Оглавление

[Оглавление 1](#_Toc166073848)

[Основы 6](#_Toc166073849)

[Литералы 6](#_Toc166073850)

[Типы данных 6](#_Toc166073851)

[Консольный ввод-вывод 6](#_Toc166073852)

[Логические операции 7](#_Toc166073853)

[Операции присваивания 7](#_Toc166073854)

[Массивы 7](#_Toc166073855)

[Одномерные массивы 7](#_Toc166073856)

[Многомерные массивы 8](#_Toc166073857)

[Массив массивов 8](#_Toc166073858)

[Методы 8](#_Toc166073859)

[Передача параметров по ссылке. Выходные параметры 9](#_Toc166073860)

[Модификатор ref 9](#_Toc166073861)

[Модификатор out 9](#_Toc166073862)

[Модификатор in 9](#_Toc166073863)

[Модификатор ref readonly (c C#12) 9](#_Toc166073864)

[Массив параметров и ключевое слово params 10](#_Toc166073865)

[Локальные функции 10](#_Toc166073866)

[Конструкция switch 10](#_Toc166073867)

[Перечисления enum 11](#_Toc166073868)

[Классы 11](#_Toc166073869)

[Создание конструкторов 11](#_Toc166073870)

[Цепочка вызовов конструкторов 12](#_Toc166073871)

[Первичные конструкторы (начиная с версии C#12) 12](#_Toc166073872)

[Инициализаторы объектов 12](#_Toc166073873)

[Деконструкторы 12](#_Toc166073874)

[Структуры 13](#_Toc166073875)

[Инициализация с помощью конструктора 13](#_Toc166073876)

[Цепочка конструкторов 13](#_Toc166073877)

[Первичный конструктор 13](#_Toc166073878)

[Инициализатор структуры 14](#_Toc166073879)

[Копирование структуры с помощью with 14](#_Toc166073880)

[Модификаторы доступа 14](#_Toc166073881)

[Свойства 14](#_Toc166073882)

[Определение свойств 14](#_Toc166073883)

[Свойства только для чтения и записи 15](#_Toc166073884)

[Модификаторы доступа 15](#_Toc166073885)

[Автоматические свойства 15](#_Toc166073886)

[Блок init (начиная с версии C# 9.0) 15](#_Toc166073887)

[Сокращенная запись свойств 16](#_Toc166073888)

[Модификатор required (добавлен в C# 11) 16](#_Toc166073889)

[Статические члены и модификатор static 16](#_Toc166073890)

[Статические поля 16](#_Toc166073891)

[Статические свойства 17](#_Toc166073892)

[Статические методы 17](#_Toc166073893)

[Статический конструктор 17](#_Toc166073894)

[Статические классы 17](#_Toc166073895)

[Константы, поля и структуры для чтения 17](#_Toc166073896)

[Константы класса 17](#_Toc166073897)

[Поля для чтения и модификатор readonly 17](#_Toc166073898)

[Сравнение констант 17](#_Toc166073899)

[Структуры для чтения 18](#_Toc166073900)

[Null и ссылочные типы 18](#_Toc166073901)

[Оператор ! (null-forgiving operator) 18](#_Toc166073902)

[Null и значимые типы 18](#_Toc166073903)

[Проверка на null, операторы ?. и ?? 18](#_Toc166073904)

[Оператор ?? 19](#_Toc166073905)

[Оператор условного null ( ?. ) 19](#_Toc166073906)

[Псевдонимы типов и статический импорт 19](#_Toc166073907)

[Псевдонимы 19](#_Toc166073908)

[Статический импорт 19](#_Toc166073909)

[ООП 19](#_Toc166073910)

[Наследование 19](#_Toc166073911)

[Ключевое слово base 20](#_Toc166073912)

[Преобразование типов 20](#_Toc166073913)

[Восоходящее преобразование. Upcasting 20](#_Toc166073914)

[Нисходящее преобразование. Downcasting 20](#_Toc166073915)

[Способы преобразований 20](#_Toc166073916)

[Виртуальные методы и свойства 21](#_Toc166073917)

[Ключевое слово base 21](#_Toc166073918)

[Переопределение свойств 21](#_Toc166073919)

[Запрет переопределения методов 22](#_Toc166073920)

[Скрытие методов и свойств 22](#_Toc166073921)

[Различие переопределения и скрытия методов 22](#_Toc166073922)

[Переопределение 22](#_Toc166073923)

[Скрытие 23](#_Toc166073924)

[Абстрактные классы и члены классов 23](#_Toc166073925)

[Класс System.Object и его методы 23](#_Toc166073926)

[Обобщения (generics) 24](#_Toc166073927)

[Статические поля обобщенных классов 24](#_Toc166073928)

[Обобщенные методы 24](#_Toc166073929)

[Ограничения обобщений 24](#_Toc166073930)

[Ограничения методов 24](#_Toc166073931)

[Ограничения обобщений в типах 24](#_Toc166073932)

[Типы ограничений и стандартные ограничения 24](#_Toc166073933)

[Использование нескольких универсальных параметров 25](#_Toc166073934)

[Наследование обобщенных типов 25](#_Toc166073935)

[Исключения 25](#_Toc166073936)

[Конструкция try..catch..finally 25](#_Toc166073937)

[Фильтры исключений 25](#_Toc166073938)

[Типы исключений. Класс Exception 26](#_Toc166073939)

[Генерация исключения и оператор throw 26](#_Toc166073940)

[Делегаты, лямбды и события 27](#_Toc166073941)

[Делегаты 27](#_Toc166073942)

[Добавление методов в делегат 27](#_Toc166073943)

[Объединение делегатов 27](#_Toc166073944)

[Вызов делегата с помощью метода Invoke() 28](#_Toc166073945)

[Обобщенные делегаты (Generic) 28](#_Toc166073946)

[Делегаты как параметры методов 28](#_Toc166073947)

[Возвращение делегатов из метода 28](#_Toc166073948)

[Анонимные методы 29](#_Toc166073949)

[Лямбды 29](#_Toc166073950)

[Параметры лямбды 30](#_Toc166073951)

[Возвращение результата 30](#_Toc166073952)

[Добавление и удаление действий в лямбда-выражении, метод Invoke 31](#_Toc166073953)

[Лямбда-выражение как аргумент метода 31](#_Toc166073954)

[Лямбда-выражение как результ метода 31](#_Toc166073955)

[События 31](#_Toc166073956)

[Определение и вызов событий 31](#_Toc166073957)

[Добавление обработчика события 31](#_Toc166073958)

[Управление обработчиками (add, remove) 32](#_Toc166073959)

[Ковариантность и контравариантность делегатов 33](#_Toc166073960)

[Ковариантность 33](#_Toc166073961)

[Контрвариантность 33](#_Toc166073962)

[Ковариантность и контравариантность в обобщенных делегатах 33](#_Toc166073963)

[Делегаты Action, Predicate и Func 34](#_Toc166073964)

[Action 34](#_Toc166073965)

[Predicate 34](#_Toc166073966)

[Func 34](#_Toc166073967)

[Замыкания 34](#_Toc166073968)

[Реализация с помощью лямбда-выражений 35](#_Toc166073969)

[Применение параметров 35](#_Toc166073970)

[Интерфейсы 35](#_Toc166073971)

[Определение интерфейсов 35](#_Toc166073972)

[Применение интерфейсов 36](#_Toc166073973)

[Множественная реализация интерфейсов 37](#_Toc166073974)

[Явная реализация интерфейсов 37](#_Toc166073975)

[Модификаторы доступа 37](#_Toc166073976)

[Реализация интерфейсов в базовых и производных классах 38](#_Toc166073977)

[Наследование интерфейсов 40](#_Toc166073978)

[Интерфейсы как ограничения обобщений 40](#_Toc166073979)

[Обобщенные интерфейсы 41](#_Toc166073980)

[Копирование объектов. Интерфейс ICloneable 41](#_Toc166073981)

[Поверхностное копирование 41](#_Toc166073982)

[Глубокое копирование 41](#_Toc166073983)

[Сортировка объектов. Интерфейс IComparable 41](#_Toc166073984)

[Применение компаратора 42](#_Toc166073985)

[Ковариантность и контравариантность обобщенных интерфейсов 43](#_Toc166073986)

[Ковариантные интерфейсы 43](#_Toc166073987)

[Контравариантные интерфейсы 43](#_Toc166073988)

[Дополнительные возможности ООП 43](#_Toc166073989)

[Определение операторов 43](#_Toc166073990)

[Определение инкремента и декремента 44](#_Toc166073991)

[Определение операций true и false 44](#_Toc166073992)

[Перегрузка операций преобразования типов 45](#_Toc166073993)

[Индексаторы 45](#_Toc166073994)

[Переменные-ссылки и возвращение ссылки 46](#_Toc166073995)

[Переменная-ссылка 46](#_Toc166073996)

[Ссылка как результат функции 46](#_Toc166073997)

[Методы расширения 47](#_Toc166073998)

[Частичные классы и методы 47](#_Toc166073999)

[Частичные методы 47](#_Toc166074000)

[Анонимные типы 48](#_Toc166074001)

[Инициализаторы с проекцией 48](#_Toc166074002)

[Кортежи 48](#_Toc166074003)

[Кортеж как результат метода 49](#_Toc166074004)

[Кортеж как параметр метода 49](#_Toc166074005)

[Records (C#9) 49](#_Toc166074006)

[Сравнение records 49](#_Toc166074007)

[Оператор with 49](#_Toc166074008)

[Позиционные классы records 50](#_Toc166074009)

[Позиционные структуры для чтения 50](#_Toc166074010)

[ToString 50](#_Toc166074011)

[Наследование 50](#_Toc166074012)

[Pattern matching 51](#_Toc166074013)

[Паттерн типов 51](#_Toc166074014)

[type pattern 51](#_Toc166074015)

[constant pattern 51](#_Toc166074016)

[в конструкции switch: 51](#_Toc166074017)

[выражение when в конструкции switch: 51](#_Toc166074018)

[Паттерн свойств 51](#_Toc166074019)

[Паттерны кортежей 52](#_Toc166074020)

[Позиционный паттерн 52](#_Toc166074021)

[Реляционный и логический паттерны (C#9.0) 53](#_Toc166074022)

[relational pattern 53](#_Toc166074023)

[logical pattern 53](#_Toc166074024)

[Паттерны списков (C#11) 53](#_Toc166074025)

[Свойства коллекций 54](#_Toc166074026)

[Коллекции 54](#_Toc166074027)

[Список List<T> 54](#_Toc166074028)

[Создание пустого списка 54](#_Toc166074029)

[Создание списка, содежащего начальные значения 54](#_Toc166074030)

[Создание списка с элементами из другой коллекции 54](#_Toc166074031)

[С C#12 54](#_Toc166074032)

[Установка начальной емкости списка 54](#_Toc166074033)

[Обращение к элементам списка 54](#_Toc166074034)

[Длина списка 54](#_Toc166074035)

[Методы списка 55](#_Toc166074036)

[Двухсвязный список LinkedList<T> 56](#_Toc166074037)

[Создание связанного списка 56](#_Toc166074038)

[Свойства LinkedList 56](#_Toc166074039)

[Свойства LinkedListNode 56](#_Toc166074040)

[Циклы: 56](#_Toc166074041)

[Методы LinkedList 56](#_Toc166074042)

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}"**);**

## Кортежи

var tuple **=** **(**5**,** 10**);** // (int, int) tuple = (5, 10);

Console**.**WriteLine**(++**tuple**.**Item1**);** // 6

**можем дать названия полям кортежа:**

**(**string**,** int**,** double**)** person **=** **(**name**:**"Tom"**,** age**:**35**,** height**:**180.5**);**

**можем выполнить декомпозицию кортежа на отдельные переменные:**

var **(**name**,** age**)** **=** **(**"Tom"**,** 23**);**

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

Console**.**WriteLine**(**age**);** // 23

**можем выполнить обмен значениями:**

string java **=** "Java"**;**

string cs **=** "C#"**;**

**(**java**,** cs**)** **=** **(**cs**,** java**);**

Console**.**WriteLine**(**java**);** // C#

Console**.**WriteLine**(**cs**);** // Java

// это можно использовать, например, в пузырьковой сортировке массива:

// if (nums[i] > nums[j])

// (nums[i], nums[j]) =(nums[j], nums[i]);

### Кортеж как результат метода

var tuple **=** GetValues**();**

Console**.**WriteLine**(**tuple**.**Item1**);** // 1

Console**.**WriteLine**(**tuple**.**Item2**);** // 3

tuple **=** GetNewValues**();**

Console**.**WriteLine**(**tuple**.**a**);** // 2

Console**.**WriteLine**(**tuple**.**b**);** // 5

**(**int**,** int**)** GetValues**()** **=>** **(**1**,** 3**);**

**(**int a**,** int b**)** GetNewValues**()** **=>** **(**2**,** 5**);**

### Кортеж как параметр метода

PrintPerson**((**"Tom"**,** 37**));** // Tom - 37

void PrintPerson**((**string name**,** int age**)** person**)**

**{**

Console**.**WriteLine**(**$"{person.name} - {person.age}"**);**

**}**

## Records **(**C#9**)**

**Records могут представлять неизменяемый (immutable) тип. Начиная с версии C# 10 добавлена поддержка структур record, соответственно мы можем создавать record-классы и record-структуры.**

**public** record Person // public record class Person { ... }

**{**

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

// чтобы сделать его действительно неизменяемым,

// необходимо с помощью модификатора init сделать свойства неизменяемыми

**}**

**public** record struct Person **{** **...** **}**

### Сравнение records

**public** record Person

**{**

**public** string **{** **get;** init**;** **}**

**}**

var person1 **=** **new** Person**(**"Tom"**);**

var person2 **=** **new** Person**(**"Tom"**);**

Console**.**WriteLine**(**person1**.**Equals**(**person2**));** // true

Console**.**WriteLine**(**person1 **==** person2**);** // true

### Оператор with

**public** record Person

**{**

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

**public** string Age **{** **get;** init**;** **}**

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

**{**

Name **=** name**;** Age **=** age**;**

**}**

**}**

var tom **=** **new** Person**(**"Tom"**,** 37**);**

var sam **=** tom with **{** Name **=** "Sam" **};**

// можно скопировать полностью: var sam = tom with {};

Console**.**WriteLine**(**$"{sam.Name} - {sam.Age}"**);** // Sam - 37

### Позиционные классы records

**public** record Person**(**string Name**,** int Age**);**

**это аналогично следующей записи:**

**public** record Person

**{**

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

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

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

**{**

Name **=** name**;** Age **=** age**;**

**}**

**public** void Deconstruct**(out** string name**,** **out** int age**)** **=>** **(**name**,** age**)** **=** **(**Name**,** Age**);**

**}**

**результат можно использовать так:**

var person **=** **new** Person **(**"Tom"**,** 37**);**

var **(**personName**,** personAge**)** **=** person**;**

Console**.**WriteLine**(**personAge**);** // 37

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

**можно совмещать стандартное определение свойств и определение свойств через конструктор:**

var person **=** **new** Person**(**"Tom"**,** 37**)** **{** Company **=** "Google"**};**

**public** record Person**(**string Name**,** int Age**)** // доступны только для чтения

// (для свойств класса record по умолчанию устанавливается модификатор init)

**{**

**public** string Company **{** **get;** **set;** **}** **=** ""**;** // можно изменять

**}**

### Позиционные структуры для чтения

**public** record struct Person**(**string Name**,** int Age**);**

// по умолчанию свойства структур record имеют стандартные сеттеры

**public** **readonly** record struct Person**(**string Name**,** int Age**);**

// readonly record структура имеет свойства с модификатором init

### ToString

**Для record реализован ToString по-умолчанию.**

### Наследование

**public** record Person**(**string Name**,** int Age**);**

**public** record Employee**(**string Name**,** int Age**,** string Company**)** **:** Person**(**Name**,** Age**);**

Pattern matching

## Паттерн типов

### type pattern

**значение is тип переменная\_типа**

**Например**

**if** **(**employee **is** Manager manager**)** manager**.**Go**();**

// происходит создание переменной manager типа Manager через приведение типа Employee к Manager

### constant pattern

**if** **(**message **is** "hello"**)** **...**

**или**

**if** **(**message **is** **null)** **...**

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

### в конструкции switch:

**switch** **(**employee**)**

**{**

**case** Manager manager**:**

// приведение к типу Manager и сохранение в переменной manager

manager**.**Go**();**

**break;**

**case** **null:**

**...**

**break;**

**...**

**}**

### выражение when в конструкции switch:

Employee employee **=** **new** Manager**()** **{** Value **=** **true** **};**

**switch** **(**employee**)**

**{**

**case** Manager manager **when** manager**.**Value**:**

manager**.**Go**();**

**break;**

**...**

**}**

## Паттерн свойств

**{**свойство**:** значение**}**

class Person

**{**

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

**public** string Status **{** **get;** **set;** **}** **=** ""**;**

**public** string Language **{** **get;** **set;** **}** **=** ""**;**

**}**

**Проверка по классу и свойству:**

**if** **(**person **is** Person **{** Language**:** "english"**,** Status**:** "admin" **}** **...**

**В конструкции switch:**

string message **=** person **switch**

**{**

**{** Language**:** "french" **}** **=>** "Salut!"**,**

// можно определять переменные (например, name) и передавать им значения //свойств

**{** Language**:** "english"**,** Name**:** var name **}** **=>** $"Hello, {name}"**,**

**...**

**}**

**С C#10:**

class Employee

**{**

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

**}**

class Company

**{**

**public** string Title **{** **get;** **}**

**}**

Employee employee **=** **new** Employee **{** MyCompany **=** **new** Company **{** Title **=** "Microsoft" **}** **};**

**if** **(**employee **is** Employee **{** Company**:** **{** Title**:** "Microsoft" **}** **})** **...**

**то же самое:**

**if** **(**employee **is** Employee **{** Company**.**Title**:** "Microsoft" **})** **...**

## Паттерны кортежей

string MyMethod**(**string a**,** string b**)** **=>** **(**a**,** b**)** **switch**

**{**

**(**"C"**,** "D"**)** **=>** "CD"**,**

**(**\_**,** "E"**)** **=>** "E"**,**

\_ **=>** "F"

**};**

## Позиционный паттерн

**Позиционный паттерн применяется к типу, у которого определен метод деконструктора.**

class MyClass

**{**

**public** string A **{** **get;** **set;** **}** **=** ""**;**

**public** string B **{** **get;** **set;** **}** **=** ""**;**

**public** void Deconstruct**(out** string a**,** **out** string b**)**

**{**

a **=** A**;**

b **=** B**;**

**}**

**}**

string MyMethod**(**MyClass myClass**)** **=>** myClass **switch**

**{**

**(**"C"**,** "D"**)** **=>** "CD"**,**

**(**\_**,** "E"**)** **=>** "E"**,**

**(**var a**,** var b**)** **=>** $"{a} {b}"**;**

\_ **=>** "F"

**};**

## Реляционный и логический паттерны **(**C#9.0**)**

### relational pattern

int MyMethod**(**int a**)** **=>** a **switch**

**{**

**<** 0 **=>** 0**,**

**>** 10 **=>** 2**,**

\_ **=>** 1

**}**

### logical pattern

int MyMethod**(**int a**)** **=>** a **switch**

**{**

**<** 0 or **>** 10 **=>** 0**,**

**>=** 3 and **<=** 5 **=>** 2**,**

not 7 **=>** 1**,**

\_ **=>** 7

**}**

## Паттерны списков **(**C#11**)**

string MyMethod**(**int**[]** array**)** **=>** array **switch**

**{**

**[**1**,** 2**,** 3**]** **=>** "1"**,**

**[**1**,** 2**,** **..,** 5**]** **=>** "2"**,**

// соответствует массиву из произвольного количества элементов,

// первые 2 из которых - 1, 2, затем - любые, а заканчивается на 5

**[**1**,**\_**]** **=>** "3"**,**

// соответствует массиву из 2 элементов,

// первый из которых - 1, а второй - любое число

**[**\_**,** **..,** \_**]** **=>** "4"**,**

// соответствует массиву как минимум из 2 элементов

**[**var a**,** var b**,** **..,** var d**]** **=>** $"{a}, {b}, {d}"**,**

// получение значений из элементов массива

**[**1**,** **..** var a**,** 2**]** **=>** $"{string.Join("**,**", a)}"**,**

// a - массив, в котором сохраняются элементы между 1 и 2

**[]** **=>** "6"**,**

\_ **=>** "7"

**};**

**Аналогично со списками. Также можно использовать в конструкции if:**

**if** **(**array **is** **[**"a"**,** "b"**,** "c"**])** **...**

### Свойства коллекций

// можно объединить паттерн списков и паттерн свойств

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

**if** **(**array **is** **{** Length**:** 3**}** and **[**var a**,** var b**,** var c**])** // свойство массива Length

Console**.**WriteLine**(**$"{a}, {b}, {c}"**);**

Коллекции

## Список List**<**T**>**

**Большая часть классов коллекций содержится в пространстве имен** System**.**Collections**.**Generic

### Создание пустого списка

List**<**string**>** list **=** **new** List**<**string**>();**

### Создание списка, содежащего начальные значения

List**<**string**>** list **=** **new** List**<**string**>()** **{** "A"**,** "B"**,** "C" **};**

### Создание списка с элементами из другой коллекции

var list **=** **new** List**<**string**>()** **{** "A"**,** "B"**,** "C" **};**

var newList **=** **new** List**<**string**>(**list**);**

**или можно так:**

var newList **=** **new** List**<**string**>(**list**)** **{** "D"**,** "E" **};**

### С C#12

List**<**string**>** list **=** **{** "A"**,** "B"**,** "C" **};**

List**<**string**>** emptyList **=** **[];** // пустой список

### Установка начальной емкости списка

List**<**string**>** list **=** **new** List**<**string**>(**10**);**

**или так:**

list**.**Capacity **=** 10**;**

### Обращение к элементам списка

string element **=** list**[**0**];** // обращение к первому элементу коллекции

### Длина списка

int size **=** list**.**Count**;**

### Методы списка

**void Add(T item): добавление нового элемента в список**

**void AddRange(IEnumerable<T> collection): добавление в список коллекции или массива**

**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.**

**void CopyTo(T[] array): копирует список в массив array**

**- void CopyTo(int index, T[] array, int arrayIndex, int count): копирует из списка начиная с индекса index элементы, количество которых равно count, и вставляет их в массив array начиная с индекса arrayIndex**

**- bool Contains(T item): возвращает true, если элемент item есть в списке**

**- void Clear(): удаляет из списка все элементы**

**- bool Exists(Predicate<T> match): возвращает true, если в списке есть элемент, который соответствует делегату match**

**- T? Find(Predicate<T> match): возвращает первый элемент, который соответствует делегату match. Если элемент не найден, возвращается null**

**- T? FindLast(Predicate<T> match): возвращает последний элемент, который соответствует делегату match. Если элемент не найден, возвращается null**

**- List<T> FindAll(Predicate<T> match): возвращает список элементов, которые соответствуют делегату match**

**- int IndexOf(T item): возвращает индекс первого вхождения элемента в списке**

**- int LastIndexOf(T item): возвращает индекс последнего вхождения элемента в списке**

**- List<T> GetRange(int index, int count): возвращает список элементов, количество которых равно count, начиная с индекса index.**

**- void Insert(int index, T item): вставляет элемент item в список по индексу index. Если такого индекса в списке нет, то генерируется исключение**

**- void InsertRange(int index, collection): вставляет коллекцию элементов collection в текущий список начиная с индекса index. Если такого индекса в списке нет, то генерируется исключение**

**- bool Remove(T item): удаляет элемент item из списка, и если удаление прошло успешно, то возвращает true. Если в списке несколько одинаковых элементов, то удаляется только первый из них**

**- void RemoveAt(int index): удаление элемента по указанному индексу index. Если такого индекса в списке нет, то генерируется исключение**

**- void RemoveRange(int index, int count): параметр index задает индекс, с которого надо удалить элементы, а параметр count задает количество удаляемых элементов.**

**- int RemoveAll((Predicate<T> match)): удаляет все элементы, которые соответствуют делегату match. Возвращает количество удаленных элементов**

**- void Reverse(): изменяет порядок элементов**

**- void Reverse(int index, int count): изменяет порядок на обратный для элементов, количество которых равно count, начиная с индекса index**

**- void Sort(): сортировка списка**

**- void Sort(IComparer<T>? comparer): сортировка списка с помощью объекта comparer, который передается в качестве параметра**

## Двухсвязный список LinkedList**<**T**>**

### Создание связанного списка

LinkedList**<**string**>** list **=** **new** LinkedList**<**string**>();**

**можно из List сделать LinkedList так:**

List**<**string**>** list **=** **new** List**<**string**>()** **{** "A"**,** "B"**,** "C" **};**

LinkedList**<**string**>** linkedList **=** **new** LinkedList**<**string**>(**list**);**

### Свойства LinkedList

**- Count: количество элементов в связанном списке**

**- First: первый узел в списке в виде объекта LinkedListNode<T>**

**- Last: последний узел в списке в виде объекта LinkedListNode<T>**

### Свойства LinkedListNode

**- Value: возвращает или устанавливает само значение узла, представленное типом T**

**- Next: возвращает ссылку на следующий элемент типа LinkedListNode<T> в списке. Если следующий элемент отсутствует, то имеет значение null**

**- Previous: возвращает ссылку предыдущий элемент типа LinkedListNode<T> в списке. Если предыдущий элемент отсутствует, то имеет значение null**

### Циклы:

// от начала до конца

var node **=** linkedList**.**First**;**

**while(**node **!=** **null)**

**{**

Console**.**WriteLine**(**node**.**Value**);**

node **=** node**.**Next**;**

**}**

// от конца до начала

var node **=** linkedList**.**Last**;**

**while(**node **!=** **null)**

**{**

Console**.**WriteLine**(**node**.**Value**);**

node **=** node**.**Previous**;**

**}**

### Методы LinkedList

**- AddAfter(LinkedListNode<T> node, LinkedListNode<T> newNode): вставляет узел newNode в список после узла node.**

**- AddAfter(LinkedListNode<T> node, T value): вставляет в список новый узел со значением value после узла node.**

**- AddBefore(LinkedListNode<T> node, LinkedListNode<T> newNode): вставляет в список узел newNode перед узлом node.**

**- AddBefore(LinkedListNode<T> node, T value): вставляет в список новый узел со значением value перед узлом node.**

**- AddFirst(LinkedListNode<T> node): вставляет новый узел в начало списка**

**- AddFirst(T value): вставляет новый узел со значением value в начало списка**

**- AddLast(LinkedListNode<T> node): вставляет новый узел в конец списка**

**- AddLast(T value): вставляет новый узел со значением value в конец списка**

**- RemoveFirst(): удаляет первый узел из списка. После этого новым первым узлом становится узел, следующий за удаленным**

**- RemoveLast(): удаляет последний узел из списка**

## Очередь Queue

### Создание очереди

Queue**<**string**>** queue **=** **new** Queue**<**string**>();**

**можно указать емкость очереди:**

Queue**<**string**>** queue **=** **new** Queue**<**string**>(**16**);**

**также можно инициализировать очередь элементами из другой коллекции или массивом:**

var list **=** **new** List**<**string**>** **{** "A"**,** "B"**,** "C" **};**

Queue**<**string**>** queue **=** **new** Queue**<**string**>(**list**);**

**для перебора можно использовать цикл foreach:**

**foreach** **(**var q **in** queue**)** **...**

queue**.**Count **- количество элементов в очереди**

### Методы Queue

**- void Clear(): очищает очередь**

**- bool Contains(T item): возвращает true, если элемент item имеется в очереди**

**- T Dequeue(): извлекает и возвращает первый элемент очереди (есть Exception)**

**- void Enqueue(T item): добавляет элемент в конец очереди**

**- T Peek(): просто возвращает первый элемент из начала очереди без его удаления (есть Exception)**

**- bool TryDequeue(out T result): передает в переменную result первый элемент очереди с его удалением из очереди, возвращает true, если очередь не пуста и элемент успешно получен**

**- bool TryPeek(out T result): передает в переменную result первый элемент очереди без его извлечения из очереди, возвращает true, если очередь не пуста и элемент успешно получен**