Оглавление

[Оглавление 1](#_Toc165188053)

[Основы 5](#_Toc165188054)

[Литералы 5](#_Toc165188055)

[Типы данных 5](#_Toc165188056)

[Консольный ввод-вывод 5](#_Toc165188057)

[Логические операции 6](#_Toc165188058)

[Операции присваивания 6](#_Toc165188059)

[Массивы 6](#_Toc165188060)

[Одномерные массивы 6](#_Toc165188061)

[Многомерные массивы 7](#_Toc165188062)

[Массив массивов 7](#_Toc165188063)

[Методы 7](#_Toc165188064)

[Передача параметров по ссылке. Выходные параметры 8](#_Toc165188065)

[Модификатор ref 8](#_Toc165188066)

[Модификатор out 8](#_Toc165188067)

[Модификатор in 8](#_Toc165188068)

[Модификатор ref readonly (c C#12) 8](#_Toc165188069)

[Массив параметров и ключевое слово params 9](#_Toc165188070)

[Локальные функции 9](#_Toc165188071)

[Конструкция switch 9](#_Toc165188072)

[Перечисления enum 10](#_Toc165188073)

[Классы 11](#_Toc165188074)

[Создание конструкторов 11](#_Toc165188075)

[Цепочка вызовов конструкторов 11](#_Toc165188076)

[Первичные конструкторы (начиная с версии C#12) 11](#_Toc165188077)

[Инициализаторы объектов 11](#_Toc165188078)

[Деконструкторы 12](#_Toc165188079)

[Структуры 12](#_Toc165188080)

[Инициализация с помощью конструктора 12](#_Toc165188081)

[Цепочка конструкторов 12](#_Toc165188082)

[Первичный конструктор 13](#_Toc165188083)

[Инициализатор структуры 13](#_Toc165188084)

[Копирование структуры с помощью with 13](#_Toc165188085)

[Модификаторы доступа 13](#_Toc165188086)

[Свойства 14](#_Toc165188087)

[Определение свойств 14](#_Toc165188088)

[Свойства только для чтения и записи 14](#_Toc165188089)

[Модификаторы доступа 14](#_Toc165188090)

[Автоматические свойства 15](#_Toc165188091)

[Блок init (начиная с версии C# 9.0) 15](#_Toc165188092)

[Сокращенная запись свойств 15](#_Toc165188093)

[Модификатор required (добавлен в C# 11) 15](#_Toc165188094)

[Статические члены и модификатор static 16](#_Toc165188095)

[Статические поля 16](#_Toc165188096)

[Статические свойства 16](#_Toc165188097)

[Статические методы 16](#_Toc165188098)

[Статический конструктор 16](#_Toc165188099)

[Статические классы 16](#_Toc165188100)

[Константы, поля и структуры для чтения 16](#_Toc165188101)

[Константы класса 16](#_Toc165188102)

[Поля для чтения и модификатор readonly 17](#_Toc165188103)

[Сравнение констант 17](#_Toc165188104)

[Структуры для чтения 17](#_Toc165188105)

[Null и ссылочные типы 17](#_Toc165188106)

[Оператор ! (null-forgiving operator) 17](#_Toc165188107)

[Null и значимые типы 17](#_Toc165188108)

[Проверка на null, операторы ?. и ?? 18](#_Toc165188109)

[Оператор ?? 18](#_Toc165188110)

[Оператор условного null ( ?. ) 18](#_Toc165188111)

[Псевдонимы типов и статический импорт 18](#_Toc165188112)

[Псевдонимы 18](#_Toc165188113)

[Статический импорт 19](#_Toc165188114)

[ООП 19](#_Toc165188115)

[Наследование 19](#_Toc165188116)

[Ключевое слово base 19](#_Toc165188117)

[Преобразование типов 20](#_Toc165188118)

[Восоходящее преобразование. Upcasting 20](#_Toc165188119)

[Нисходящее преобразование. Downcasting 20](#_Toc165188120)

[Способы преобразований 20](#_Toc165188121)

[Виртуальные методы и свойства 20](#_Toc165188122)

[Ключевое слово base 21](#_Toc165188123)

[Переопределение свойств 21](#_Toc165188124)

[Запрет переопределения методов 21](#_Toc165188125)

[Скрытие методов и свойств 21](#_Toc165188126)

[Различие переопределения и скрытия методов 22](#_Toc165188127)

[Переопределение 22](#_Toc165188128)

[Скрытие 22](#_Toc165188129)

[Абстрактные классы и члены классов 22](#_Toc165188130)

[Класс System.Object и его методы 23](#_Toc165188131)

[Обобщения (generics) 23](#_Toc165188132)

[Статические поля обобщенных классов 23](#_Toc165188133)

[Обобщенные методы 23](#_Toc165188134)

[Ограничения обобщений 24](#_Toc165188135)

[Ограничения методов 24](#_Toc165188136)

[Ограничения обобщений в типах 24](#_Toc165188137)

[Типы ограничений и стандартные ограничения 24](#_Toc165188138)

[Использование нескольких универсальных параметров 24](#_Toc165188139)

[Наследование обобщенных типов 24](#_Toc165188140)

[Исключения 25](#_Toc165188141)

[Конструкция try..catch..finally 25](#_Toc165188142)

[Фильтры исключений 25](#_Toc165188143)

[Типы исключений. Класс Exception 25](#_Toc165188144)

[Генерация исключения и оператор throw 26](#_Toc165188145)

[Делегаты, лямбды и события 26](#_Toc165188146)

[Делегаты 26](#_Toc165188147)

[Добавление методов в делегат 26](#_Toc165188148)

[Объединение делегатов 27](#_Toc165188149)

[Вызов делегата с помощью метода Invoke() 27](#_Toc165188150)

[Обобщенные делегаты (Generic) 27](#_Toc165188151)

[Делегаты как параметры методов 28](#_Toc165188152)

[Возвращение делегатов из метода 28](#_Toc165188153)

[Анонимные методы 28](#_Toc165188154)

[Лямбды 29](#_Toc165188155)

[Параметры лямбды 29](#_Toc165188156)

[Возвращение результата 30](#_Toc165188157)

[Добавление и удаление действий в лямбда-выражении, метод Invoke 30](#_Toc165188158)

[Лямбда-выражение как аргумент метода 30](#_Toc165188159)

[Лямбда-выражение как результ метода 30](#_Toc165188160)

[События 31](#_Toc165188161)

[Определение и вызов событий 31](#_Toc165188162)

[Добавление обработчика события 31](#_Toc165188163)

[Управление обработчиками (add, remove) 32](#_Toc165188164)

[Ковариантность и контравариантность делегатов 32](#_Toc165188165)

[Ковариантность 32](#_Toc165188166)

[Контрвариантность 32](#_Toc165188167)

[Ковариантность и контравариантность в обобщенных делегатах 33](#_Toc165188168)

[Делегаты Action, Predicate и Func 33](#_Toc165188169)

[Action 33](#_Toc165188170)

[Predicate 33](#_Toc165188171)

[Func 34](#_Toc165188172)

[Замыкания 34](#_Toc165188173)

[Реализация с помощью лямбда-выражений 34](#_Toc165188174)

[Применение параметров 34](#_Toc165188175)

[Интерфейсы 35](#_Toc165188176)

[Определение интерфейсов 35](#_Toc165188177)

[Применение интерфейсов 35](#_Toc165188178)

[Множественная реализация интерфейсов 36](#_Toc165188179)

[Явная реализация интерфейсов 36](#_Toc165188180)

[Модификаторы доступа 37](#_Toc165188181)

[Реализация интерфейсов в базовых и производных классах 37](#_Toc165188182)

[Наследование интерфейсов 39](#_Toc165188183)

[Интерфейсы как ограничения обобщений 39](#_Toc165188184)

[Обобщенные интерфейсы 40](#_Toc165188185)

[Копирование объектов. Интерфейс ICloneable 40](#_Toc165188186)

[Поверхностное копирование 40](#_Toc165188187)

[Глубокое копирование 40](#_Toc165188188)

[Сортировка объектов. Интерфейс IComparable 40](#_Toc165188189)

[Применение компаратора 41](#_Toc165188190)

[Ковариантность и контравариантность обобщенных интерфейсов 41](#_Toc165188191)

[Ковариантные интерфейсы 42](#_Toc165188192)

[Контравариантные интерфейсы 42](#_Toc165188193)

[Дополнительные возможности ООП 42](#_Toc165188194)

[Определение операторов 42](#_Toc165188195)

[Определение инкремента и декремента 43](#_Toc165188196)

[Определение операций true и false 43](#_Toc165188197)

[Перегрузка операций преобразования типов 43](#_Toc165188198)

[Индексаторы 44](#_Toc165188199)

[Переменные-ссылки и возвращение ссылки 44](#_Toc165188200)

[Переменная-ссылка 45](#_Toc165188201)

[Ссылка как результат функции 45](#_Toc165188202)

[Методы расширения 45](#_Toc165188203)

[Частичные классы и методы 46](#_Toc165188204)

[Частичные методы 46](#_Toc165188205)

[Анонимные типы 46](#_Toc165188206)

[Инициализаторы с проекцией 46](#_Toc165188207)

Основы

## Литералы

0b11 **-** 3 число в двоичной форме

0x0A **-** 10 число в шестнадцатеричной форме

3.2e3 **(**3.2E3**)** **-** 3.2 **\*** 10**^**3

'\x78' **-** x символ в ASCII

'\u0420' **-** P символ в Unicode

## Типы данных

bool**:** хранит значение **true** или **false** **(**логические литералы**).**

byte**:** хранит целое число от 0 до 255 и занимает 1 байт**.**

sbyte**:** хранит целое число от **-**128 до 127 и занимает 1 байт**.**

short**:** хранит целое число от **-**32768 до 32767 и занимает 2 байта**.**

ushort**:** хранит целое число от 0 до 65535 и занимает 2 байта**.**

int**:** хранит целое число от **-**2147483648 до 2147483647 и занимает 4 байта**.**

uint**:** хранит целое число от 0 до 4294967295 и занимает 4 байта**.** Суффикс u **(**U**)** **-** 10U

long**:** хранит целое число от –9 223 372 036 854 775 808 до 9 223 372 036 854 775 807 и занимает 8 байт**.** Суффикс l **(**L**)** **-** 10L

ulong**:** хранит целое число от 0 до 18 446 744 073 709 551 615 и занимает 8 байт**.** Суффикс ul **(**UL**)** **-** 10L

float**:** хранит число с плавающей точкой от **-**3.4e38 до 3.4e38 и занимает 4 байта**.** Суффикс f **(**F**)** **-** 3.14F

double**:** хранит число с плавающей точкой от ±5.0e-324 до ±1.7e308 и занимает 8 байта**.**

decimal**:** хранит десятичное дробное число**.** Если употребляется без десятичной запятой**,** имеет значение от ±1.0**\***10**-**28 до ±7.9228**\***1028**,** может хранить 28 знаков после запятой и занимает 16 байт**.** Суффикс m **(**M**)** **-** 3.14M

char**:** хранит одиночный символ в кодировке Unicode и занимает 2 байта**.**

string**:** хранит набор символов Unicode**.**

**object:** может хранить значение любого типа данных и занимает 4 байта на 32**-**разрядной платформе и 8 байт на 64**-**разрядной платформе**.**

## Консольный ввод-вывод

Convert**.**ToInt32**()** **(**преобразует к типу int**)**

Convert**.**ToDouble**()** **(**преобразует к типу double**)**

Convert**.**ToDecimal**()** **(**преобразует к типу decimal**)**

Console**.**Write**(**"Hello"**);** // Hello

int age **=** Convert**.**ToInt32**(**Console**.**ReadLine**());**

double height **=** Convert**.**ToDouble**(**Console**.**ReadLine**());**

Console**.**WriteLine**(**$"Age = {age} years old, height = {height} meters"**);**

Console**.**WriteLine**(**"Age = {0} years old, height = {1} meters"**,** age**,** height**);**

## Логические операции

int x **=** 2**;** // 010

int y **=** 5**;** //101

x**&**y **-** 0 // 000 AND

x**|**y **-** 7 // 111 OR

x**^**y **-** 7 // 111 XOR

**~**x **-** **-**2 // 1...010 старший бит равен 1

x**<<**1 **-** 4 // 100

x**>>**1 **-** 1 // 001

## Операции присваивания

int a**,** b**,** c**;**

a **=** b **=** c **=** 34**;**

**+=**

**-=**

**\*=**

**/=**

**%=**

**&=**

**|=**

**^=**

**>>=**

**<<=**

## Массивы

### Одномерные массивы

int**[]** nums **=** **new** int**[**4**];**

int**[]** nums **=** **new** int**[**4**]** **{** 1**,** 2**,** 3**,** 5 **};**

int**[]** nums **=** **new** int**[]** **{** 1**,** 2**,** 3**,** 5 **};**

int**[]** nums **=** **new[]** **{** 1**,** 2**,** 3**,** 5 **};**

int**[]** nums **=** **{** 1**,** 2**,** 3**,** 5 **};**

c C#12

int**[]** nums **=** **[** 1**,** 2**,** 3**,** 5 **];**

int**[]** nums **=** **[];** // пустой массив

nums**[**0**]** **-** 1

nums**[^**1**]** **-** 5m

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

**{**

Console**.**WriteLine**(**nums**[**i**]);**

**}**

**foreach(**var n **in** nums**)**

Console**.**WriteLine**(**n**);**

### Многомерные массивы

int**[,]** nums**;**

int**[,]** nums **=** **new** int**[**2**,** 3**];**

int**[,]** nums **=** **new** int**[**2**,** 3**]** **{** **{** 0**,** 1**,** 2 **},** **{** 3**,** 4**,** 5 **}** **};**

int**[,]** nums **=** **new** int**[,]** **{** **{** 0**,** 1**,** 2 **},** **{** 3**,** 4**,** 5 **}** **};**

int**[,]** nums **=** **new** **[,]{** **{** 0**,** 1**,** 2 **},** **{** 3**,** 4**,** 5 **}** **};**

int**[,]** nums **=** **{** **{** 0**,** 1**,** 2 **},** **{** 3**,** 4**,** 5 **}** **};**

**foreach** **(**int i **in** numbers**)**

Console**.**Write**(**$"{i} "**);** // 012345

**for** **(**val i **=** 0**;** i **<=** nums**.**GetUpperBound**(**0**);** i**++)**

**for** **(**val j **=** 0**;** j **<=** nums**.**GetUpperBound**(**1**);** j**++)**

Console**.**Write**(**nums**[**i**,**j**]);** // 012345

### Массив массивов

int**[][]** nums **=** **new** int**[**3**][];**

nums**[**0**]** **=** **new** int**[**2**]** **{** 1**,** 2 **};** // выделяем память для первого подмассива

nums**[**1**]** **=** **new** int**[**3**]** **{** 1**,** 2**,** 3 **};** // выделяем память для второго подмассива

nums**[**2**]** **=** **new** int**[**5**]** **{** 1**,** 2**,** 3**,** 4**,** 5 **};** // выделяем память для третьего подмассива

int**[][]** nums **=** **{**

**new** int**[]** **{** 1**,** 2 **},**

**new** int**[]** **{** 1**,** 2**,** 3 **},**

**new** int**[]** **{** 1**,** 2**,** 3**,** 4**,** 5 **}**

**};**

**foreach** **(**int**[]** row **in** nums**)**

**foreach** **(**int num **in** row**)**

Console**.**Write**(**num**);** // 12345

**for** **(**val i **=** 0**;** i **<** nums**.**Length**;** i**++)**

**for** **(**val j **=** 0**;** j **<** nums**[**i**].**Length**;** j**++)**

Console**.**Write**(**nums**[**i**][**j**]);** \\ 12345

## Методы

int Sum**(**int x**,** int y **=** 0**)**

**{**

**return** x **+** y**;**

**}**

int Sum**(**int x**,** int y **=** 0**)** **=>** x **+** y**;**

Sum**(**4**)** **-** 4

Sum**(**4**,** 5**)** **-** 9

Sum**(**y**:**10**,** x**:**5**)** **-** 15

## Передача параметров по ссылке. Выходные параметры

### Модификатор ref

При передаче значений параметрам по ссылке метод получает адрес переменной в памяти**.**

Параметр с модификатором **ref** обязан быть инициализирован перед передачей в метод**.**

void Increment**(ref** int n**)**

**{**

n**++;**

**}**

int number **=** 5**;**

Increment**(ref** number**)** // number = 6

### Модификатор out

Методы**,** использующие параметры **out,** обязательно должны присваивать им определенное значение**,** даже если оно им уже присвоено где**-**то в коде перед вызовом метода **(**в отличие от **ref).**

void Sum**(**int x**,** int y**,** **out** int sum**,** **out** int mul**)**

**{**

sum **=** x **+** y**;**

mul **=** x **\*** y**;**

**}**

int numSum**;**

Sum**(**10**,** 5**,** **out** numSum**,** **out** int numMul**);** // numSum = 15, numMul = 50 (переменную можно определить непосредственно при вызове метода)

### Модификатор in

Модификатор **in** указывает**,** что данный параметр будет передаваться в метод по ссылке**,** однако внутри метода его значение параметра нельзя будет изменить**.**

int Sum**(in** int x**,** **in** int y **=** 0**)** **=>** x **+** y**;**

### Модификатор ref readonly (c C#12)

**ref-**параметры только для чтения**.**

void Increment**(ref** **readonly** int n**)**

**{**

// n++; // нельзя, иначе будет ошибка компиляции

**}**

int number **=** 5**;**

Increment**(ref** number**);**

## Массив параметров и ключевое слово params

int Sum**(params** int**[]** numbers**)**

**{**

int result **=** 0**;**

**foreach** **(**var n **in** numbers**)**

result **+=** n**;**

**return** result**;**

**}**

int**[]** nums **=** **{** 1**,** 2**,** 3**,** 4**,** 5**};**

Sum**(**nums**);**

Sum**(**1**,** 2**,** 3**,** 4**);**

Sum**();**

## Локальные функции

Локальные функции представляют функции**,** определенные внутри других методов**.**

Локальная функция**,** как правило**,** содержит действия**,** которые применяются только в рамках ее метода**.**

bool AreEqual**(**int**[]** nums1**,** int**[]** nums2**)**

**{**

int sum1 **=** Sum**(**nums1**);**

int sum2 **=** Sum**(**nums2**);**

**return** sum1 **==** sum2**;**

int Sum**(**int**[]** nums**)** **=>** nums**.**Sum**();**

// static int Sum(int[] nums) => nums.Sum();

// static функции не могут обращаться к переменным окружения, то есть метода,

// в котором статическая функция определена в отличие от нестатических функций.

**}**

## Конструкция switch

**switch** **(**number**)**

**{**

**case** 1**:**

Console**.**WriteLine**(**"case 1"**);**

**goto** **case** 5**;** // переход к case 5

**case** 3**:**

Console**.**WriteLine**(**"case 3"**);**

**break;**

**case** 5**:**

Console**.**WriteLine**(**"case 5"**);**

**break;**

**default:**

Console**.**WriteLine**(**"default"**);**

**break;**

**}**

int DoOperation**(**int op**,** int a**,** int b**)**

**{**

**switch** **(**op**)**

**{**

**case** 1**:** **return** a **+** b**;**

**case** 2**:** **return** a **-** b**;**

**case** 3**:** **return** a **\*** b**;**

**default:** **return** 0**;**

**}**

**}**

int result **=** op **switch** **{**

1 **=>** a **+** b**,**

2 **=>** a **-** b**,**

3 **=>** a **\*** b**,**

\_ **=>** 0

**};**

## Перечисления enum

enum DayTime

**{**

Morning**,**

Afternoon**,**

Evening**,**

Night

**}**

DayTime**.**Morning **-** обращение к значению Morning

enum Time **:** byte

**{**

Morning**,** // 0

Afternoon**,** // 1

Evening**,** // 2

Night // 3

**}**

**(**int**)** DayTime**.**Night **-** 3

enum DayTime

**{**

Morning **=** 3**,** // каждый следующий элемент по умолчанию увеличивается на единицу

Afternoon**,** // 4

Evening**,** // 5

Night **=** Morning // 3

**}**

Классы

## Создание конструкторов

class Person

**{**

**public** string name**;**

**public** Person**()** **{** name **=** "Неизвестно"**}** // 1 конструктор

**public** Person**(**string name**)** **{** **this.**name **=** name **}** // 2 конструктор

**}**

Person tom **=** **new** **();** // аналогично new Person();

Person bob **=** **new** **(**"Bob"**);** // аналогично new Person("Bob");

### Цепочка вызовов конструкторов

class Person

**{**

**public** string name**;**

**public** Person**()** **:** **this(**"Неизвестно"**)** **{** **}** // первый конструктор

**public** Person**(**string name**)** **{** **this.**name **=** name **}** // второй конструктор

**}**

### Первичные конструкторы (начиная с версии C#12)

**public** class Person**(**string name**,** int age**)**

**{**

**public** Person**(**string name**)** **:** **this(**name**,** 18**)** **{** **}**

**}**

Person tom **=** **new** **(**"Tom"**,** 34**);**

Person bob **=** **new** **(**"Bob"**);**

### Инициализаторы объектов

class Person

**{**

**public** string name**;**

**public** Company company**;**

**public** Person**()**

**{**

name **=** "Undefined"**;**

company **=** **new** Company**();**

**}**

**}**

class Company

**{**

**public** string title **=** "Unknown"**;**

**}**

Person tom **=** **new** Person**{** name **=** "Tom"**,** company **=** **{** title **=** "Microsoft"**}** **};**

### Деконструкторы

class Person

**{**

string name**;**

int age**;**

**public** Person**(**string name**,** int age**)**

**{**

**this.**name **=** name**;**

**this.**age **=** age**;**

**}**

**public** void Deconstruct**(out** string personName**,** **out** int personAge**)**

**{**

personName **=** name**;**

personAge **=** age**;**

**}**

**}**

Person person **=** **new** Person**(**"Tom"**,** 33**);**

**(**string name**,** int age**)** **=** person**;**

// То же самое, что и:

// string name; int age;

// person.Deconstruct(out name, out age);

// Если нужна только одна переменная, можно так:

// (\_, int age) = person;

Console**.**WriteLine**(**name**);** // Tom

Console**.**WriteLine**(**age**);** // 33

## Структуры

### Инициализация с помощью конструктора

struct Person

**{**

// элементы структуры:

// поля и методы

int age **=** 10**;** // инициализация полей значениями по умолчанию - доступна с C#10

**public** Person**()** // начиная с C#10 мы можем определить свой конструктор без параметров

**{**

age **=** 20**;**

**}**

**}**

Person tom **=** **new** Person**();** // вызов конструктора

Person bob **=** **new();**

### Цепочка конструкторов

struct Person

**{**

**public** int age**;**

**public** Person**()** **:** **this(**10**)**

**{** **}**

**public** Person**(**string name**)** **:** **this(**name**,** 1**)**

**{** **}**

**public** Person**(**int age**)**

**{**

**this.**age **=** age**;**

**}**

**}**

var tom **=** **new** Person**(**20**);**

### Первичный конструктор

**public** struct Person**(**string name**,** int age**)**

**{**

**public** Person**(**string name**)** **:** **this(**name**,** 18**)** **{** **}**

**}**

### Инициализатор структуры

struct Person

**{**

**public** string name**;**

**public** int age**;**

**}**

Person tom **=** **new** Person **{** name **=** "Tom"**,** age **=** 22 **};**

### Копирование структуры с помощью with

Person tom **=** **new** Person **{** name **=** "Tom"**,** age **=** 22 **};**

Person bob **=** tom with **{** name **=** "Bob" **};**

// объект bob получает все значения объекта tom,

// а затем после оператора with в фигурных скобках указывается поля со значениями,

// которые мы хотим изменить.

## Модификаторы доступа

**private:** закрытый или приватный компонент класса или структуры**.** Приватный компонент доступен только в рамках своего класса или структуры**.**

**private** **protected:** компонент класса доступен из любого места в своем классе или в производных классах**,** которые определены в той же сборке**.**

**file:** добавлен в версии C# 11 и применяется к типам**,** например**,** классам и структурам**.** Класс или структура с такми модификатором доступны только из текущего файла кода**.**

**protected:** такой компонент класса доступен из любого места в своем классе или в производных классах**.** При этом производные классы могут располагаться в других сборках**.**

**internal:** компоненты класса или структуры доступен из любого места кода в той же сборке**,** однако он недоступен для других программ и сборок**.**

**protected** **internal:** совмещает функционал двух модификаторов **protected** и **internal.** Такой компонент класса доступен из любого места в текущей сборке и из производных классов**,** которые могут располагаться в других сборках**.**

**public:** публичный**,** общедоступный компонент класса или структуры**.** Такой компонент доступен из любого места в коде**,** а также из других программ и сборок**.**

## Свойства

### Определение свойств

**private** string name **=** "Undefined"**;**

**public** string Name

**{**

**get**

**{**

**return** name**;** // возвращаем значение свойства

**}**

**set**

**{**

name **=** **value;** // устанавливаем новое значение свойства

**}**

**}**

person**.**Name **=** "Tom"**;**

### Свойства только для чтения и записи

// свойство только для записи

**public** int Age

**{**

**set** **{** age **=** **value;** **}**

**}**

// свойство только для чтения

**public** string Name

**{**

**get** **{** **return** name**;** **}**

**}**

### Модификаторы доступа

**public** string Name

**{**

**get** **{** **return** name**;** **}**

**private** **set** **{** name **=** **value;**

**}**

*Ограничения****:***

**-** Модификатор для блока **set** или **get** можно установить**,** если свойство имеет оба блока **(**и **set,** и **get);**

**-** Только один блок **set** или **get** может иметь модификатор доступа**,** но не оба сразу**;**

**-** Модификатор доступа блока **set** или **get** должен быть более ограничивающим**,** чем модификатор доступа свойства**.**

### Автоматические свойства

string Name **{** **get;** **set;** **}**

int Age **{** **get;** **set;** **}** **=** 37**;** // значение по умолчанию

string City **{** **private** **set;** **get;}** // модификатор доступа

### Блок init (начиная с версии C# 9.0)

После инициализации значений свойств их значения изменить нельзя **-** они доступны только для чтения**.**

*1 способ****:***

string Name **{** **get;** init**;** **}** **=** "Undefined"**;** // изменить нельзя

*2 способ****:***

Person**(**string name**)** **=>** Name **=** name**;** // инициализация в конструкторе

string Name **{** **get;** init**;** **}**

*3 способ****:***

class Person

**{**

string Name **{** **get;** init**;** **}** **=** ""**;**

**}**

Person person **=** **new()** **{** Name **=** "Bob"**};**

*4 способ****:***

string name **=** ""**;**

string Name // свойство Name управляет полем для чтения name

**{**

**get** **{** **return** name**;** **}**

init

**{**

name **=** **value;**

**}**

**}**

### Сокращенная запись свойств

string name**;**

string Name

**{**

**get** **=>** name**;**

**set** **=>** name **=** **value;**

**}**

string name**;**

**public** string Name **=>** name**;** // эквивалентно public string Name { get { return name; } }

### Модификатор required (добавлен в C# 11)

Модификатор required указывает**,** что поле или свойство с этим модификатором обязательно должно быть инициализировано**.**

Person**(**string name**)**

**{**

Name **=** name**;**

**}**

required string Name **{** **get;** **set;** **}** // инициализировано в конструкторе

required int Age **{** **get;** **set;** **}** **=** 22**;**

## Статические члены и модификатор static

### Статические поля

Статические поля хранят состояние всего класса **/** структуры**.**

### Статические свойства

static int age **=** 65**;**

static int Age

**{**

**get** **{** **return** age**;** **}**

**set** **{** age **=** **value;** **}**

**}**

### Статические методы

Статические методы могут обращаться только к статическим членам класса**.**

### Статический конструктор

**-** Статические конструкторы не должны иметь модификатор доступа и не принимают параметров**;**

**-** Как и в статических методах**,** в статических конструкторах нельзя использовать ключевое слово **this** для ссылки на текущий объект класса и можно обращаться только к статическим членам класса**;**

**-** Статические конструкторы нельзя вызвать в программе вручную**.** Они выполняются автоматически при самом первом создании объекта данного класса или при первом обращении к его статическим членам **(**если таковые имеются**).**

### Статические классы

Статические классы объявляются с модификатором static и могут содержать только статические поля**,** свойства и методы**.**

## Константы, поля и структуры для чтения

### Константы класса

Значение константы устанавливается один раз непосредственно при её объявлении и впоследствии не может быть изменено**.**

Константы хранят некоторые данные**,** которые относятся не к одному объекту**,** а ко всему классу в целом**.**

Для обращения к константам применяется не имя объекта**,** а имя класса**:**

class Person

**{**

const string type **=** "Person"**;**

**}**

Console**.**WriteLine**(**Person**.**type**);** // Person

### Поля для чтения и модификатор readonly

Таким полям можно присвоить значение либо при непосредственно при их объявлении**,** либо в конструкторе**.**

Их значение нельзя изменить**.**

**readonly** string name **=** "Tom"**;**

### Сравнение констант

**-** const должны быть определены во время компиляции**,** а **readonly** могут быть определены во время выполнения программы в конструкторе**.**

**-** const в отличие от **readonly** не могут использовать модификатор static**,** так как уже неявно являются статическими**.**

### Структуры для чтения

Все их поля должны быть также полями для чтения**:**

**readonly** struct Person

**{**

**readonly** int age **=** 5**;**

**readonly** string Name **{** **get;** **}** **=** "Tom"**;** // указывать readonly необязательно

**}**

## Null и ссылочные типы

#nullable disable

string text **=** **null;** // здесь nullable-контекст не действует

#nullable restore

string**?** name **=** **null;** // здесь nullable-контекст снова действует

### Оператор ! (null-forgiving operator)

string**?** name **=** "Tom"

Console**.**WriteLine**(**name**!);**

## Null и значимые типы

int**?** val **=** **null;** // nullable int тип

// то же самое, что и:

Nullable**<**int**>** val **=** **null;**

Структура Nullable**<**T**>** имеет два свойства**:**

**-** Value **-** значение объекта**;**

**-** HasValue**:** возвращает **true,** если объект хранит некоторое значение**,** и **false,** если объект равен **null.**

**...** и метод**:**

**-** GetValueOrDefault**():** возвращает значение переменной**/**параметра**,** если они не равны **null.** Если они равны **null,** то возвращается значение по умолчанию**.** Значение по умолчанию можно передать в метод**.**

int**?** number **=** **null;** // если значения нет, метод возвращает значение по умолчанию

Console**.**WriteLine**(**number**.**GetValueOrDefault**());** // 0 - значение по умолчанию для числовых типов

Console**.**WriteLine**(**number**.**GetValueOrDefault**(**10**));** // 10

В арифметических операциях**,** если один из операндов равен **null,** то результатом операции также будет **null:**

int**?** x **=** **null;**

int**?** w **=** x **+** 7**;** // w = null

В операциях сравнения **>,** **<,** **>=** и **<=,** если хотя бы один из операндов равен **null,** то возвращается **false** **(**кроме операции **!=).**

## Проверка на null, операторы ?. и ??

С помощью оператора **is** мы можем проверить значение объекта**:**

string name **=** **null;**

**if** **(**name **is** **null)** **...** // объект is значение

// Также можно проверить на соответствие типу:

// name is string

С помощью **is** not можно проверить отсутствие значения**:**

**if** **(**name **is** not **null)** **...**

### Оператор ??

Оператор **??** возвращает левый операнд**,** если этот операнд не равен **null.**

Иначе возвращается правый операнд**.**

При этом левый операнд должен принимать **null.**

int**?** id **=** 200**;**

int personid **=** id **??** 0**;** // равно 200, так как id не равен null

### Оператор условного null ( ?. )

Person**?.**name

## Псевдонимы типов и статический импорт

### Псевдонимы

**using** printer **=** System**.**Console**;**

printer**.**WriteLine**(**"Hello, World"**);**

**using** User **=** Person**;**

User tom **=** **new** User**(**"Tom"**);**

### Статический импорт

Выражение **using** static подключает в программу все статические методы и свойства**,** а также константы**.**

И после этого мы можем не указывать название класса при вызове метода**.**

**using** static System**.**Console**;**

WriteLine**(**"Hello from C# 8.0"**);**

ООП

## Наследование

class Person **{** string Name **{** **get;** **set;** **}** **}**

class Employee **:** Person **{** **...** **}**

Person person **=** **new** Person **{** Name **=** "Tom" **};**

Person employee **=** **new** Employee **{** Name **=** "Sam" **};**

*Ограничения****:***

**-** Класс может наследоваться только от одного класса**.**

**-** Тип доступа к производному классу должен быть таким же**,** как и у базового класса**,** или более строгим**.** Если базовый и производный класс находятся в разных сборках **(**проектах**),** то производый класс может наследовать только от класса **public.**

**-** **sealed** класс не допускает создание наследников**:** **sealed** class Admin **{}**

**-** Нельзя унаследовать класс от статического класса**.**

### Ключевое слово base

Если в базовом классе не определен конструктор по умолчанию без параметров**,** а только конструкторы с параметрами**,** то в производном классе мы обязательно должны вызвать один из этих конструкторов через ключевое слово **base.**

При вызове конструктора класса сначала отрабатываются конструкторы базовых классов и только затем конструкторы производных**.**

class Person

**{**

**public** string Name **{** **get;** **set;}**

**public** Person**(**string name**)**

**{**

Name **=** name**;**

**}**

**}**

class Employee **:** Person

**{**

**public** string Company **{** **get;** **set;** **}**

**public** Employee**(**string name**,** string company**)** **:** **base(**name**)**

**{**

Company **=** company**;**

**}**

**}**

## Преобразование типов

### Восоходящее преобразование. Upcasting

Employee employee **=** **new** Employee**(**"Tom"**,** "Microsoft"**);**

Person person **=** employee**;** // преобразование от Employee к Person

employee и person будут указывать на один и тот же объект в памяти**,** но переменной person будет доступна только та часть**,** которая представляет функционал типа Person**.**

### Нисходящее преобразование. Downcasting

Person person **=** **new** Person**();**

Employee employee2 **=** **(**Employee**)**person**;** // преобразование от Person к Employee

### Способы преобразований

Person person **=** **new** Person**();**

Employee**?** employee **=** person **as** Employee**;** // В случае неудачного преобразования выражение будет содержать значение null

**if** **(**person **is** Employee employee**)**

**{**

Console**.**WriteLine**(**employee**.**Company**);**

**}**

Выражение **if** **(**person **is** Employee employee**)** проверяет**,** является ли переменная person объектом типа Employee**.** Если person является объектом Employee**,** то автоматически преобразует значение переменной person в тип Employee и преобразованное значение сохраняет в переменную employee**.**

**if** **(**person **is** Employee**)** // без преобразования типа Person в Employee

**{**

Console**.**WriteLine**(**"Представляет тип Employee"**);**

**}**

## Виртуальные методы и свойства

Методы в родительском классе**,** помеченные модификатором **virtual,** доступны для переопределения**.**

В классе**-**наследнике мы можем переопределить его**,** пометив модификатором **override.**

class Person

**{**

**virtual** void Method**(){}**

**}**

class Employee **:** Person

**{**

**virtual** void Method**(){}**

**}**

Ограничения**:**

**-** Виртуальный и переопределенный методы должны иметь один и тот же модификатор доступа**.**

**-** Нельзя переопределить или объявить виртуальным статический метод**.**

### Ключевое слово base

class Employee **:** Person

**{**

Employee**(** **...** **)** **:base(** **...** **)** // из базового класса можно вызвать и конструктор

**{**

**...**

**}**

**override** void Method**()**

**{**

**base.**Print**();** // и метод с помощью ключевого слова base

**...**

**}**

**}**

### Переопределение свойств

Можно переопределять свойства**:**

class Person

**{**

**private** int age**;**

**public** **virtual** int Age **{** **get;** **set;** **}**

**}**

class Employee **:** Person

**{**

**public** **override** Age

**{**

**get** **=>** **base.**Age**;**

**set** **{** **if** **(value** **>** 18**)** **base.**Age **=** **value;** **}**

**}**

**}**

### Запрет переопределения методов

**override** **sealed** запрещает переопределение методов и свойств**.** Оно работает только в уже переопределенных методах в классах**-**наследниках**,** т**.**к**.** в родительском классе достаточно не ставить модификатор **virtual:**

class Employee **:** Person

**{**

**override** **sealed** void Method**()** **{** **...** **}**

**}**

## Скрытие методов и свойств

class Person

**{**

**public** int age **=** 1**;**

**public** string Name **{** **get;** **set;** **}**

**public** void Print**()** **{}** // не виртуальный метод - нельзя переопределить

**}**

class Employee **:** Person

**{**

**public** **new** int age **=** 18**;** // скрываем родительскую переменную или константу своей

**public** **new** string Name **{** **get** **=>** **...** **;** **set** **=>** **base.**Name **=** **value;** **}** // скрываем родительское свойство своим

**public** **new** void Print**()** **{}** // скрываем родительский метод своим

**}**

## Различие переопределения и скрытия методов

### Переопределение

class Person

**{**

**virtual** void Print**()** **{** Console**.**WriteLine**(**"Person"**);** **}**

**}**

class Employee **:** Person

**{**

**override** void Print**()** **{** Console**.**WriteLine**(**"Employee"**);** **}**

**}**

Person person **=** **new** Employee**();**

tom**.**Print**();** // Employee

Через Таблицу Виртуальных Методов вызывается метод класса Employee**,** хотя объект типа Person.

### Скрытие

class Person

**{**

void Print**()** **{** Console**.**WriteLine**(**"Person"**);** **}**

**}**

class Employee **:** Person

**{**

**new** void Print**()** **{** Console**.**WriteLine**(**"Employee"**);** **}**

**}**

Person person **=** **new** Employee**();**

tom**.**Print**();** // Person

Вызывается метод класса Person объекта типа Person.

## Абстрактные классы и члены классов

**abstract** class Transport

**{**

int age **=** 0**;** // может иметь переменные

int Age **{** **get** **=>** age**;** **set** **=>** age **=** **value;** **}** // может иметь свойства

void Move**()** **{** **...** реализация **}** // может иметь методы с реализацией

**abstract** string Name **{** **get;** **set;** **}** // абстрактные свойства

**abstract** int Speed **{** **get;** **set;** **}**

**abstract** void Stop**();** // абстрактный метод

**}**

class Car **:** Transport

**{**

string name**;**

**override** string Name **{** **get** **=>** name**;** **set** **=>** name **=** **value;** **}** // реализации свойств

**override** int Speed **{** **get;** **set;** **}**

**override** void Stop**()** **{** **...** реализация **}** // реализация метода

**}**

## Класс System.Object и его методы

У класса Object есть основные методы**,** которые можно переопределить с помощью ключевого слова **override:**

**override** string**?** ToString**()**

**override** int GetHashCode**()**

**override** bool Equals**(object?** obj**)**

**...** и метод который не переопределяется**:**

Type GetType**()** // проверить тип можно с помощью конструкций:

// if (person.GetType() == typeof(Person)

// if (person is Person)

## Обобщения (generics)

class Person**<**T**>** **{...}**

class Person**<**T**,** K**>** **{...}**

### Статические поля обобщенных классов

class Person**<**T**>**

**{**

static T**?** Id**;**

**}**

Person**<**int**>** tom **=** **new** Person**<**int**>();**

tom**.**Id **=** 1**;**

Person**<**string**>** bob **=** **new** Person**<**string**>();**

bob**.**Id **=** "Bob"**;**

Для класса Person**<**int**>** и для класса Person**<**string**>** в куче будет создана своя переменная Id**.**

### Обобщенные методы

void Method**<**T**>(**T x**)** **{...}**

## Ограничения обобщений

### Ограничения методов

void SendMessage**<**T**>(**T message**)** **where** T**:** Message **{...}**

// через универсальный параметр T будут передаваться объекты класса Message и производных классов

### Ограничения обобщений в типах

class Messenger**<**T**>** **where** T **:** Message **{...}**

### Типы ограничений и стандартные ограничения

В качестве ограничений мы можем использовать следующие типы**:**

**-** Классы **(where** T **:** Message**)**

**-** Интерфейсы **(where** T **:** IMessage**)**

**-** class **-** универсальный параметр должен представлять класс **(where** T **:** class**)**

**-** struct **-** универсальный параметр должен представлять структуру **(where** T **:** struct**)**

**-** **new()** **-** универсальный параметр должен представлять тип**,** который имеет общедоступный **(public)** конструктор без параметров **(where** T **:** **new())**

Порядок следования нескольких ограничений**:**

1. Название класса**,** class**,** struct**.** Причем мы можем одновременно определить только одно из этих ограничений

2. Название интерфейса

3. **new()**

class Messenger**<**T**>** **where** T**:** Message**,** **new()**

### Использование нескольких универсальных параметров

class Messenger**<**T**,** P**>**

**where** T **:** Message

**where** P**:** Person

**{...}**

## Наследование обобщенных типов

Пример 1**:**

class Person**<**T**>** **{...}**

class Employee**<**T**>** **:** Person**<**T**>** **{...}**

Пример 2**:**

class Person**<**T**>** **{...}**

class Employee **:** Person**<**string**>** **{...}**

Пример 3**:**

class Person**<**T**>** **{...}**

class Employee**<**T**>** **:** Person**<**int**>** **{...}**

Пример 4**:**

class Person**<**T**>** **{...}**

class Employee**<**T**,** K**>** **:** Person**<**T**>** **where** K **:** struct **{...}**

Пример 5**:**

class Person**<**T**>** **where** T **:** class **{...}**

class Employee**<**T**>** **:** Person**<**T**>** **where** T **:** class **{...}**

Если в базовом классе в качестве ограничение указано class**,** то есть любой класс**,** то в производном классе также надо указать в качестве ограничения class**,** либо же какой**-**то конкретный класс**.**

Исключения

## Конструкция try..catch..finally

**try** **{}**

**catch** **{}** // просто catch или catch(Exception), или catch(Exception e)

**finally** **{}** // опционально

### Фильтры исключений

int x **=** 1**,** y **=** 0**;**

**try**

**{**

int result **=** x **/** y**;**

**}**

**catch** **when** **(**y **==** 0**)** **{...}** // catch(Exception) when (), catch(Exception e) when()

## Типы исключений. Класс Exception

***Свойства класса Exception:***

**-** InnerException**:** хранит информацию об исключении**,** которое послужило причиной текущего исключения

**-** Message**:** хранит сообщение об исключении**,** текст ошибки

**-** Source**:** хранит имя объекта или сборки**,** которое вызвало исключение

**-** StackTrace**:** возвращает строковое представление стека вызывов**,** которые привели к возникновению исключения

**-** TargetSite**:** возвращает метод**,** в котором и было вызвано исключение

***Некоторые типы исключений:***

**-** DivideByZeroException**:** представляет исключение**,** которое генерируется при делении на ноль

**-** ArgumentOutOfRangeException**:** генерируется**,** если значение аргумента находится вне диапазона допустимых значений

**-** ArgumentException**:** генерируется**,** если в метод для параметра передается некорректное значение

**-** IndexOutOfRangeException**:** генерируется**,** если индекс элемента массива или коллекции находится вне диапазона допустимых значений

**-** InvalidCastException**:** генерируется при попытке произвести недопустимые преобразования типов

**-** NullReferenceException**:** генерируется при попытке обращения к объекту**,** который равен **null** **(**то есть по сути неопределен**)**

## Генерация исключения и оператор throw

**throw** **new** Exception**(**"..."**);**

**catch**

**{**

**...**

**throw;**

**}** // в блоке catch исключение обрабатывается, а затем передается внешнему блоку кода

Делегаты, лямбды и события

## Делегаты

Operation operation **=** Add**;** // делегат указывает на метод Add

int result **=** operation**(**4**,** 5**);** // фактически Add(4, 5)

// можно создать объект делегата с помощью конструктора

operation **=** **new** Operation**(**Multiply**);**

result **=** operation**(**4**,** 5**);** // фактически Multiply(4, 5)

int Add**(**int x**,** int y**)** **=>** x **+** y**;**

int Multiply**(**int x**,** int y**)** **=>** x **\*** y**;**

**delegate** int Operation**(**int x**,** int y**);**

**Делегат можно определять внутри или вне класса.**

**Методы должны соответствовать делегату по возвращаемому типу и наборы параметров (учитывая ref, out, in)**

### Добавление методов в делегат

**В делегат можно добавлять методы (с помощью +=), которые попадают в список - invokation list, и будут вызываться последовательно, но, ЕСЛИ ДЕЛЕГАТ ВОЗВРАЩАЕТ ЗНАЧЕНИЕ, то ВОЗВРАЩАЕТСЯ ЗНАЧЕНИЕ ТОЛЬКО ПОСЛЕДНЕГО МЕТОДА ИЗ СПИСКА. Можно несколько раз добавлять один и тот же метод. Можно удалять метод (если список содержит несколько одинаковых методов, то с конца списка) с помощью -=.**

Message message **=** Hello**;**

message **+=** HowAreYou**;** // теперь message указывает на два метода

message **+=** Hello**;**

message **+=** HowAreYou**;**

message **-=** Hello**;** // удалили последний Hello()

message**();** // методы вызываются по очереди

void Hello**()** **=>** Console**.**WriteLine**(**"Hello/n"**);**

void HowAreYou**()** **=>** Console**.**WriteLine**(**"How are you?/n"**);**

**delegate** void Message**();**

/\*

Hello

How are you?

How are you?

\*/

### Объединение делегатов

Message mes1 **=** Hello**;**

Message mes2 **=** HowAreYou**;**

Message mes3 **=** mes1 **+** mes2**;** // объединяем делегаты

mes3**();** // вызываются все методы из mes1 и mes2

void Hello**()** **=>** Console**.**WriteLine**(**"Hello/n"**);**

void HowAreYou**()** **=>** Console**.**WriteLine**(**"How are you?/n"**);**

**delegate** void Message**();**

/\*

Hello

How are you?

\*/

### Вызов делегата с помощью метода Invoke()

Operation**?** op **=** Add**;**

// другой способ вызова делегата - с помощью Invoke()

int result **=** op**?.**Invoke**(**4**,** 5**)** **?:** 0**;** // проверяем на null (вдруг список делегата пуст)

int Add**(**int x**,** int y**)** **=>** x **+** y**;**

**delegate** int Operation**(**int x**,** int y**);**

### Обобщенные делегаты (Generic)

Operation**<**double**,** int**>** op **=** Square**;**

int result **=** op**(**4**);** // 16.0

double Square**(**int n**)** **=>** **(**double**)** n **\*** n**;**

**delegate** T Operation**<**T**,** K**>(**K val**);**

### Делегаты как параметры методов

DoOperation**(**5**,** 4**,** Add**);** // 9

DoOperation**(**5**,** 4**,** Multiply**);** // 20

void DoOperation**(**int a**,** int b**,** Operation op**)**

**{**

Console**.**WriteLine**(**op**(**a**,**b**));**

**}**

int Add**(**int x**,** int y**)** **=>** x **+** y**;**

int Multiply**(**int x**,** int y**)** **=>** x **\*** y**;**

**delegate** int Operation**(**int x**,** int y**);**

**При вызове метода DoOperation мы можем передать в него в качестве третьего параметра метод, который соответствует делегату Operation.**

### Возвращение делегатов из метода

Letter op **=** SelectOp**(true);**

string result **=** op**();** // A

Letter SelectOp**(**bool a**)** **=>** a **==** "A" **?** A**()** **:** B**();**

string A**()** **=>** "A"

string B**()** **=>** "B"

**delegate** string Letter**();**

## Анонимные методы

***Анонимные методы используются для создания экземпляров делегатов.***

Operation operation **=** **delegate** **(**int x**,** int y**)**

**{**

**return** x **+** y**;**

**};**

int result **=** operation**(**4**,** 5**);** // 9

**delegate** int Operation**(**int x**,** int y**);**

***Если параметров нет, то скобки опускаются***

Message message **=** **delegate**

**{**

Console**.**WriteLine**(**"Message"**);**

**};**

message**();** // Message

**delegate** void Operation**();**

***Даже если делегат содержит параметры, то в анонимном методе они могут не использоваться, и скобки опускаются***

Operation operation **=** **delegate**

**{**

**return** 0**;**

**};**

int result **=** operation**(**4**,** 5**);** // 9

**delegate** int Operation**(**int x**,** int y**);**

***Можно передавать анонимный метод в качестве параметра метода типа делегата***

ShowMessage**(**"Hello!"**,** **delegate** **(**string mes**)**

**{**

Console**.**WriteLine**(**mes**);**

**});**

static void ShowMessage**(**string message**,** MessageHandler handler**)**

**{**

handler**(**message**);**

**}**

**delegate** void MessageHandler**(**string message**);**

## Лямбды

***(список\_параметров) => выражение***

**Лямбда-выражения представляют упрощенную запись анонимных методов. С точки зрения типа данных лямбда-выражение представляет делегат.**

Message hello **=** **()** **=>** Console**.**WriteLine**(**"Hello"**);**

hello**();** // Hello

**delegate** void Message**();**

**Если лямбда-выражение содержит несколько действий, то они помещаются в фигурные скобки:**

Message hello **=** **()** **=>**

**{**

Console**.**Write**(**"Hello "**);**

Console**.**WriteLine**(**"World"**);**

**};**

hello**();** // Hello World

**delegate** void Message**();**

**Начиная с версии C# 10:**

var hello **=** **()** **=>** Console**.**WriteLine**(**"Hello"**);**

// переменной hello присваивается тип встроенного делегата Action,

// который не принимает параметров и ничего не возвращает

hello**();** // Hello

### Параметры лямбды

**При определении списка параметров мы можем не указывать для них тип данных:**

Operation sum **=** **(**x**,** y**)** **=>** Console**.**WriteLine**(**$"{x + y}"**);**

sum**(**1**,** 2**);** // 3

**delegate** void Operation**(**int x**,** int y**);**

**Но при использовании неявной типизации (var) их нужно указать:**

var sum **=** **(**int x**,** int y**)** **=>** Console**.**WriteLine**(**$"{x + y}"**);**

**Если лямбда имеет один параметр, для которого не требуется указывать тип данных, то скобки можно опустить:**

Print print **=** message **=>** Console**.**WriteLine**(**message**);**

print**(**"Hello"**);** // Hello

**delegate** void Print**(**string message**);**

**Начиная с C# 12 параметры лямбда-выражений могут иметь значения по умолчанию:**

var welcome **=** **(**string message **=** "Hello"**)** **=>** Console**.**WriteLine**(**message**);**

welcome**();** // Hello

welcome**(**"Bye"**);** // Bye

### Возвращение результата

var sum **=** **(**int x**,** int y**)** **=>** x **+** y**;**

int result **=** sum**(**3**,** 2**);** // 5

// Если лямбда-выражение содержит несколько выражений (или одно выражение в фигурных

// скобках), тогда нужно использовать оператор return, как в обычных методах:

Operation multiply **=** **(**x**,** y**)** **=>**

**{**

**...**

**return** x **\*** y**;**

**}**

result **=** **(**3**,** 2**);** // 6

**delegate** int Operation**(**int x**,** int y**);**

### Добавление и удаление действий в лямбда-выражении, метод Invoke

var hello **=** **()** **=>** Console**.**WriteLine**(**"Hello "**);**

var world **=** **()** **=>** Console**.**WriteLine**(**"World"**);**

hello **+=** world**;**

hello **+=** **()** **=>** Console**.**WriteLine**(**"!"**);**

hello**?.**Invoke**();** // Hello World!

hello **-=** world**;**

hello**();** // Hello !

### Лямбда-выражение как аргумент метода

ShowMessage**(**"Hello!"**,** mes **=>** Console**.**WriteLine**(**mes**));**

static void ShowMessage**(**string message**,** MessageHandler handler**)**

**{**

handler**(**message**);**

**}**

**delegate** void MessageHandler**(**string message**);**

### Лямбда-выражение как результ метода

Letter op **=** SelectOp**(true);**

string result **=** op**();** // A

Letter SelectOp**(**bool a**)** **=>**

**{**

**return** a **==** "A" **?** **()** **=>** "A" **:** **()** **=>** "B"**;**

**}**

**delegate** string Letter**();**

## События

### Определение и вызов событий

**В одном классе определяем событие, и в нужных местах (в нужных методах) вызываем это событие (с разными параметрами, если нужно)**

class MyClass

**{**

**delegate** void MyDelegate**(**string message**);** // 1. Определение делегата

**event** MyDelegate MyEvent**;** // 2.Определение события

**...**

MyEvent**(**"Произошло хорошее событие"**);** // 3.Вызов события

**...**

MyEvent**?.**Invoke**(**"Произошло плохое событие"**);** // Вызов события в другом месте

**}**

### Добавление обработчика события

**В другом классе создаем обработчик события - метод с той же сигнатурой, что и делегат события, который реализует поведение-реакцию на событие (в зависимости от пришедших параметров, если они есть). Добавляем обработчик события с помощью +=**

MyClass**.**MyEvent **+=** EventHandler**;** // 5. Добавление обработчика события

MyClass**.**MyEvent **+=** EventHandlerRed**;** // Добавление другого обработчика события

MyClass**.**MyEvent **-=** EventHandler**;** // Удаление обработчика события

void EventHandler**(**string message**)** **=>** Console**.**WriteLine**(**message**);** // 4. Реализация обработчика события

void EventHandlerRed**(**string message**)** // Реализация другого обработчика события

**{**

// Устанавливаем красный цвет символов

Console**.**ForegroundColor **=** ConsoleColor**.**Red**;**

Console**.**WriteLine**(**message**);**

// Сбрасываем настройки цвета

Console**.**ResetColor**();**

**}**

***Установка в качестве обработчика делегата:***

MyClass**.**MyEvent **+=** **new** MyClass**.**MyDelegate**(**EventHandler**);** // установка делегата через конструктор

***Установка в качестве обработчика анонимного метода:***

MyClass**.**MyEvent **+=** **delegate** **(**string message**)** **=>** Console**.**WriteLine**(**message**);**

***Установка в качестве обработчика лямбда-выражения:***

MyClass**.**MyEvent **+=** message **=>** Console**.**WriteLine**(**message**);**

### Управление обработчиками (add, remove)

class MyClass

**{**

**delegate** void MyDelegate**(**string message**);** // делегат

MyDelegate**?** myEvent**;** // переменная делегата

**event** MyDelegate MyEvent // переменная события

**{**

**add** // будет вызываться при добавлении обработчика

**{**

myEvent **+=** **value;** // value - ключевое слово - добавляемый обработчик

**}**

**remove** // будет вызываться при удалении обработчика

**{**

myEvent **-=** **value;**

**}**

**}**

**...**

**}**

MyClass**.**MyEvent **+=** EventHandler**;**

void EventHandler**(**string message**)** **=>** Console**.**WriteLine**(**message**);**

## Ковариантность и контравариантность делегатов

class Message **{**

string text**;**

Message**(**string text**)** **=>** **this.**text **=** text**;**

**}**

class Email **:** Message **{** Email**(**string text**)** **:** **base(**text**)** **{}** **}**

**Дочерний класс - это частный случай родительского класса, и он может иметь больший функционал, чем родительский класс.**

**Ковариантность - это возвращение дочерних классов вместо родительских.**

**(могу вернуть Email вместо Message, т.к. Email - это и есть Message, но не наоборот, и то, что есть в Message, есть и в Email).**

**Контрвариантность - это использование родительских классов вместо дочерних.**

**(в качестве реализации делегата могу использовать метод, принимающий Message вместо Email, т.к. то, что есть в Message, есть и в Email).**

### Ковариантность

**Ковариантность позволяет передать делегату метод, возвращаемый тип которого является дочерним от возвращаемого типа делегата.**

**delegate** Message MyDelegate**(**string text**);** // делегат возвращает родителя

Email MyMethod**(**string text**)** **=>** **new** Email**(**text**);** // метод возвращает экземпляр дочернего //класса

MyDelegate myDelegate **=** MyMethod**;** // ковариантность

### Контрвариантность

**delegate** void MyDelegate**(**Email message**);** // делегат принимает экземпляр дочернего класса

void MyMethod**(**Message message**);** // метод принимает родителя

MyDelegate myDelegate **=** MyMethod**;** // контравариантность

### Ковариантность и контравариантность в обобщенных делегатах

out - можно этот тип или родительский

in - можно этот тип или дочерний

#### Ковариантность

**delegate** T MyDelegate**<out** T**>(**string text**);** // делегат

// реализация делегата возвращает экземпляр дочернего типа

MyDelegate**<**Email**>** myMethod1 **=** text **=>** **new** Email**(**text**);**

// реализация делегата возвращает экземпляр родительского типа

MyDelegate**<**Message**>** myMethod2 **=** myDelegate1**;** // ковариантность

Message message **=** myMethod2**(**"Message"**);** // вызов делегата. Возвращение родительского //типа

#### Контравариантность

**delegate** void MyDelegate**<in** T**>(**T message**);**

// реализация делегата принимает кземпляр родительского типа

MyDelegate**<**Message**>** myMethod1 **=** message **=>** Console**.**WriteLine**(**message**.**text**);**

// реализация делегата принимает кземпляр дочернего типа

MyDelegate**<**Email**>** myMethod2 **=** myDelegate1**;** // контравариантность

myMethod2**(new** Email**(**"Email"**));** // вызов делегата. Использование дочернего типа

## Делегаты Action, Predicate и Func

### Action

**public** **delegate** void Action**()**

**public** **delegate** void Action**<in** T**>(**T obj**)**

**Принимает от 0 до 16 объектов, ничего не возвращает.**

void Add**(**int x**,** int y**)** **=>** Console**.**WriteLine**(**$**{**x **+** y**});** // метод соответсвует делегату Action

void DoOperation**(**int a**,** int b**,** Action**<**int**,** int**>** operation**)** // метод принимает делегат Action

**=>** operation**(**a**,** b**);**

DoOperation**(**5**,** 6**,** Add**);** // 11

### Predicate

**delegate** bool Predicate**<in** T**>(**T obj**);**

**Принимает один параметр и возвращает значение типа bool.**

Predicate**<**int**>** isPositive **=** **(**int x**)** **=>** x **>** 0**;** // проверяет на положительность

### Func

TResult Func**<out** TResult**>()**

TResult Func**<in** T**,** **out** TResult**>(**T obj**)**

**Принимает от 0 до 16 объектов, возвращает результат.**

int Add**(**int x**,** int y**)** **=>** x **+** y**;** // метод соответсвует делегату Func

int DoOperation**(**int a**,** int b**,** Func**<**int**,** int**>** operation**)** // метод принимает делегат Func

**=>** operation**(**a**,** b**);**

int result **=** DoOperation**(**5**,** 6**,** Add**);** // 11

## Замыкания

**Замыкание (closure) представляет объект функции, который запоминает свое лексическое окружение и может его менять даже при вызове вне своей области видимости.**

**Технически замыкание включает три компонента:**

**- внешняя функция, которая определяет некоторую область видимости**

**- переменные и параметры (лексическое окружение), которые определены во внешней функции**

**- вложенная функция, которая использует переменные и параметры внешней функции**

Action Outer**()** // метод или внешняя функция

**{**

int x **=** 5**;** // лексическое окружение - локальная переменная

void Inner**()** // локальная функция

**{**

x**++;** // операции с лексическим окружением

Console**.**WriteLine**(**x**);**

**}**

**return** Inner**;** // возвращаем локальную функцию

**}**

Action fn **=** Outer**();** // fn = Inner, так как метод Outer возвращает функцию Inner

// вызываем внутреннюю функцию Inner

fn**();** // 6

fn**();** // 7

fn**();** // 8

### Реализация с помощью лямбда-выражений

Action outerFun **=** **()** **=>**

**{**

int x **=** 5**;**

var innerFun **=** **()** **=>** Console**.**WriteLine**(++**x**);**

**return** innerFun**;**

**};**

Action fn **=** outerFun**();** // fn = innerFun, так как outerFun возвращает innerFn

// вызываем innerFun

fn**();** // 6

fn**();** // 7

fn**();** // 8

### Применение параметров

Func**<**int**,** int**>** Multiply**(**int n**)**

**{**

int Inner**(**int m**)** **=>** n **\*** m**;**

**return** Inner**;**

**}**

Func**<**int**,** int**>** func **=** Multiply**(**5**);**

// var multiply = (int n) => (int m) => n \* m;

// var func = multiply(5);

Console**.**WriteLine**(**func**(**5**));** // 25

Console**.**WriteLine**(**func**(**6**));** // 30

Console**.**WriteLine**(**func**(**7**));** // 35

Интерфейсы

## Определение интерфейсов

**Интерфейсы могут определять:**

**- Методы**

**- Свойства**

**- Индексаторы**

**- События**

**- Статические поля и константы (с C# 8.0)**

**Не могут определять нестатические переменные. Члены интерфейсов public по умолчанию. Сами интерфейсы, как и классы, по умолчанию internal. Но и к членам и к самим интерфейсам можно применять другие модификаторы доступа.**

**interface** IMyInterface

**{**

// константа

const int myConst **=** 0**;**

// статическая переменная

static int myStatic **=** 0**;**

// свойство

string MyProperty **{** **get;** **set;** **}** // это не автосвойство, а свойство без реализации

// реализация свойства по умолчанию

int MyIntProperty **{** **get** **{** **return** 0**;** **}** **}**

// метод

void MyMethod**();**

// метод с реализацией по умолчанию

void MyMethod2**()** **{** **...** **}** // с C# 8.0

// статический метод должен иметь реализацию по умолчанию

static void MyStaticMethod**()** **{** **...** **}**

**delegate** void MyDelegate**(**string text**);** // определение делегата для события

// событие

**event** MyDelegate MyEvent**;**

**}**

## Применение интерфейсов

**interface** IMyInterface **{** **...** **}**

class MyClass **:** IMyInterface **{** **...** **}**

struct MyStruct **:** IMyInterface **{** **...** **}**

**Если методы и свойства интерфейса не имеют модификатора доступа, то по умолчанию они public. При их реализации можно применять только public.**

**interface** IMovable

**{**

void Move**()** **=>** **...** // реализация по умолчанию

**}**

class Person **:** IMovable**;**

**...**

Person p **=** **new** Person**();**

// p.Move(); - ошибка, т.к. класс Person не имеет реализации метода Move(), но:

IMovable p **=** **new** Person**();**

p**.**Move**();** // используется реализация по умолчанию интерфейса

Множественная реализация интерфейсов

class MyClass **:** IMyInterface1**,** IMyInterface2 **...** **{** **...** **}**

## Явная реализация интерфейсов

**interface** ISchool

**{**

void Study**();**

**}**

**interface** IUniversity

**{**

void Study**();**

**}**

class Person **:** ISchool**,** IUniversity

**{**

// public void Study() => ... // реализация метода Stydy() сразу обоих интерфейсов

//или :

**public** ISchool**.**Study**()** **=>** **...** // метод существует только для интерфейса ISchool

**public** IUniversity**.**Study**()** **=>** **...** // метод существует только для интерфейса //IUniversity

**}**

Person person **=** **new** Person**();**

// person.Study(); ошибка, т.к. в Person нет реализации этого метода, она только для //интерфейсов

**((**ISchool**)** person**).**Study**();**

// или :

// безопасное приведение типа

**if** **(**person **is** IUniversity univirsity1**)** univirsity1**.**Study**();**

// или :

IUniversity univirsity2 **=** **new** Person**();**

univirsity2**.**Study**();**

Модификаторы доступа

**Если члены интерфейса не public, то можно использовать явную реализацию интерфейса без модификаторов:**

**interface** IMyInterface

**{**

**protected** **internal** string Name **{** **get;** **}**

**}**

class MyClass **:** IMyInterface

**{**

string name**;**

// явная реализация свойства в виде автосвойства

string IMyInterface**.**Name **{** **get** **=>** name**;** **}** // обратиться к такому свойству можно //только через объект типа интерфейса

**}**

**Либо использовать неявную реализацию с модификатором public:**

**interface** IMyInterface

**{**

**protected** **internal** string Name **{** **get;** **}**

**}**

class MyClass **:** IMyInterface

**{**

string name**;**

// неявная реализация свойства в виде автосвойства

**public** string Name **{** **get** **=>** name**;** **}** // обратиться к такому свойству можно через //объект типа класса

**}**

## Реализация интерфейсов в базовых и производных классах

**В абстрактном классе, расширяющем интерфейс, можно не реализовывать метод интерфейса, а сделать его абстрактным.**

**interface** MyInterface **{** string MyMethod**();** **}**

**abstract** class MyAbstractClass **{** **abstract** string MyMethod**();** **}**

class MyClass **:** MyAbstractClass **{** **public** **override** string MyMethod**()** **=>** "MyClass"**;** **}**

**Название класса-родителя идет перед названием интерфейсов:**

class MyClass **:** MyBaseClass**,** MyInterface1**,** MyInterface2**...**

**1. Если в классе-родителе и в расширяемом интерфейсе одинаковые методы, то его реализация в родителе переопределяет метод из интерфейса. В дочернем классе тогда его можно не переопределять.**

**interface** MyInterface **{** string MyMethod**();** **}**

class MyBaseClass **{** **public** string MyMethod**();** **}**

class MyClass **:** MyBaseClass**,** MyInterface **{** **}**

**2. Либо переопределить (override), если в родительском классе этот метод virtual (или abstract).**

**interface** MyInterface **{** string MyMethod**();** **}**

class MyBaseClass **:** MyInterface

**{**

**public** **virtual** string MyMethod**()** **=>** "MyBaseClass"**;**

**}**

class MyClass **:** MyBaseClass

**{**

**public** **override** string MyMethod**()** **=>** "MyClass"**;**

**}**

MyBaseClass myBaseClass **=** **new** MyClass**();**

string result **=** myBaseClass**.**MyMethod**();** // MyClass - т.к. есть переопределенный метод

MyInterface myInterface **=** **new** MyClass**();**

result **=** myInterface**.**MyMethod**();** // MyClass

**3. Либо в дочернем классе скрыть метод**

**interface** MyInterface **{** string MyMethod**();** **}**

class MyBaseClass **:** MyInterface

**{**

**public** string MyMethod**()** **=>** "MyBaseClass"**;**

**}**

class MyClass **:** MyBaseClass // класс наследуется от класса-родителя

**{**

**public** **new** string MyMethod**()** **=>** "MyClass"**;**

**}**

MyBaseClass myBaseClass **=** **new** MyClass**();**

string result **=** myBaseClass**.**MyMethod**();** // MyBaseClass - т.к. нет переопределенного метода

MyInterface myInterface **=** **new** MyClass**();**

result **=** myInterface**.**MyMethod**();** // MyBaseClass - т.к. интерфейс реализован только в базовом классе

MyClass myClass **=** **new** MyClass**();**

result **=** myClass**.**MyMethod**();** // MyClass

**4. Либо повторно реализовать интерфейс в классе-наследнике.**

**interface** MyInterface **{** string MyMethod**();** **}**

class MyBaseClass **:** MyInterface

**{**

**public** string MyMethod**()** **=>** "MyBaseClass"**;**

**}**

class MyClass **:** MyBaseClass**,** MyInterface // класс также напрямую наследуется от интерфейса

**{**

**public** **new** string MyMethod**()** **=>** "MyClass"**;**

**}**

MyBaseClass myBaseClass **=** **new** MyClass**();**

string result **=** myBaseClass**.**MyMethod**();** // MyBaseClass - т.к. для BaseClass нет переопределенного метода

MyInterface myInterface **=** **new** MyClass**();**

result **=** myInterface**.**MyMethod**();** // MyClass - т.к. получается, что интерфейс реализован в классе-наследнике

MyClass myClass **=** **new** MyClass**();**

result **=** myClass**.**MyMethod**();** // MyClass

**5. Либо явно реализовать интерфейс**

**interface** MyInterface **{** string MyMethod**();** **}**

class MyBaseClass **:** MyInterface

**{**

**public** string MyMethod**()** **=>** "MyBaseClass"**;**

**}**

class MyClass **:** MyBaseClass**,** MyInterface // класс также напрямую наследуется от интерфейса

**{**

**public** **new** string MyMethod**()** **=>** "MyClass"**;**

// явная реализация интерфейса

string MyInterface**.**MyMethod**()** **=>** "MyInterface"**;**

**}**

MyBaseClass myBaseClass **=** **new** MyClass**();**

string result **=** myBaseClass**.**MyMethod**();** // MyBaseClass - т.к. для BaseClass нет переопределенного метода

MyInterface myInterface **=** **new** MyClass**();**

result **=** myInterface**.**MyMethod**();** // MyInterface - т.к. интерфейс реализован в классе-наследнике

MyClass myClass **=** **new** MyClass**();**

result **=** myClass**.**MyMethod**();** // MyClass

## Наследование интерфейсов

**В отличие от классов к интерфейсам мы не можем применять модификатор sealed, чтобы запретить наследование интерфейсов. Но интерфейсы могут ипользовать new для сокрытия методов.**

**interface** MyInterface1 **{** string MyMethod**()** **=>** "MyInterface1"**;** **}**

**interface** MyInterface2 **{** **new** string MyMethod**)()** **=>** "MyInterface2"**;** **}**

class MyClass **:** MyInterface2 **{}**

MyInterface1 myInterface1 **=** **new** MyClass**();**

string result **=** myInterface1**.**MyMethod**();** // MyInterface1

MyInterface2 myInterface2 **=** **new** MyClass**();**

result **=** myInterface2**.**MyMethod**();** // MyInterface2

**При наследовании интерфейсов, как и классов, дочерний интерфейс должен иметь тот же уровень доступа или более строгий. Т.к., если он будет более открытый, то он откроет и методы и поля своего родительского класса / интерфейса.**

**public** **interface** MyInterface1 **{}**

**internal** **interface** MyInterface2 **:** MyInterface1 **{}**

## Интерфейсы как ограничения обобщений

**interface** MyInterface1 **{}**

**interface** MyInterface2 **{}**

class MyClass1 **:** MyInterface1**,** MyInterface2 **{}**

// ниже параметр типа Т должен реализовывать сразу оба интерфейса, как MyClass1

class MyClass2**<**T**>** **where** T**:** MyInterface1**,** MyInterface2

**{**

void MyMethod**(**T myInterface**)** **{** **...** **}**

**}**

MyClass1 myClass1 **=** **new** MyClass1**();**

MyClass2**<**MyClass1**>** myClass2 **=** **new** MyClass2**<**MyClass1**>();**

myClass2**.**MyMethod**(**myClass1**);**

### Обобщенные интерфейсы

**interface** MyInterface**<**T**>** **{** void MyMethod**(**T attribute**);** **}**

class MyClass1**<**T**>** **:** MyInterface**<**T**>** **{** **public** void MyMethod**(**T attribute**)** **{** **...** **}** **}**

// можно явно указать, какой будет тип параметра Т

class MyClass2 **:** MyInterface**<**int**>** **{** **public** void MyMethod**(**int attribute**)** **{** **...** **}** **}**

## Копирование объектов. Интерфейс ICloneable

**Реализация интерфейса ICloneable позволяет клонировать объекты.**

### Поверхностное копирование

class MyClass **:** ICloneable

**{**

**public** int Id **{** **get;** **set** **}**

**public** MyClass**(**int id**)** **=>** Id **=** id**;**

**public** **object** Clone**()** // реализация метода интерфейса ICloneable

**{**

**return** **new** MyClass**(**Id**);**

// MemberwiseClone(); - либо так - это поверхностное копирование

// если в классе есть ссылочные поля, то скопируются ссылки этих полей

**}**

**}**

MyClass myClass **=** **new** MyClass**(**1**);** // id = 1

MyClass myClass2 **=** myClass**.**Clone**();** // создание клона с id = 1

### Глубокое копирование

class Person **:** ICloneable

**{**

**public** Company MyCompany **{** **get;** **set;** **}**

**public** Person**(**Company company**)** **=>** MyCompany **=** company**;**

**public** **object** Clone**()** **=>** **new** Person**(new** Company**());**

**}**

class Company **{}**

## Сортировка объектов. Интерфейс IComparable

**Для сортировки наборов сложных объектов применяется интерфейс IComparable. Он имеет всего один метод:**

**public** **interface** IComparable

**{**

int CompareTo**(object?** o**);**

**}**

**Результат метода:**

**- Меньше нуля. Значит, текущий объект должен находиться перед объектом, который передается в качестве параметра;**

**- Равен нулю. Значит, оба объекта равны;**

**- Больше нуля. Значит, текущий объект должен находиться после объекта, передаваемого в качестве параметра;**

class Person **:** IComparable

**{**

**public** int Age **{** **get;** **set** **}**

**public** int CompareTo**(object?** o**)**

**{**

**if** **(**o **is** Person person**)**

**return** Age**.**CompareTo**(**person**.**Age**);**

// или return Age - person.Age;

**else** **throw** **new** ArgumentException**();**

**}**

**}**

**Интерфейс IComparable имеет обобщенную версию:**

class Person **:** IComparable**<**Person**>**

**{**

**...**

**public** int CompareTo**(**Person**?** person**)** **{** **...** **}**

**}**

### Применение компаратора

**public** **interface** IComparer**<in** T**>**

**{**

int Compare**(**T**?** x**,** T**?** y**);**

**}**

**Компаратор в качестве второго параметра принимает метод Array.Sort(Array array, IComparer comparer).**

class MyComparer **:** IComparer**<**Person**>**

**{**

**public** int Compare**(**Person**?** p1**,** Person**?** p2**)** **{** **...** **}**

**}**

Person**[]** people **=** **new** Person**[...];**

Array**.**Sort**(**people**,** **new** MyComparer**);**

## Ковариантность и контравариантность обобщенных интерфейсов

**- Ковариантность: позволяет использовать более конкретный (дочерний) тип, чем заданный изначально;**

**- Контравариантность: позволяет использовать более универсальный (родительский) тип, чем заданный изначально;**

**- Инвариантность: позволяет использовать только заданный тип.**

### Ковариантные интерфейсы

**out - можно использовать вместо более конкретного (дочернего) типа более общий (родительский)**

// объект, реализующий этот интерфейс с типом Т, можно присвоить объекту интерфейса с //родительским по отношению к Т типом

**interface** MyInterface**<out** T**>**

**{**

T MyMethod**();** // метод может возвращать тип Т и его родителей

**}**

### Контравариантные интерфейсы

**in - можно использовать вместо более общего (родительского) типа более конкретный (дочерний)**

// объект, реализующий этот интерфейс с типом Т, можно присвоить объекту интерфейса с //дочерним по отношению к Т типом

**interface** MyInterface**<in** T**>**

**{**

void MyMethod**(**T attribute**);** // в параметры можно положить данный тип или дочерний

**}**

Дополнительные возможности ООП

## Определение операторов

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

**Этот метод должен иметь модификаторы public static. Хотя бы один параметр должен быть типа, в котором он определяется.**

class Counter

**{**

**public** int Value **{** **get;** **set;** **}**

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

**=>** **new** Counter **{** Value **=** c1**.**Value **+** c2**.**Value **};**

**public** static Counter **operator** **+** **(**Counter c**,** int val**)** // перегрузка метода

**=>** **new** Counter **{** Value **=** c**.**Value **+** val **};**

**public** static bool **operator** **>** **(**Counter c1**,** Counter c2**)** **=>** c1**.**Value **>** c2**.**Value**;**

**...**

**}**

**...**

Counter counter3 **=** counter1 **+** counter2**;**

bool result **=** counter1 **>** counter2**;**

**Можно определить логику для следующих операторов:**

**- унарные операторы +x, -x, !x, ~x, ++, --, true, false**

**- бинарные операторы +, -, \*, /, %**

**- операции сравнения ==, !=, <, >, <=, >=**

**- поразрядные операторы &, |, ^, <<, >>**

**- логические операторы &&, ||**

**Кроме того, есть несколько операторов, которые надо определять парами:**

**== и !=**

**< и >**

**<= и >=**

**Нельзя определить логику операторов равенства = , тернарный оператор ?: и некоторые др.**

### Определение инкремента и декремента

**В коде оператора не должны изменяться те объекты, которые передаются в оператор через параметры.**

// префиксный и постфиксный инкремент будет работать правильно

**public** static Counter **operator** **++** **(**Counter c**)**

**{**

// c.Value += 10; - неправильная логика

// return c;

**return** **new** Counter **{** Value **=** c**.**Value **+** 10 **};**

**}**

Counter c1 **=** **new** Counter **{** Value **=** 10 **};**

**++**c1**;** // 20

Counter c2 **=** c1**++;** // c2 = 20, c1 = 30

### Определение операций true и false

**public** static bool **operator** **true(**Counter c**)** **=>** c**.**Value **!=** 0**;**

**public** static bool **operator** **false(**Counter c**)** **=>** c**.**Value **==** 0**;**

**public** static bool **operator** **!** **(**Counter c**)** **=>** c**.**Value **==** 0**;**

**...**

**if** **(**counter**)** **...** // true

**else** **...** // false

// if (!counter) ... // false

## Перегрузка операций преобразования типов

**explicit - если преобразование явное, то есть нужна операция приведения типов**

**implicit - если преобразование неявное**

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

**{**

// логика преобразования

**}**

**Например,**

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**;**

**}**

**}**

Counter counter1 **=** **new** Counter **{** Seconds **=** 23 **};**

int x **=** **(**int**)**counter1**;** // 23

Counter counter2 **=** x**;** // counter2.Seconds == 23

**Какие операции преобразования делать явными, а какие неявные, в данном случае не столь важно, это решает разработчик по своему усмотрению. Оператор преобразования, определенный в типе Counter, должен либо принимать в качестве параметра объект типа Counter, либо возвращать объект типа Counter.**

## Индексаторы

**Индексатор должен иметь как минимум один параметр.**

**Индексатор не может быть статическим и применяется только к экземпляру класса. Но при этом индексаторы могут быть виртуальными и абстрактными и могут переопределяться в произодных классах. Индексатор можно перегружать подобно методам.**

возвращаемый\_тип **this** **[**Тип параметр1**,** **...]**

**{**

**get** **{** **...** **}**

**set** **{** **...** **}**

**}**

**Пример,**

class Person**;**

class Company

**{**

**public** Person**[]** personal**;**

**...**

// индексатор

**public** Person **this[**int index**]**

**{**

// get или set как и в свойствах можно опускать или ограничивать модификаторами

**get** **=>** personal**[**index**];**

**set** **=>** personal**[**index**]** **=** **value;**

**}**

**}**

var company **=** **new** Company **{** personal **=** **new[]** **{** **...** **}** **};**

// получаем объект из индексатора

Person person **=** company**[**0**];**

// переустанавливаем объект

company**[**0**]** **=** **new** Person**();**

## Переменные**-**ссылки и возвращение ссылки

### Переменная-ссылка

**Переменной-ссылке обязательно надо присвоить значение.**

int x **=** 5**;**

**ref** int xRef **=** **ref** x**;**

### Ссылка как результат функции

**ref** int Method**(**int**[]** array**)** // ref в сигнатуре

**{**

**return** **ref** array**[**0**];** // ref после return

**}**

**ref** int i **=** **ref** Method**(new[]** **{** 100 **});** // ref перед переменной и перед методом

**Другой пример:**

int a **=** 5**;**

int b **=** 8**;**

**ref** int pointer **=** **ref** Max**(ref** a**,** **ref** b**);**

pointer **=** 34**;** // меняем значением максимального числа

Console**.**WriteLine**(**$"a: {a} b: {b}"**);** // a: 5 b: 34

**ref** int Max**(ref** int n1**,** **ref** int n2**)**

**{**

**if** **(**n1 **>** n2**)**

**return** **ref** n1**;**

**else**

**return** **ref** n2**;**

**}**

**Такой метод не может возвращать:**

**- Значение null**

**- Константу**

**- Значение перечисления enum**

**- Свойство класса или структуры**

**- Поле для чтения (которое имеет модификатор read-only)**

## Методы расширения

**Mетоды расширения действуют на уровне пространства имен.**

**public** static class StringExtension

**{**

**public** static char GetChar**(this** string str**,** int i**)** **=>** str**[**i**];** // расширение класса string

**}**

"hello"**.**GetChar**(**1**);** // e

## Частичные классы и методы

**Mы можем иметь несколько файлов с определением одного и того же класса, и при компиляции все эти определения будут скомпилированы в одно.**

Program1**.**cs

**public** **partial** class Program **{** **public** static void Move**()** **{** **...** **}** **}**

Program2**.**cs

**public** **partial** class Program **{** **public** static void Eat**()** **{** **...** **}** **}**

Program**.**Move**();**

Program**.**Eat**();**

### Частичные методы

**По умолчанию к частичным методам применяется ряд ограничений:**

**- Они не могут иметь модификаторы доступа**

**- Они имеют тип void**

**- Они не могут иметь out-параметры**

**- Они не могут иметь модификаторы virtual, override, sealed, new или extern**

**Если же они не соответствуют какому-то из этих ограничений, то для них должна быть предоставлена реализация.**

// первая реализация класса и его методов

**public** **partial** class Person

**{**

**public** **partial** void Read**();**

**}**

// вторая реализация класса и его методов

**public** **partial** class Person

**{**

**public** **partial** void Read**()**

**{**

Console**.**WriteLine**(**"I am reading a book"**);**

**}**

**}**

## Анонимные типы

**Свойства анонимных типов доступны только для чтения.**

var user **=** **new** **{** Name **=** "Tom"**,** Age **=** 34 **};**

Console**.**WriteLine**(**user**.**Name**);**

### Инициализаторы с проекцией

class Person

**{**

**public** string Name **{** **get;** **set;** **}**

**public** Person**(**string name**)** **=>** Name **=** name**;**

**}**

Person tom **=** **new** Person**(**"Tom"**);**

int age **=** 34**;**

var student **=** **new** **{** tom**.**Name**,** age **};** // инициализатор с проекцией

Console**.**WriteLine**(**$"{student.Name} {student.age}"**);**