value = 45 };

bool result = c1 > c2;

Console.WriteLine(result); // false

Counter c3 = c1 + c2;

Console.WriteLine(c3.Value); // 23 + 45 = 68

Console.ReadKey();

}

Стоит отметить, что так как, по сути, определение оператора представляет собой метод, то этот метод мы также можем перегрузить, то есть создать для него еще одну версию. Например, добавим в класс Counter еще один оператор:

public static int operator +(Counter c1, int val)

{

    return c1.Value + val;

}

Данный метод складывает значение свойства Value и некоторое число, возвращая их сумму. И также мы можем применить этот оператор:

Counter c1 = new Counter { Value = 23 };

int d = c1 + 27; // 50

Console.WriteLine(d);

Следует учитывать, что **при перегрузке не должны изменяться те объекты, которые передаются в оператор через параметры**. Например, мы можем определить для класса Counter оператор инкремента:

public static Counter operator ++(Counter c1)

{

    c1.Value += 10;

    return c1;

}

Поскольку оператор унарный, он принимает только один параметр - объект того класса, в котором данный оператор определен. Но это **неправильное** определение инкремента, так как *оператор не должен менять значения своих параметров*.

Корректная перегрузка оператора инкремента будет выглядеть так:

public static Counter operator ++(Counter c1)

{

return new Counter { Value = c1.Value + 10 };

}

То есть возвращается новый объект, который содержит в свойстве Value инкрементированное значение.

При этом **нам не надо определять отдельно операторы для префиксного и для постфиксного инкремента** (а также декремента), **так как одна реализация будет работать в обоих случаях**.

Например, используем операцию префиксного инкремента:

Counter counter = new Counter() { Value = 10 };

Console.WriteLine($"{counter.Value}"); // 10

Console.WriteLine($"{(++counter).Value}"); // 20

Console.WriteLine($"{counter.Value}"); // 20

Консольный вывод:

10

20

20

Теперь используем постфиксный инкремент:

Counter counter = new Counter() { Value = 10 };

Console.WriteLine($"{counter.Value}"); // 10

Console.WriteLine($"{(counter++).Value}"); // 10

Console.WriteLine($"{counter.Value}"); // 20

Консольный вывод:

10

10

20

■ Также стоит отметить, что мы можем переопределить операторы **true** и **false**. Например, определим их в классе Counter:

class Counter

{

public int Value { get; set; }

public static bool operator true(Counter c1)

{

return c1.Value != 0;

}

public static bool operator false(Counter c1)

{

return c1.Value == 0;

}

// остальное содержимое класса

}

Эти операторы перегружаются, когда мы хотим использовать объект типа в качестве условия. Например:

Counter counter = new Counter() { Value = 0 };

if (counter)

Console.WriteLine(true);

else

Console.WriteLine(false);

При перегрузке операторов надо учитывать, что не все операторы можно перегрузить. В частности, мы можем перегрузить следующие операторы:

* унарные операторы +, -, !, ~, ++, --
* бинарные операторы +, -, \*, /, %
* операции сравнения ==, !=, <, >, <=, >=
* логические операторы &&, ||

И есть ряд операторов, которые нельзя перегрузить, например, операцию равенства = или тернарный оператор ?:, а также ряд других.

Полный список перегружаемых операторов можно найти в [документации msdn](http://msdn.microsoft.com/ru-ru/library/8edha89s.aspx)

При перегрузке операторов также следует помнить, что мы не можем изменить приоритет оператора или его ассоциативность, мы не можем создать новый оператор или изменить логику операторов в типах, который есть по умолчанию в .NET.

## ► Перегрузка операций преобразования типов

Ранее мы рассматривали явные и неявные преобразования примитивных типов. Например:

int x = 50;

byte y = (byte)x; // явное преобразование от int к byte

int z = y;  // неявное преобразование от byte к int

И было бы неплохо иметь возможность определять логику преобразования одних типов в другие. И с помощью перегрузки операторов мы можем это делать. Для этого в классе определяется метод следующей формы:

public static explicit|implicit operator Тип\_в\_который\_надо\_преобразовать(исходный\_тип param)

{

// логика преобразования

}

После модификаторов **public static** идет ключевое слово **explicit** (если преобразование явное, то есть нужна операция приведения типов) или **implicit** (если преобразование неявное). Затем идет ключевое слово **operator** и далее возвращаемый тип, в который надо преобразовать объект. В скобках в качестве параметра передается объект, который надо преобразовать.

Например, пусть у нас есть следующий класс Counter, который представляет счетчик-секундомер и который хранит количество секунд в свойстве Seconds:

class Counter

{

public int Seconds { get; set; }

public static implicit operator Counter(int x)

{

return new Counter { Seconds = x };

}

public static explicit operator int(Counter counter)

{

return counter.Seconds;

}

}

Первый оператор преобразует число - объект типа int к типу Counter. Его логика проста - создается новый объект Counter, у которого устанавливается свойство Seconds.

Второй оператор преобразует объект Counter к типу int, то есть получает из Counter число.

Применение операторов преобразования в программе:

static void Main(string[] args)

{

Counter counter1 = new Counter { Seconds = 23 };

int x = (int)counter1;

Console.WriteLine(x); // 23

Counter counter2 = x;

Console.WriteLine(counter2.Seconds); // 23

}

Поскольку операция преобразования из Counter в int определена с ключевым словом explicit, то есть как явное преобразование, то в этом случае необходимо применить операцию приведения типов:

int x = (int)counter1;

В случае с операцией преобразования от int к Counter ничего подобного делать не надо, поскольку данная операция определена с ключевым словом implicit, то есть как неявная. Какие операции преобразования делать явными, а какие неявные, в данном случае не столь важно, это решает разработчик по своему усмотрению.

Следует учитывать, *что* **оператор преобразования типов должен преобразовывать из типа или в тип, в котором этот оператор определен**. То есть оператор преобразования, определенный в типе Counter, должен либо принимать в качестве параметра объект типа Counter, либо возвращать объект типа Counter.

## ► Список List<T>

Класс List<T> из пространства имен System.Collections.Generic представляет простейший список однотипных объектов.

Среди его методов можно выделить следующие:

* **void Add(T item)**: добавление нового элемента в конец списка
* **int BinarySearch(T item)**: бинарный поиск элемента в списке. Если элемент найден, то метод возвращает индекс этого элемента в коллекции. При этом список должен быть отсортирован.
* **bool Remove(T item)**: удаляет элемент item из списка, и если удаление прошло успешно, то возвращает true
* **void RemoveAt(int index)**: удаление элемента по указанному индексу index
* **void Sort()**: сортировка списка
* **bool Contains (T item)**: проверяет содержится ли элемент item в списке
* **int Count { get; }**: возвращает количество элементов в списке

List<string> dinosaurs = new List<string>();

dinosaurs.Add("Tyrannosaurus");

dinosaurs.Add("Amargasaurus");

dinosaurs.Add("Mamenchisaurus");

Console.WriteLine("Count: {0}", dinosaurs.Count);

Посмотрим реализацию списка на примере:

using System;

using System.Collections.Generic;

namespace Collections

{

class Program

{

static void Main(string[] args)

{

List<int> numbers = new List<int>() { 1, 2, 3, 45 };

numbers.Add(6); // добавление элемента

numbers.AddRange(new int[] { 7, 8, 9 });

numbers.Insert(0, 666); // вставляем на первое место в списке число 666

numbers.RemoveAt(1); //  удаляем второй элемент

foreach (int i in numbers)

{

Console.WriteLine(i);

}

List<Person> people = new List<Person>(3);

people.Add(new Person() { Name = "Том" });

people.Add(new Person() { Name = "Билл" });

foreach (Person p in people)

{

Console.WriteLine(p.Name);

}

Console.ReadLine();

}

}

class Person

{

public string Name { get; set; }

}

}

## ► Методы расширения

Методы расширения (extension methods) **позволяют** добавлять новые методы в уже существующие типы без создания нового производного класса. Эта функциональность бывает особенно полезна, когда нам хочется добавить в некоторый тип новый метод, но сам тип (класс или структуру) мы изменить не можем, поскольку у нас нет доступа к исходному коду. Либо если мы не можем использовать стандартный механизм наследования, например, если классы определены с модификатором sealed.

Например, нам надо добавить для типа string новый метод:

class Program

{

static void Main(string[] args)

{

string s = "Привет мир";

char c = 'и';

int i = s.CharCount(c);

Console.WriteLine(i);

Console.Read();

}

}

public static class StringExtension

{

public static int CharCount(this string str, char c)

{

int counter = 0;

for (int i = 0; i < str.Length; i++)

{

if (str[i] == c)

counter++;

}

return counter;

}

}

Для того, чтобы создать метод расширения, **вначале надо создать статический класс, который и будет содержать этот метод**. В данном случае это класс StringExtension. **Затем объявляем статический метод**. Суть нашего метода расширения - подсчет количества определенных символов в строке.

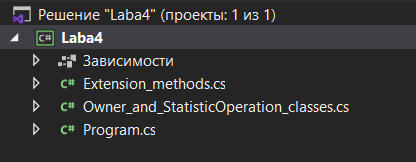
Собственно метод расширения - это статический метод, который отличается от обычного тем, что **в качестве первого параметра всегда принимает такую конструкцию: *this имя\_типа название\_параметра***, то есть в нашем случае this string str. Так как наш метод будет относиться к типу string, то мы и используем данный тип.

Затем у всех строк мы можем вызвать данный метод: int i = s.CharCount(c);. Причем нам уже не надо указывать первый параметр. Значения для остальных параметров передаются в обычном порядке.

Применение методов расширения очень удобно, но при этом надо помнить, что метод расширения никогда не будет вызван, если он имеет ту же сигнатуру, что и метод, изначально определенный в типе.

Также следует учитывать, что *методы расширения действуют на уровне пространства имен*. То есть, если добавить в проект другое пространство имен, то метод не будет применяться к строкам, и в этом случае надо будет подключить пространство имен метода через директиву using.

**КОД ВСЕЙ ЛАБЫ**

****

**Extension\_methods.cs**

using System;

using System.Collections.Generic;

using static Laba4.Program; //для Set !!!

namespace Laba4

{

static class Extension\_methods

{

//выделение первого числа в строке

public static int FindFirstNum(this string str)

{

string num = "";

foreach(char i in str)

{

//если символ строки равен от 0 до 9, значит записываем его в строку num

if (i >= 48 && i <= 57)

{

num += i;

}

//если в строке num уже есть число, а мы встретили символ, отличный от цифры - значит мы уже выделили первое число

else if (num != "")

{

break;

}

}

//если в строке чисел нет

if (num == "")

{

return -1;

}

else

{

return Convert.ToInt32(num);

}

}

//------------------------------------------------------------------

//Удаление положительных элементов из множества Set

public static List<int> RemovePosEls(this Set set)

{

List<int> posSet = new List<int> { };

foreach(int i in set.Data)

{

if (i < 0)

{

posSet.Add(i);

}

}

return posSet;

}

}

}

**Owner\_and\_StatisticOperation\_classes.cs**

using System;

using static Laba4.Program; //для Set !!!

namespace Laba4

{

//класс Owner

public class Owner

{

readonly int id = 8;

readonly string name = "Софья";

readonly string organization = "БГТУ";

//конструкторы

public Owner(int id, string name, string organization)

{

this.id = id;

this.name = name;

this.organization = organization;

}

public Owner() { }

//метод для вывода значения полей объекта типа Owner

public void Print()

{

Console.WriteLine($"\nid : {this.id}\n" +

$"name : {this.name}\n" +

$"organization : {this.organization}\n");

}

}

//статический класс StatisticOperation с тремя методами для работы со множеством

static class StatisticOperation

{

public static int Sum(Set set)

{

int sum = 0;

foreach(int el in set.Data)

{

sum += el;

}

return sum;

}

public static int MaxMinDiff(Set set)

{

int min = set.Data[0];

int max = set.Data[0];

foreach(int el in set.Data)

{

if (el < min) { min = el; }

if (el > max) { max = el; }

}

return max - min;

}

public static int AmountOfEls(Set set)

{

return set.Data.Count;

}

}

}

**Program.cs**

using System;

using System.Collections.Generic; //List<type\_name>

namespace Laba4

{

class Program

{

public class Set

{

//инициализация объекта типа Owner

public Owner owner = new Owner();

//------------------------------------------------------------------

//конструкторы

public Set() {

date = DateTime.Now;

}

public Set(List<int> data)

{

date = DateTime.Now;

this.data = data;

}

private List<int> data = new List<int> { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

public List<int> Data

{

get { return data; }

set { data = value; }

}

//------------------------------------------------------------------

//Символы > и < напоминают мне символы включения в дискретной математике, так что это value1 < value2 - значит, что value2 включено в value1

//а это value1 > value2 - значит, что value1 включено в value2

//Проверка на подмножество

//Множество subset принадлежит множеству set (является его подмножеством)

//Если любой элемент множества subset также принадлежит множеству set

public static bool operator <(Set set, Set subset)

{

int subset\_length = subset.data.Count;

int num\_of\_subset\_elements\_in\_set = 0;

//если в множестве data содержится число num, значит num принадлежит data

foreach (int subset\_item in subset.data)

{

if (set.data.Contains(subset\_item)) {

num\_of\_subset\_elements\_in\_set++;

}

}

if (num\_of\_subset\_elements\_in\_set == subset\_length)

{

return true;

}

return false;

}

//компилятор требует перегрузки аналогичной выше, но для >

public static bool operator >(Set subset, Set set)

{

int subset\_length = subset.data.Count;

int num\_of\_subset\_elements\_in\_set = 0;

//если в множестве data содержится число num, значит num принадлежит data

foreach (int subset\_item in subset.data)

{

if (set.data.Contains(subset\_item))

{

num\_of\_subset\_elements\_in\_set++;

}

}

if (num\_of\_subset\_elements\_in\_set == subset\_length)

{

return true;

}

return false;

}

//------------------------------------------------------------------

//Проверка на принадлежность

//Число num принадлежит множеству set

public static bool operator >(int num, Set set)

{

foreach (int item in set.data)

{

if (num == item)

{

//если num содержится в set

return true; }

}

return false;

}

//компилятор требует перегрузки аналогичной выше, но для <

public static bool operator <(int num, Set set)

{

foreach (int item in set.data)

{

if (num != item)

{

//если num содержится в set

return false;

}

}

return true;

}

//------------------------------------------------------------------

//Пересечение множеств

//Возвращает массив из элементов общих для обоих множеств

public static List<int> operator \*(Set set1, Set set2)

{

List<int> intersection = new List<int> { };

foreach (int item1 in set1.data)

{

foreach (int item2 in set2.data)

{

if (item1 == item2)

{

intersection.Add(item1);

}

}

}

return intersection;

}

//------------------------------------------------------------------

//перегрузка операции преобразования типов

//явное приведение к типу DateTime

//преобразует тип Set к типу DateTime, при этом в переменной сохраняется время создания объекта типа Set

private DateTime date;

public static explicit operator DateTime(Set set)

{

return set.date;

}

//------------------------------------------------------------------

//вывод множества в виде { <элементы множества через запятую> }

public void PrintSet()

{

Console.Write("{");

foreach (int i in this.data)

{

Console.Write(" " + i + " ");

}

Console.WriteLine("}");

}

//проверка перегруженнных операторов

public static void OverloadOpTest(Set set1, Set set2, Set set3, int num)

{

Console.Write("set1 : ");

set1.PrintSet();

Console.Write("\nset2 : ");

set2.PrintSet();

Console.Write("\nset3 : ");

set3.PrintSet();

Console.Write($"\nnumber : {num}");

Console.WriteLine("\n\n|||||||||||||||||||| проверка на принадлежность ||||||||||||||||||||");

Console.WriteLine($"set1 принадлежит number (number < set1) : {num < set1}");

Console.WriteLine($"number принадлежит set1 (number > set1) : {num > set1}");

Console.WriteLine("\n|||||||||||||||||||| проверка на подмножество ||||||||||||||||||||");

Console.WriteLine($"set1 является подмножеством set2 (set1 > set2) : {set1 > set2}");

Console.WriteLine($"set3 является подмножеством set2 (set2 < set3) : {set2 < set3}");

Console.WriteLine("\n|||||||||||||||||||| пересечение множеств ||||||||||||||||||||");

Console.Write("пересечением множеств set1 и set2 является список (set1 \* set2) : ");

Console.Write("{");

foreach (int i in set1 \* set2)

{

Console.Write(" " + i + " ");

}

Console.WriteLine("}\n");

}

//------------------------------------------------------------------

//вложенный класс Date(дата создания)

public class Date

{

public readonly DateTime dateOfCreation = DateTime.Now;

}

public Date dateObj = new Date();

}

static void Main(string[] args)

{

//вывод значения поля типа Owner множества Set

Console.Write("------------- вывод значения поля типа Owner множества setOwnerCheck -------------");

Set setOwnerCheck = new Set();

setOwnerCheck.owner.Print();

//------------------------------------------------------------------

//вывод значения поля объекта типа Date множества setDateCheck

Console.WriteLine("------------- вывод значения поля объекта типа Date множества setDateCheck -------------");

Set setDateCheck = new Set();

Console.WriteLine("Дата создания объекта dateObj класса Date, который вложен в класс Set : " + setDateCheck.dateObj.dateOfCreation);

//------------------------------------------------------------------

//явное приведение объекта типа Set к типу DateTime

Console.WriteLine("\n------------- явное приведение объекта типа Set к типу DateTime -------------");

Set explTypeCast = new Set();

Console.WriteLine("Приведение (дата создания объекта типа Set) : "+ (DateTime)explTypeCast);

//------------------------------------------------------------------

//проверка перегруженнных операторов

Console.WriteLine("\n------------- проверка перегруженнных операторов -------------");

Set set1 = new Set(new List<int> { 1, 5, 6, 9, -7, 5, 34 });

Set set2 = new Set(new List<int> { 4, 8, 6, -9, -4, 0, 34, 6 });

Set set3 = new Set(new List<int> { 6, -9, -4, 34 });

int num = 9;

Set.OverloadOpTest(set1, set2, set3, num);

//------------------------------------------------------------------

//выделение первого числа из строки

Console.WriteLine("-------------Выделение первого числа из строки-------------");

Console.WriteLine($"ты предал христа : {"ты предал христа".FindFirstNum()}");

Console.WriteLine($"aaabbb0111ccc34 : {"aaabbb0111ccc34".FindFirstNum()}");

//------------------------------------------------------------------

//Удаление положительных элементов из множества Set

Console.WriteLine("\n-------------Удаление положительных элементов из множества Set-------------");

Set set\_pos\_neg = new Set(new List<int> { -2, 5, 0, -2, 52, -7 });

Console.Write("Удаление положительных элементов из множества : {");

foreach(int i in set\_pos\_neg.Data)

{

Console.Write(" " + i + " ");

}

Console.WriteLine("}");

Set set\_pos = new Set(set\_pos\_neg.RemovePosEls());

Console.Write("Результат : {");

foreach (int i in set\_pos.Data)

{

Console.Write(" " + i + " ");

}

Console.WriteLine("}");

//------------------------------------------------------------------

//Проверка методов из класса StatisticOperation

Console.WriteLine("\n-------------Проверка методов из класса StatisticOperation-------------");

Set set = new Set(new List<int> { 1, 2, 3, -8, 19 });

Console.Write("Множество : ");

set.PrintSet();

//Sum - сумма элементов множества

Console.WriteLine("сумма элементов множества : " + StatisticOperation.Sum(set));

//MaxMinDiff - разница между максимальным и минимальным элементами

Console.WriteLine("разница между максимальным и минимальным элементами : " + StatisticOperation.MaxMinDiff(set));

//AmountOfEls - подсчет количества элементов

Console.WriteLine("подсчет количества элементов : " + StatisticOperation.AmountOfEls(set));

}

}

}

**Вопросы**

1. Перечислите свойства внутренних и вложенных классов?

##### 1. Понятие вложенного класса. Общая форма объявления вложенного класса

В языке C# любой класс в своей реализации может содержать объявление другого класса. Класс, который объявляется в пределах фигурных скобок другого класса, называется ***вложенным классом***.

В простейшем случае, общая форма объявления вложенного класса в классе имеет вид:

// класс, который содержит в реализации объявление другого класса

class Outer

{

// поля и методы класса Outer

// ...

class Inner

{

// поля и методы класса Inner

// ...

}

// поля и методы класса Outer

// ...

}

где

* *Outer* – имя класса, который содержит в своем теле объявление другого класса с именем *Inner*;
* *Inner* – имя класса, который объявляется в классе *Outer*.

Классы Outer и Inner могут содержать различные спецификаторы, которые определяют доступ (private, protected, public) или другие свойства класса (static, sealed и т.д.).

##### 2. Как выглядит общая форма объявления объекта вложенного класса?

Объект вложенного класса можно объявить в случае, когда вложенный класс объявлен как видимый. Это значит, что видимый вложенный класс должен быть объявлен с любым модификатором доступа кроме private.

Если вложенный класс объявлен как не private-класс, то создание экземпляра этого класса имеет следующий вид:

Outer.Inner objInner = new Outer.Inner();

где

* Outer – имя внешнего класса, в котором объявляется вложенный класс Inner;
* Inner – имя вложенного класса, который объявлен в пределах фигурных скобок класса Outer;
* objInner – имя объекта (экземпляра) класса, который создается;
* Outer.Inner() – имя конструктора по умолчанию, который вызывается для создания объекта класса. Если в классе Inner реализованы другие параметризированные конструкторы, то этот вызов может быть другим.

##### 3. Какие модификаторы доступа могут применяться к вложенным классам?

К вложенным классам могут применяться точно такие же модификаторы доступа как и к обычным – невложенным классам:

* private. В этом случае вложенный класс является видимым в пределах фигурных скобок { } внешнего класса;
* public. В этом случае вложенный класс доступен в пределах внешнего класса, из экземпляра внешнего класса, из унаследованного класса а также за пределами сборки;
* protected. В этом случае класс доступен в пределах внешнего класса а также из методов унаследованных классов;
* internal. Вложенный класс доступен в пределах сборки и недоступен за пределами сборки;
* protected internal. Вложенный класс может быть доступен из методов внешнего класса, методов унаследованных классов или недоступен из методов за пределами текущей сборки.

Действие модификаторов точно такое же как и в случае с членами данных (методами) класса.

##### 5. Объявление и использование статического вложенного класса в нестатическом. Пример

В нестатическом классе может быть объявлен статический класс (с ключевым словом static). Это означает, что статический класс данного класса есть уникальным общим ресурсом. Доступ к элементам статического класса осуществляется непосредственно по имени статического класса, перед которым через точку следует имя (имена) внешнего класса (классов).

**Пример.** Объявляется класс Outer, содержащий объявление вложенного статического (static) класса Inner. В классе Inner объявляется статическая внутренняя public-переменная. Текст класса Outer следующий:

// класс Outer - нестатический

public class Outer

{

// внутренние переменные класса Outer

public int d;

static public int sd;

// статический вложенный класс Inner

public static class Inner

{

// только статические переменные могут объявляться в статическом классе

public static int sd; // статическая внутренняя переменная класса Inner

}

}

Использование статического вложенного класса может быть приблизительно таким:

// доступ к статической переменной статического класса Outer.Inner

Outer.Inner.sd = 45;

// создать объект (экземпляр) класса Outer

Outer o = new Outer();

o.d = 30; // есть доступ к переменным экземпляра класса через объект

Outer.sd = 102; // доступ к статической переменной класса Outer

// создавать объект статического класса нельзя

//Outer.Inner i = new Outer.Inner(); - это ошибка!

### 

##### 6. Можно ли в статическом классе объявить нестатический вложенный класс?

Да, можно. В статическом классе *можно* объявлять нестатические вложенные *классы*. Однако в статическом классе *нельзя* объявлять нестатические *переменные* и *методы*.

**Например**. Пусть объявляется статический класс Outer, содержащий реализацию нестатического класса Inner

// класс Outer - статический

public static class Outer

{

// внутренние переменные класса Outer

// только статические экземпляры можно объявлять в статическом классе

// public int d; - ошибка!

static public int sd;

// нестатический вложенный класс Inner - может объявляться в статическом классе

public class Inner

{

public int d; // нестатическая переменная

public static int sd; // статическая внутренняя переменная класса Inner

}

}

Использование классов Outer и Outer.Inner может быть, например, таким:

// использование нестатического класса в статическом классе

// доступ к статической переменной sd класса Outer.Inner

Outer.Inner.sd = 230;

// доступ к статической переменной sd класса Outer

Outer.sd = 132;

// Нельзя создавать экземпляр (объект) статического класса

// Outer o = new Outer(); // ошибка, поскольку Outer - статический класс

// создать экземпляр вложенного класса Outer.Inner

Outer.Inner i = new Outer.Inner();

і.d = 323; // доступ к нестатической переменной d класса Outer.Inner

2. Что такое статический класс и какие у него свойства?

### Статические классы

Статические классы объявляются с модификатором static и **могут содержать только статические поля, свойства и методы**. Например, если бы класс Account имел бы только статические переменные, свойства и методы, то его можно было бы объявить как статический:

static class Account

{

private static decimal minSum = 100; // минимальная допустимая сумма для всех счетов

public static decimal MinSum

{

get { return minSum; }

set { if (value > 0) minSum = value; }

}

// подсчет суммы на счете через определенный период по определенной ставке

public static decimal GetSum(decimal sum, decimal rate, int period)

{

decimal result = sum;

for (int i = 1; i <= period; i++)

result = result + result \* rate / 100;

return result;

}

## }

Кроме этого, на статический класс, накладываются как минимум **следующие ограничения**:

* статический класс *не может служить базовым для других классов*, а так же, сам *не может быть наследником* какого-либо произвольного класса;
* статический класс *не может содержать сущности с атрибутами доступа****protected****и protected****internal***(да они и не нужны, все равно наследование запрещено);
* класс *не может иметь перегруженных операций (операторов).*

3. Каково назначение перегрузки операторов?

Мы можем использовать некоторые операции, типа сложения, разности и прочее, с примитивными типами данных. Но с объектами не можем. Поэтому есть возможность перегружать операторы для сохранения простого синтаксиса. То есть вместо использования метода, куда мы в качестве параметров передадим 2 объекта, мы напишем, например *obj1 + obj2*.

4. Как используется ключевое слово operator?

При **перегрузке операторов** мы используем специальный синтаксис:

public static возвращаемый\_тип operator оператор(параметры){ }

где вместо названия метода идет ключевое слово **operator** и собственно сам оператор.

Это же слово мы используем при **перегрузке операции преобразования типов**.

5. Какие операции нельзя перегружать в C#?

Следующая таблица содержит сведения о возможности перегрузки операторов C#:

Следующая таблица содержит сведения о возможности перегрузки операторов C#:

| **ПЕРЕГРУЖАЕМЫЕ ОПЕРАТОРЫ** | |
| --- | --- |
| **Операторы** | **Возможность перегрузки** |
| [+x](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/operators/arithmetic-operators#unary-plus-and-minus-operators), [-x](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/operators/arithmetic-operators#unary-plus-and-minus-operators), [!x](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/operators/boolean-logical-operators#logical-negation-operator-), [~x](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/operators/bitwise-and-shift-operators#bitwise-complement-operator-), [++](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/operators/arithmetic-operators#increment-operator-), [--](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/operators/arithmetic-operators#decrement-operator---), [true](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/operators/true-false-operators), [false](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/operators/true-false-operators) | Эти унарные операторы могут быть перегружены. |
| [x + y](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/operators/addition-operator), [x - y](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/operators/subtraction-operator), [x \* y](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/operators/arithmetic-operators#multiplication-operator-), [x / y](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/operators/arithmetic-operators#division-operator-), [x % y](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/operators/arithmetic-operators#remainder-operator-), [x & y](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/operators/boolean-logical-operators#logical-and-operator-), [x | y](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/operators/boolean-logical-operators#logical-or-operator-), [x ^ y](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/operators/boolean-logical-operators#logical-exclusive-or-operator-), [x << y](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/operators/bitwise-and-shift-operators#left-shift-operator-), [x >> y](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/operators/bitwise-and-shift-operators#right-shift-operator-), [x == y](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/operators/equality-operators#equality-operator-), [x != y](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/operators/equality-operators#inequality-operator-), [x < y](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/operators/comparison-operators#less-than-operator-), [x > y](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/operators/comparison-operators#greater-than-operator-), [x <= y](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/operators/comparison-operators#less-than-or-equal-operator-), [x >= y](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/operators/comparison-operators#greater-than-or-equal-operator-) | Эти бинарные операторы могут быть перегружены. Некоторые операторы должны перегружаться парами; дополнительные сведения см. в примечаниях после этой таблицы. |
| [x && y](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/operators/boolean-logical-operators#conditional-logical-and-operator-), [x || y](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/operators/boolean-logical-operators#conditional-logical-or-operator-) | Условный логический оператор не может быть перегружен. При этом, если тип с перегруженными [операторами true и false](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/operators/true-false-operators) также перегружает оператор & или |, оператор && или ||, соответственно, может быть применен для операндов этого типа. Дополнительные сведения см. в разделе [Пользовательские условные логические операторы](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/language-specification/expressions#user-defined-conditional-logical-operators) в [Спецификации языка C#](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/language-specification/introduction). |
| [a[i]](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/operators/member-access-operators#indexer-operator-), [a?[i]](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/operators/member-access-operators#null-conditional-operators--and-) | *Доступ к элементам не считается перегружаемым оператором*, но вы можете определить [индексатор](https://docs.microsoft.com/ru-ru/dotnet/csharp/programming-guide/indexers/). |
| [(T)x](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/operators/type-testing-and-cast#cast-expression) | Оператор приведения невозможно перегрузить, но можно определить пользовательские преобразования типа и выполнять его с помощью выражения приведения. Дополнительные сведения см. в разделе [Операторы пользовательского преобразования](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/operators/user-defined-conversion-operators). |
| [+=](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/operators/arithmetic-operators#compound-assignment), [-=](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/operators/arithmetic-operators#compound-assignment), [\*=](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/operators/arithmetic-operators#compound-assignment), [/=](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/operators/arithmetic-operators#compound-assignment), [%=](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/operators/arithmetic-operators#compound-assignment), [&=](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/operators/boolean-logical-operators#compound-assignment), [|=](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/operators/boolean-logical-operators#compound-assignment), [^=](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/operators/boolean-logical-operators#compound-assignment), [<<=](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/operators/bitwise-and-shift-operators#compound-assignment), [>>=](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/operators/bitwise-and-shift-operators#compound-assignment) | Составные операторы присваивания не могут быть перегружены явным образом. Однако при перегрузке бинарного оператора соответствующий составной оператор присваивания (если таковой имеется) также неявно перегружается. Например, += вычисляется с помощью +, который может быть перегружен. |
| [^x](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/operators/member-access-operators#index-from-end-operator-), [x = y](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/operators/assignment-operator), [x.y](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/operators/member-access-operators" \l "member-access-expression-), [x?.y](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/operators/member-access-operators" \l "null-conditional-operators--and-), [c ? t : f](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/operators/conditional-operator), [x ?? y](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/operators/null-coalescing-operator), [x ??= y](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/operators/null-coalescing-operator), [x..y](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/operators/member-access-operators" \l "range-operator-), [x->y](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/operators/pointer-related-operators#pointer-member-access-operator--), [=>](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/operators/lambda-operator), [f(x)](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/operators/member-access-operators#invocation-expression-), [as](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/operators/type-testing-and-cast#as-operator), [await](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/operators/await), [checked](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/keywords/checked), [unchecked](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/keywords/unchecked), [default](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/operators/default), [delegate](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/operators/delegate-operator), [is](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/operators/type-testing-and-cast#is-operator), [nameof](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/operators/nameof), [new](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/operators/new-operator), [sizeof](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/operators/sizeof), [stackalloc](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/operators/stackalloc), [switch](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/operators/switch-expression), [typeof](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/operators/type-testing-and-cast" \l "typeof-operator), [with](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/operators/with-expression) | Эти операторы не могут быть перегружены. |

6. Можно ли перегрузкой отменить очередность выполнения операции?

Приоритет операций диктуется грамматикой языка и наследуется из грамматики языка. Грамматике языка не интересно, встроенные ли это операторы или перегруженные.

Например, имеется три объекта одного класса с перегруженными операциями + и \*. Сохранят ли эти операции приоритет или нет?

*rezult = a + b \* c.*

Где a, b, c - объекты одного класса. Какая будет последовательность выполненных операций и почему?

Ответ: Если считать, что все операторы в данном выражении перегружены, то сначала будет выполнено умножение b \* c, затем сложение a + ..., и затем - присваивание result = ... .

7. Истинно ли следующее утверждение: операция >= может быть перегружена.

(5 вопрос)

8. Сколько аргументов требуется для определения перегруженной унарной операции?

Бинарные операторы принимают два параметра, унарные - один параметр. И в любом случае один из параметров должен представлять тот тип - класс или структуру, в котором определяется оператор.

9. Можно ли перегружать операцию []?

(5 вопрос)

10. Можно ли перегружать операцию ->?

(5 вопрос)

11. Приведите пример оператора приведения типа

using System;

namespace ConsoleApp1

{

class Program

{

class A

{

private int[] num = new int[] { 1, 2, 45 };

**static public explicit operator string(A el)**

**{**

**string str = "";**

**foreach(int i in el.num)**

**{**

**str += i.ToString();**

**}**

**return str;**

**}**

}

static void Main(string[] args)

{

A obj = new A();

Console.WriteLine((string)obj);

}

}

}

12. Что такое метод расширения? Как и где его можно использовать?

(где-то в заданиях)

13. Пусть дан фрагмент кода определения оператора преобразования типа. Определить форму преобразования.

public static implicit operator Point2D(Point3D a)

{/\* код\*/;}

После модификаторов **public static** идет ключевое слово **explicit** (если преобразование явное, то есть нужна операция приведения типов) или **implicit** (если преобразование неявное). Затем идет ключевое слово **operator** и далее возвращаемый тип, в который надо преобразовать объект. В скобках в качестве параметра передается объект, который надо преобразовать.

Таким образом, преобразование из типа Point3D в тип Point2D неявное.

14. Выберите верное утверждение. Метод расширения может:

1) получать доступ к public членам расширяемого класса

2) получать доступ к protected членам расширяемого класса ← доступ к protected членам имеют производные классы и сам класс, где этот член объявлен

3) получать доступ к internal членам расширяемого класса ← доступ к internal членам доступен в пределах сборки

4) быть объявлен в любом классе ← только в статическом

5) быть без параметров ← обязательно наличие первого параметра *this название\_типа имя\_переменной*

15. Выберите из списка неверное правило перегрузки операторов для C#.

1) префиксные операции ++ и – – перегружаются парами

2) операции сравнения перегружаются парами: == и != ; < и >;<= и >=

3) перегруженные операции обязаны возвращать значения

4) должны объявляться как protected ← должно включать public и модификатор static

5) true и false можно перегружать

