# **ЛК 3-4.** Введение в язык C#

# Введение в язык C#

## МЕСТО ЯЗЫКА C# СРЕДИ ДРУГИХ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ.

# Место языка C# среди других языков программирования

C# (произносится как "си шарп") — современный *объектно-ориентированный* и *типобезопасный* язык программирования.

C# относится к семейству языков C

# Место языка C# среди других языков программирования

Главный архитектор языка с момента его первой версии - Андерс Хейлсберг (создатель Turbo Pascal и архитектор Delphi).

# Место языка C# среди других языков программирования

Разработка Microsoft много особенностей унаследовала у Delphi, Smalltalk и Java. При этом создатели нового языка исключили из своего детища многие практики и спецификации, считающиеся «проблемными»

# Место языка C# среди других языков программирования

Язык C # не зависит от платформы и работает с рядом платформ (под управлением .NET)

# Место языка C# среди других языков программирования

Первая версия языка вышла вместе с релизом Microsoft Visual Studio .NET в феврале 2002 года.

Текущей версией языка является версия C# 10.0, которая вышла 8 ноября 2021 года вместе с релизом .NET 6.

# Место языка C# среди других языков программирования

**Индекс TIOBE** (*TIOBE programming community index*) — индекс, оценивающий популярность языков программирования, на основе подсчёта результатов ***поисковых запросов***, содержащих название языка

# Место языка C# среди других языков программирования

https://www.tiobe.com/tiobe-index/

| Dec 2021 | Dec 2020 | Change | Programming Language | Ratings | Change |
|---|---|---|---|---|---|
| 1 | 3 | ^ | Python | 12.90% | +0.69% |
| 2 | 1 | v | C | 11.80% | -4.69% |
| 3 | 2 | v | Java | 10.12% | -2.41% |
| 4 | 4 | | C++ | 7.73% | +0.82% |
| 5 | 5 | | C# | 6.40% | +2.21% |
| 6 | 6 | | Visual Basic | 5.40% | +1.48% |
| 7 | 7 | | JavaScript | 2.30% | -0.06% |

# Что можно создать на языке C#

- Кроссплатформенные настольные приложения (Desktop) - .Net MAUI, WPF
- Игры с использованием движка Unity
- Web-приложения (full-stack) – ASP.Net Core + Blazor
- Web-сервисы
- Программы для работы с нейросетями и машинным обучением — ML.NET
- Мобильные приложения (для Android, Ios) – Xamarin, .Net MAUI

# Введение в язык C#

## ОСНОВЫ СИНТАКСИСА ЯЗЫКА C#

# Основы синтаксиса языка C#

Синтаксис языка C# во многом аналогичен синтаксису C/C++.

- Регистрозависимый.
- Те же правила формирования имен переменных и типов.
- В C# ни одна функция не может существовать вне класса.

# Основы синтаксиса языка C#

Выражения C# разделяются символом ;

Количество пробелов и переводов строки в выражениях значения не имеет

```
int a = 20;
                int b
                          =
              8;
```

# Основы синтаксиса языка C#. Комментарии

```
// Строчный комментарий

/* Блочный
комментарий */

/// <summary>
/// Комментарии для документации могут
/// содержать специальные XML-тэги.
/// </summary>
/// <param name="args"></param>
```

# Основы синтаксиса языка C#

В C# для имен **общедоступных** полей, свойств, методов, классов принято использовать нотацию паскаля (PascalCase), когда все слова названия начинаются с заглавной буквы.

NewItem

SelectedModeChanged

# Основы синтаксиса языка C#

В C# для имен **приватных и локальных** полей, принято использовать нотацию верблюда (CamelCase), в которой в отличие от PascalCase первая буква маленькая:

isVisible
startRotationIndex
_dataContext

# Основы синтаксиса языка C#

Группа кода (программный блок) заключается в фигурные скобки {}.

*В C# принято открывающуюся и закрывающуюся скобку располагать на одном уровне*:

```
{
        int a = 20;
        int b = 8;
}
```

# Введение в язык C#

## КОНСОЛЬНЫЙ ВВОД-ВЫВОД

# Консольный ввод-вывод

Для работы с консолью используются статические методы класса **Console**

# Консольный ввод-вывод

```
string name = "Tom";
int age = 34;
double height = 1.7;

Console.WriteLine("Имя:{0} Возраст:{2} Рост:{1}м", name, height, age);

Console.WriteLine($"Имя: {name}  Возраст: {age}  Рост: {height}м");

Console.ReadKey();
```

также можно задать:

`{0, 10}` - ширина поля

`{0:#.###}` - формат числа

# Консольный ввод-вывод

**Console.Write()**, работает точно так же, как и **Console.WriteLine()** за тем исключением, что не добавляет переход на следующую строку.

Для перехода на следующую строку можно записать:

```
Console.Write(Environment.NewLine);
```

или

```
Console.Write("\n");
```

# Консольный ввод-вывод

Для ввода с консоли используется метод **Console.ReadLine()**.

Он позволяет получить введенную строку.

```
string data = Console.ReadLine();
```

# Консольный ввод-вывод

Метод **Console.ReadLine()** возвращает строку. Для преобразования строки в число можно использовать методы:

**Convert.ToInt32()** (преобразует к типу int)
**Convert.ToDouble()** (преобразует к типу double)
**Convert.ToDecimal()** (преобразует к типу decimal)

# Консольный ввод-вывод

## (Управляющие последовательности)

| | |
|---|---|
| \a | Звуковой сигнал (звонок) |
| \b | Возврат на одну позицию |
| \f | Перевод страницы (переход на новую страницу) |
| \n | Новая строка (перевод строки) |
| \r | Возврат каретки |
| \t | Горизонтальная табуляция |
| \v | Вертикальная табуляция |
| \0 | Пустой символ |
| \' | Одинарная кавычка |
| \" | Двойная кавычка |
| \\ | Обратная косая черта |

# Введение в язык C#

## ПЕРЕМЕННЫЕ, КОНСТАНТЫ, ЛИТЕРАЛЫ

# Переменные, константы, литералы

Переменная представляет именованную область памяти, в которой хранится значение определенного типа. Переменная имеет тип, имя и значение. Тип определяет, какого рода информацию может хранить переменная.

<тип> <имя переменной>

```
int a;
string b = "text";
```

# Переменные, константы, литералы

- имя может содержать любые цифры, буквы и символ подчеркивания, при этом первый символ в имени должен быть буквой или символом подчеркивания
- в имени не должно быть знаков пунктуации и пробелов
- имя не может быть ключевым словом языка C#.

# Переменные, константы, литералы

**Константа**, в отличие от переменной, не может менять значение в процессе работы программы.

**Константа** должна быть обязательно инициализирована при определении.

Для определения констант используется ключевое слово **const**, которое указывается перед типом константы:

```
const int zeroValue = 0;
```

# Переменные, константы, литералы

**Литералы** представляют неизменяемые значения (иногда их еще называют константами).

Литералы можно передавать переменным в качестве значения.

Литералы бывают логическими, целочисленными, вещественными, символьными и строчными.

# Переменные, константы, литералы

Логические литералы: `true` (истина) и `false` (ложь).

Целочисленные литералы: `1`, `-7`, `0b100001`, `0x0A`.

Вещественные литералы: `-0.38`, `1.2E-1`.

Символьные литералы: `'A'`, `'\n'`, `'\x78'`, `'\u0421'`,

```
"Компания \"Рога и копыта\"",
@"There is \t no tab"

null
```

# Переменные, константы, литералы

`123x0   -14u    234L  -13ul`

- U u  – uint, ulong
- L l – long, ulong
- UL, Ul, uL, ul, LU, Lu, lU, lu – ulong

`12.56   12.34f    12.34d    12.34m`

- f F – float
- d D – double
- m M – decimal

# Область видимости (контекст) переменных

Каждая переменная доступна в рамках определенного контекста или области видимость. Вне этого контекста переменная уже не существует.

Существуют различные контексты:

- Контекст класса. Переменные, определенные на уровне класса, доступны в любом методе этого класса
- Контекст метода. Переменные, определенные на уровне метода, являются локальными и доступны только в рамках данного метода. В других методах они недоступны
- Контекст блока кода. Переменные, определенные на уровне блока кода, также являются локальными и доступны только в рамках данного блока. Вне своего блока кода они недоступны.

# Модификатор readonly

Для полей пользовательских типов возможно применение модификатора `readonly`, который фактически превращает их в константу.

Однако в отличие от констант, тип такого поля может быть любым:

```
public readonly int Age;
```

# Модификаторы доступа

`public` публичный, общедоступный класс или член класса. Такой член класса доступен из любого места в коде, а также из других программ и сборок.

`private` закрытый класс или член класса. Представляет полную противоположность модификатору `public`. Такой закрытый класс или член класса доступен только из кода в том же классе или контексте.

`protected` такой член класса доступен из любого места в текущем классе или в производных классах. При этом производные классы могут располагаться в других сборках.

`internal` класс и члены класса с подобным модификатором доступны из любого места кода в той же сборке, однако он недоступен для других программ и сборок (как в случае с модификатором `public`).

`protected internal` совмещает функционал двух модификаторов. Классы и члены класса с таким модификатором доступны из текущей сборки и из производных классов.

`private protected` такой член класса доступен из любого места в текущем классе или в производных классах, которые определены в той же сборке.

# Введение в язык C#

## ТИПЫ ДАННЫХ В ЯЗЫКЕ C#

# СИСТЕМА ТИПОВ

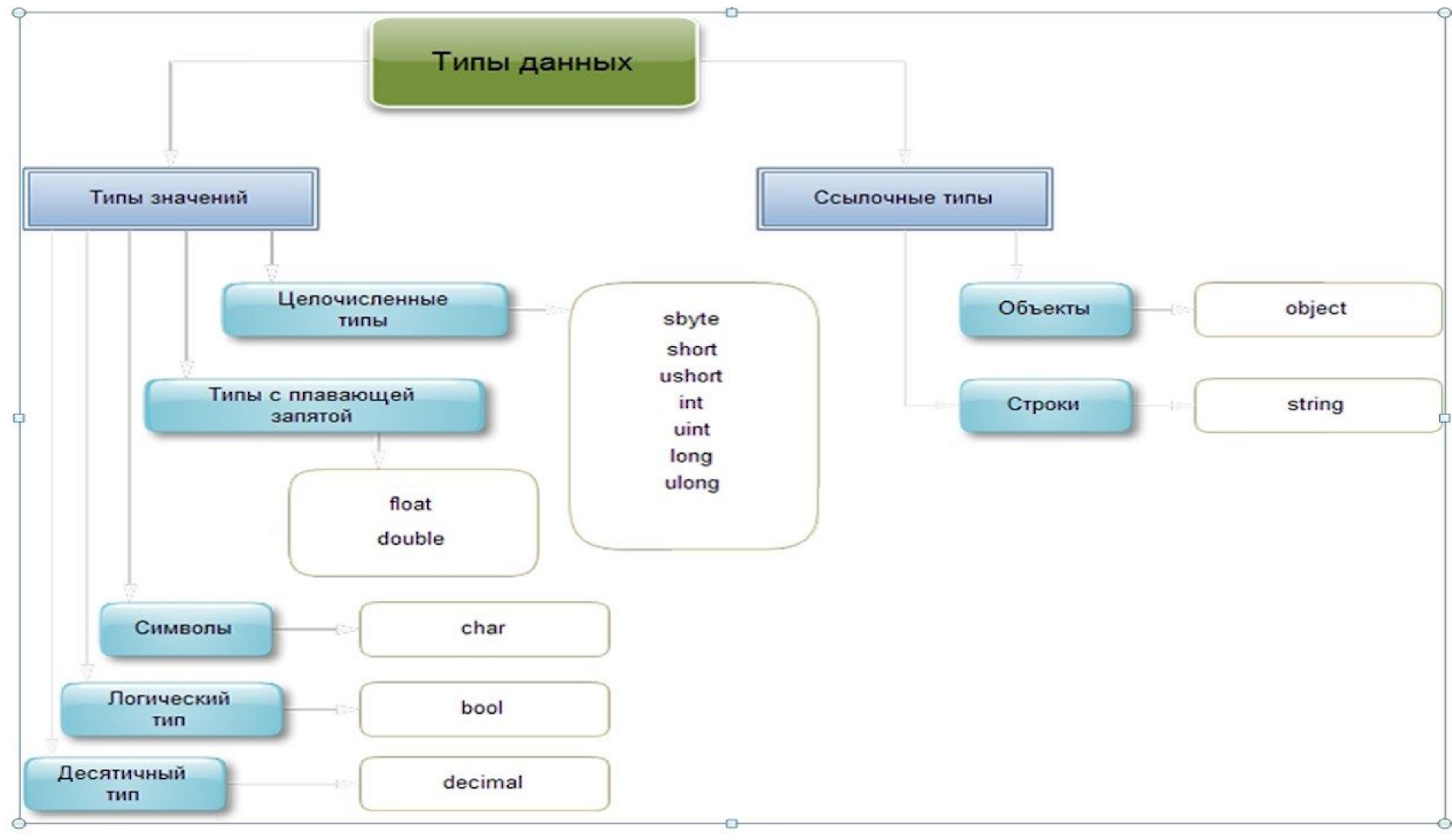
- Переменная структурного типа содержит непосредственно данные и размещается в стеке.

*Структурными типами являются примитивные типы, перечисления и структуры.*

- Переменная ссылочного типа, далее называемая *объектом*, содержит ссылку на данные, которые размещены в управляемой динамической памяти.

*Ссылочные типы – это классы, интерфейсы, массивы и делегаты.*

# СИСТЕМА ТИПОВ



# Числовые типы

| Категория | Размер (бит) | Имя типа | Диапазон/Точность |
|---|---|---|---|
| Знаковые целые | 8 | ***sbyte*** | –128...127 |
| | 16 | ***short*** | –32 768...32 767 |
| | 32 | ***int*** | –2 147 483 648...2 147 483 647 |
| | 64 | ***long*** | -9 223 372 036 854 775 808...9 223 372 036 854 775 807 |
| Беззнаковые целые | 8 | ***byte*** | 0...255 |
| | 16 | ***ushort*** | 0...65535 |
| | 32 | ***uint*** | 0...4294967295 |
| | 64 | ***ulong*** | 0...18446744073709551615 |
| Вещественные | 32 | ***float*** | Точность: от $1.5 \times 10^{-45}$ до $3.4 \times 10^{38}$, 7 цифр |
| | 64 | ***double*** | Точность: от $5.0 \times 10^{-324}$ до $1.7 \times 10^{308}$, 15 цифр |
| | 128 | ***decimal*** | Точность: от $1.0 \times 10^{-28}$ до $7.9 \times 10^{28}$, 28 цифр |

# Строки

При работе с символами и строками в C# используется кодировка Unicode.

Тип char представляет символ в 16-битной Unicode-кодировке, тип `string` – это последовательность Unicode-символов.

Хотя тип `string` относится к примитивным, переменная этого типа хранит адрес строки в динамической памяти.

# Синонимы типов в Framework Class Library

Имя примитивного типа в языке C# является синонимом соответствующего типа Framework Class Library.

Например:

- типу `int` в C# соответствует тип `System.Int32`,
- типу `float` – тип `System.Single` и т. д.

# Неявная типизация

При использовании ключевого слова var компилятор сам определяет тип данных.

```
var hello = "Hell to World";
var c = 20;
```

Но для этого необходимо сразу присваивать значение переменной и нельзя использовать литерал `null`.

# Пользовательские типы

1. **Класс** – тип, поддерживающий всю функциональность объектно-ориентированного программирования, включая наследование и полиморфизм.

2. **Структура** – тип, обеспечивающий всю функциональность ООП, кроме наследования. Структура в C# очень похожа на класс, за исключением метода размещения в памяти и отсутствия поддержки наследования.

3. **Интерфейс** – абстрактный тип, реализуемый классами и структурами для обеспечения оговоренной функциональности.

4. **Массив** – пользовательский тип для представления упорядоченного набора значений некоторых (примитивных или пользовательских) типов.

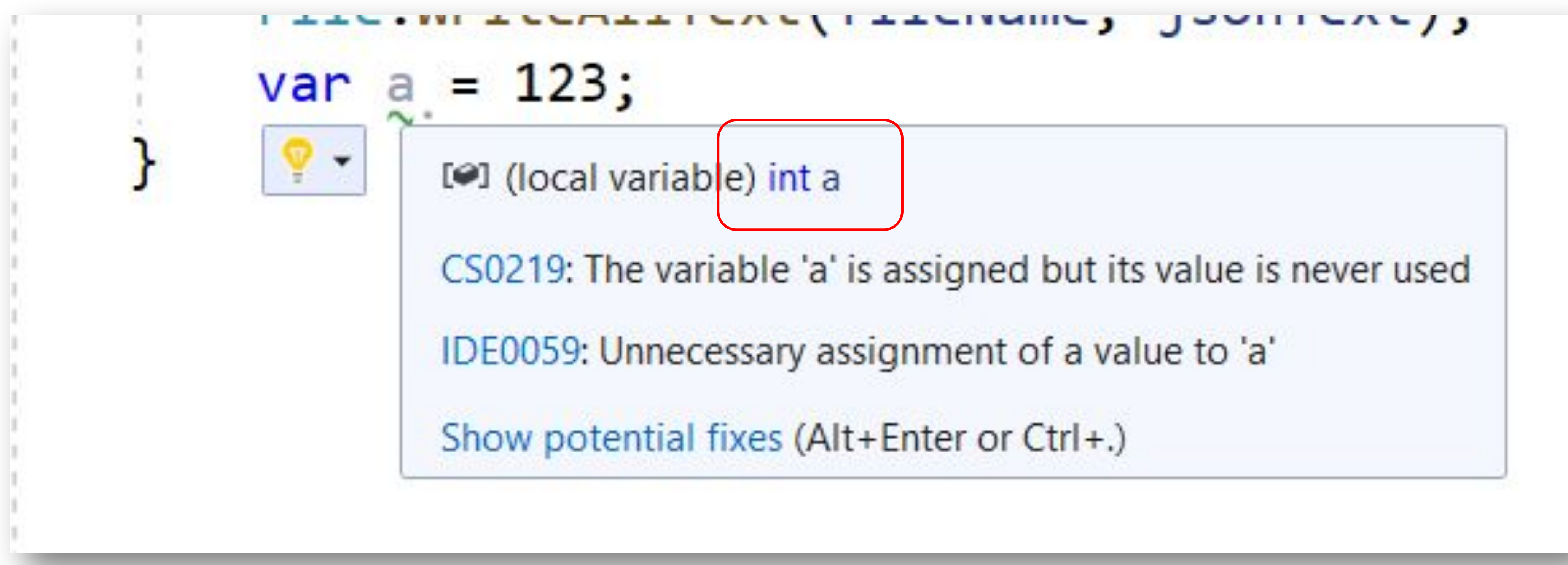
5. **Перечисление** – тип, содержащий в качестве членов именованные целочисленные константы.

# Целочисленные типы

Если литерал не имеет суффикса, его типом будет первый из следующих типов, в котором может быть представлено его значение:

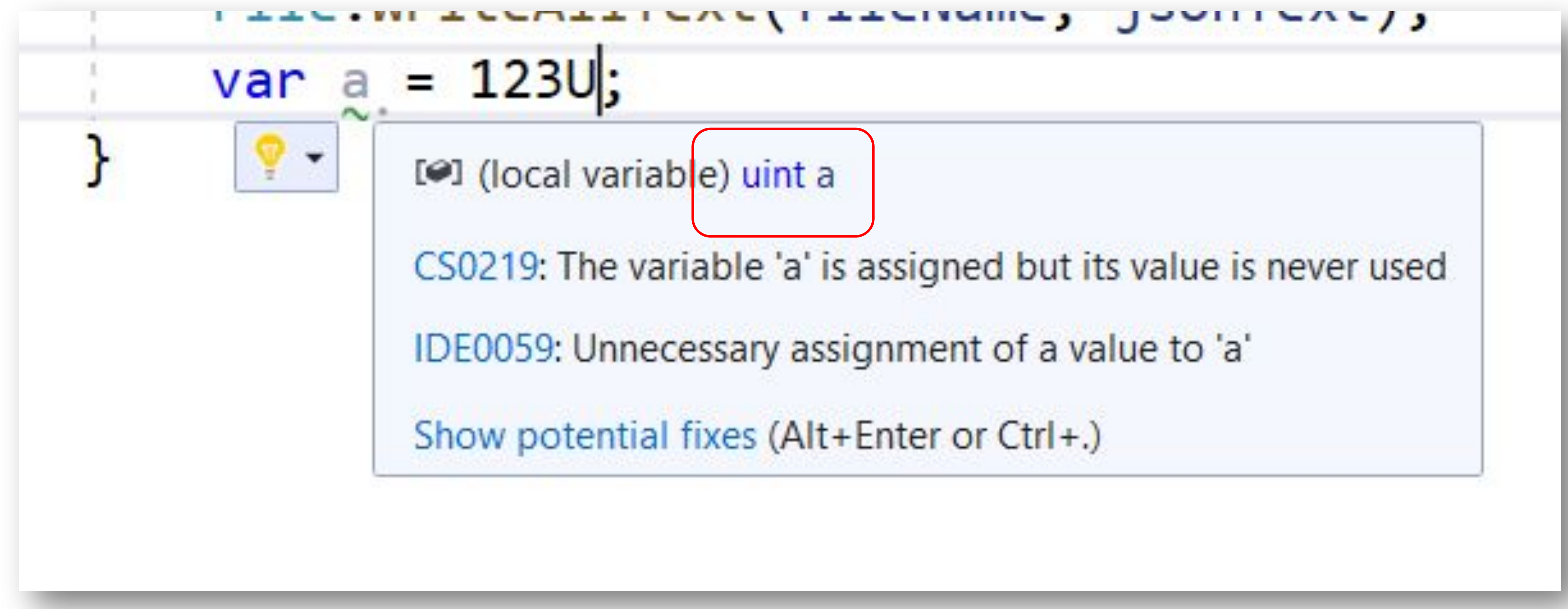
**int, uint, long, ulong**.

# Целочисленные типы



# Целочисленные типы

- Если у литерала есть суффикс **U** или **u**, его типом будет первый из следующих типов, в котором может быть представлено его значение: **uint, ulong**.
- Если у литерала есть суффикс **L** или **l**, его типом будет первый из следующих типов, в котором может быть представлено его значение: **long, ulong**.


var a = 123U;
}
(local variable) uint a
CS0219: The variable 'a' is assigned but its value is never used
IDE0059: Unnecessary assignment of a value to 'a'
Show potential fixes (Alt+Enter or Ctrl+.)

# Типы с плавающей точкой

| Тип | Системный тип | Разрядность в битах | Диапазон |
|---|---|---|---|
| float | System.Single | 32 | $-3.4*10^{38}$ : $3.4*10^{38}$ |
| double | System.Double | 64 | $\pm5.0*10^{-324}$ : $\pm1.7*10^{308}$ |
| decimal | System.Decimal | 128 | 1E-28 : 7,9E+28 |

# Типы с плавающей точкой

Если литерал с плавающей точкой не имеет суффикса, его типом будет **double**

Для явного указания типа **float** используется суффикс **F (f)**

Для явного указания типа **decimal** используется суффикс **M (m)**

# Типы с плавающей точкой

Точность чисел с плавающей точкой:

**float** : 6-9 знаков
**double** : 15-17 знаков
**decimal** : 28-29 знаков

# Строки

В C# есть символьный класс Char, основанный на классе System.Char и использующий двухбайтную кодировку Unicode представления символов. Для этого типа в языке определены символьные константы - символьные литералы. Константу можно задавать:

- символом, заключенным в одинарные кавычки;
- escape-последовательностью, задающей код символа;
- Unicode-последовательностью, задающей Unicode-код символа.

```
char ch1='A', ch2 ='\x5A', ch3='\u005A';
```

# Строки

Основным типом при работе со строками в C# является класс string, задающий строки переменной длины. Класс string относится к ссылочным типам.

```
string s;
s = "jgjfdgjhkfhjkj";          // неявный вызов конструктора
s = new string('F', 10);
char[] s10 = { 'f', 'g', 'h', 'y' };// нельзя как в C++ "fghy"
s = new string(s10);           //fghy
s = new string(s10, 1, 2);       //gh
Console.WriteLine(s[1]);        //h
```

# Строки

Операция присваивания строк имеет важную особенность. Поскольку `string` – это ссылочный тип, то в результате присваивания создается ссылка на константную строку, хранимую в динамической памяти.

С одной и той же строковой константой в динамической памяти может быть связано несколько переменных.

Но когда одна из переменных получает новое значение, она связывается с новым константным объектом в динамической памяти.

Остальные переменные сохраняют свои связи.

Для программиста это означает, что семантика присваивания строк аналогична семантике присваивания структурных типов.

# Строки

В отличие от других ссылочных типов операции, проверяющие эквивалентность строк, сравнивают значения строк, а не ссылки. Эти операции выполняются как над структурными типами.

Возможность взятия индекса при работе со строками отражает тот факт, что строку можно рассматривать как массив и получать каждый ее символ.

Внимание: символ строки доступен только для чтения, но не для записи.

# Строки

В языке C# существует понятие *неизменяемый класс* (*immutable class*).

Для такого класса невозможно изменить значение объекта при вызове его методов.

К неизменяемым классам относится и класс System.String.

Ни один из методов этого класса не меняет значения существующих объектов.

Конечно, некоторые из методов создают новые значения и возвращают в качестве результата новые строки.

# Статические элементы класса System.String

| Имя элемента | Описание |
| --- | --- |
| Empty | Возвращается пустая строка. Свойство со статусом readonly |
| Compare() | Сравнение двух строк. Можно учесть регистр и локаль |
| CompareOrdinal() | Сравнение двух строк. Сравниваются коды символов |
| Concat() | Конкатенация строк |
| Copy() | Создается копия строки |
| Format() | Выполняет форматирование строки в соответствии с заданными спецификациями формата |
| Join() | Конкатенация массива строк в единую строку. При конкатенации между элементами массива вставляются разделители. Обратная операция к Split() |

# Экземплярные методы класса System.String

| Имя метода | Описание |
|---|---|
| `Insert()` | Вставляет подстроку в заданную позицию |
| `Remove()` | Удаляет подстроку в заданной позиции |
| `Replace()` | Заменяет подстроку в заданной позиции на новую подстроку |
| `Substring()` | Выделяет подстроку в заданной позиции |
| `IndexOf()`, `IndexOfAny()`, `LastIndexOf()`, `LastIndexOfAny()` | Определяются индексы первого и последнего вхождения заданной подстроки или любого символа из заданного набора |
| `StartsWith()`, `EndsWith()` | Возвращается ***true*** или ***false***, в зависимости от того, начинается или заканчивается строка заданной подстрокой |
| `PadLeft()`, `PadRight()` | Выполняет набивку нужным числом пробелов в начале и в конце строки |
| `Trim()`, `TrimStart()`, `TrimEnd()` | Удаляются пробелы в начале и в конце строки, или только с одного ее конца |
| `ToCharArray()` | Преобразование строки в массив символов |

# Введение в язык C#

## ПРЕОБРАЗОВАНИЕ ТИПОВ

# Преобразования типов

Преобразование типов необходимо:

- Если операнды имеют разные типы
- Когда тип операндов не согласован с типом операции (для сложения byte должны быть приведены к int, поскольку сложение не определено над байтами.)
- При выполнении присваивания x = e тип источника e и тип цели x должны быть согласованы.
- При вызове метода также должны быть согласованы типы фактического и формального аргументов.

# Сужающие и расширяющие преобразования типов

Расширяющие преобразования расширяют размер объекта в памяти.

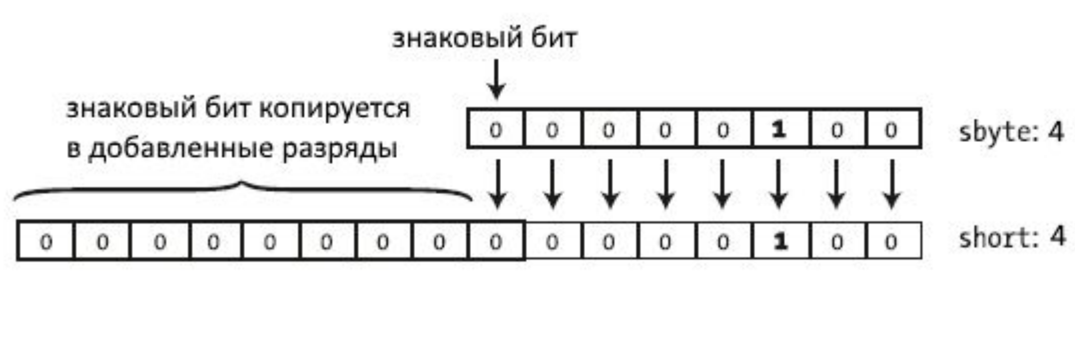
```
byte a = 4;             // 00000100
ushort b = a;   // 0000000000000100
```

Сужающие преобразования, наоборот, сужают значение до типа меньшей разрядности.

```
ushort a = 4;
byte b = (byte) a;
```

# Явные и неявные преобразования

Расширяющие преобразования обычно **неявные** (implicit).
Есть особенность преобразования из знаковых типов в беззнаковые и наоборот

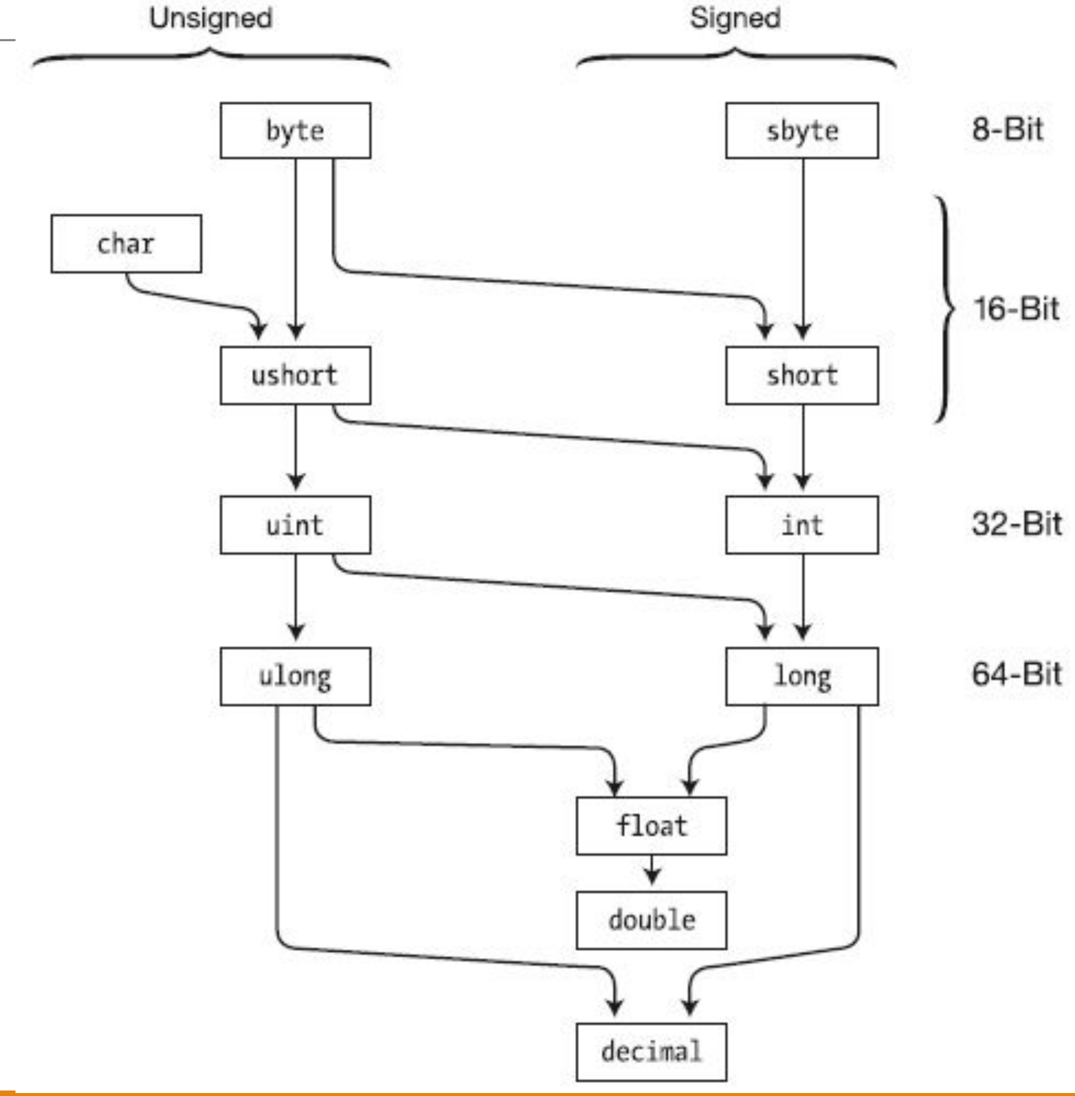


# Явные преобразования типов

При явных преобразованиях (explicit) мы сами должны применить операцию преобразования ().

```
int a = 4;
int b = 6;
byte c = (byte)(a + b);
```

# Явные преобразования типов



# Контролируемый и неконтролируемый контент

Для более гибкого контроля значений, получаемых при работе с числовыми выражениями, в языке C# предусмотрено использование контролируемого и неконтролируемого контента.

*Контролируемый контент* объявляется при помощи ключевого слова `checked` перед выражением или блоком. В этом случае, если преобразовании типов вызовет переполнение, то генерируется либо ошибка компиляции (для константных выражений), либо ошибка времени выполнения (для выражений с переменными).

Если контент неконтролируемый (по умолчанию или с использованием слова `unchecked`) - ошибки не возникает, данные урезаются.

# Контролируемый и неконтролируемый контент

```
try
{
    int a = 33;
    int b = 600;
    byte c = checked((byte)(a + b));
    Console.WriteLine(c);
}
catch (OverflowException ex)
{
    Console.WriteLine(ex.Message);
}
```

# Упаковка и распаковка

В C# допускается рассмотрение значений структурных типов как переменных типа object. Преобразование в объект называется *операцией упаковки* (*boxing*), обратное преобразование – *операцией распаковки* (*unboxing*).

```
int i = 123;
object o = i;          // Упаковка (автоматическая)
int j = (int)o;        // Распаковка
```

Возможность автоматического преобразование каждого типа в тип object позволяет создавать универсальные классы, работающие с любыми типами.

# Преобразование в строковый тип

Все типы – потомки `object`, а, следовательно, обладают методом `ToString()`

Метод `ToString()` можно вызывать явно, но, если явный вызов не указан, то он будет вызываться неявно, всякий раз, когда требуется преобразование к строковому типу.

```
Console.WriteLine(24.5);
```

# Преобразование типов. Методы Parse и TryParse

Все примитивные типы имеют два метода, которые позволяют преобразовать строку к данному типу. Это методы **Parse()** и **TryParse()**.
Метод **Parse()** в качестве параметра принимает строку и возвращает объект текущего типа.

```
int a = int.Parse("10");
double b = double.Parse("23,56");
decimal c = decimal.Parse("12,45");
byte d = byte.Parse("4");
Console.WriteLine($"a={a}  b={b}  c={c}  d={d}");
```

# Преобразование типов. Методы Parse и TryParse

Метода TryParse() пытается преобразовать строку к типу и, если преобразование прошло успешно, то возвращает true.

```
int number;
Console.WriteLine("Введите строку:");
string input = Console.ReadLine();

bool result = int.TryParse(input, out number);
if (result == true)
     Console.WriteLine("Преобразование прошло успешно");
else
     Console.WriteLine("Преобразование завершилось неудачно");
```

# Convert

Класс **Convert** представляет еще один способ для преобразования значений. Для этого в нем определены следующие статические методы:

ToBoolean(value) ToByte(value)
ToChar(value) ToDateTime(value)
ToDecimal(value) ToDouble(value)
ToInt16(value) ToInt32(value)
ToInt64(value) ToSByte(value)
ToSingle(value) ToUInt16(value)
ToUInt32(value) ToUInt64(value)

# Convert

В качестве параметра в эти методы может передаваться значение различных примитивных типов, необязательно строки:

```
int n = Convert.ToInt32("23");
bool b = true;
double d = Convert.ToDouble(b);
Console.WriteLine($"n={n}  d={d}");
```

Если методу не удастся преобразовать значение к нужному типу, то он выбрасывает исключение FormatException.

# Введение в язык C#

## ПРОСТРАНСТВА ИМЕН

# Пространства имен

Пространство имен (namespace) в C# представляет собой некий контейнер для логического объединения именованных сущностей, таких как классы, интерфейсы, перечисления и т.д.

# Пространства имен

Пространство имен можно использовать для организации элементов кода и для создания глобально уникальных типов.

( https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/keywords/namespace )

# Пространства имен

Пространства имен используются:

- для упорядочения классов .NET
- для объявления собственных пространств имен и соответсвенно для контролироля областей имен классов и методов в более крупных проектах.

# Пространства имен

Упорядочение классов .NET

пространства имен позволяют логически группировать классы и другие сущности,

# Пространства имен

## Объявление собственных пространств имен

Позволяет использовать одни и те же имена для сущностей в разных пространствах имен

# Пространства имен

| | |
|---|---|
| System | Содержит базовые типы, позволяющие иметь дело с внутренними данными, математическими вычислениями, генерированием случайных чисел, переменными среды и сборкой мусора, а также ряд наиболее часто применяемых исключений и атрибутов |
| System.Collections<br>System.Collections.Generic | В этих пространствах имен содержится ряд контейнерных типов, а также несколько базовых типов и интерфейсов, которые позволяют создавать специальные коллекции |
| System.IO<br>System.IO.Compression<br>System.IO.Ports | В этих пространствах содержится много типов, предназначенных для работы с операциями файлового ввода-вывода, сжатия данных и манипулирования портами |
| System.Reflection<br>System.Reflection.Emit | В этих пространствах имен содержатся типы, которые поддерживают обнаружение типов во время выполнения, а также динамическое создание типов |

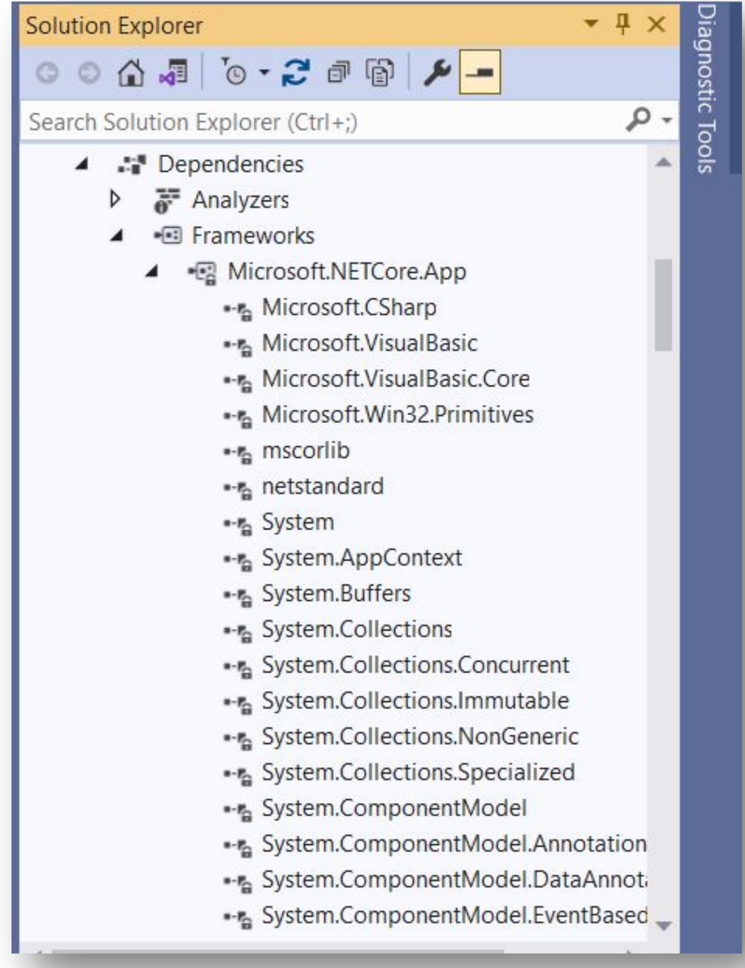
# Пространства имен

| | |
|---|---|
| System.Linq<br>System.Xml.Linq | В этих пространствах имен содержатся типы, применяемые при выполнении программирования с использованием API-интерфейса LINQ |
| System.Threading<br>System.Threading.Tasks | В этом пространстве имен содержатся многочисленные типы для построения многопоточных приложений, способных распределять рабочую нагрузку среди нескольких ЦП. |
| System.Xml | В этом ориентированном на XML пространстве имен содержатся многочисленные типы, которые можно применять для взаимодействия с XML-данными |

На каждом компьютере, на котором установлена среда CLR, есть кэш кода в масштабе всей машины, называемый Global Assembly Cache (GAC).
В глобальном кэше сборок хранятся сборки, специально предназначенные для совместного использования несколькими приложениями на компьютере.

В OS Windows GAC находится по пути

Windows\assembly


Solution Explorer
Diagnostic Tools
Search Solution Explorer (Ctrl+;)
Dependencies
Analyzers
Frameworks
Microsoft.NETCore.App
Microsoft.CSharp
Microsoft.VisualBasic
Microsoft.VisualBasic.Core
Microsoft.Win32.Primitives
mscorlib
netstandard
System
System.AppContext
System.Buffers
System.Collections
System.Collections.Concurrent
System.Collections.Immutable
System.Collections.NonGeneric
System.Collections.Specialized
System.ComponentModel
System.ComponentModel.Annotation
System.ComponentModel.DataAnnot
System.ComponentModel.EventBased

# Пространства имен

```
class Program
{
   static void Main()
   {
      string data = System.IO.File.ReadAllText("demo.txt");
   }
}
```

# Пространства имен

```
using System;
using System.IO;

. . .

    class Program
    {
        static void Main()
        {
            string data = File.ReadAllText("demo.txt");
        }
    }
}
```

# Пространства имен

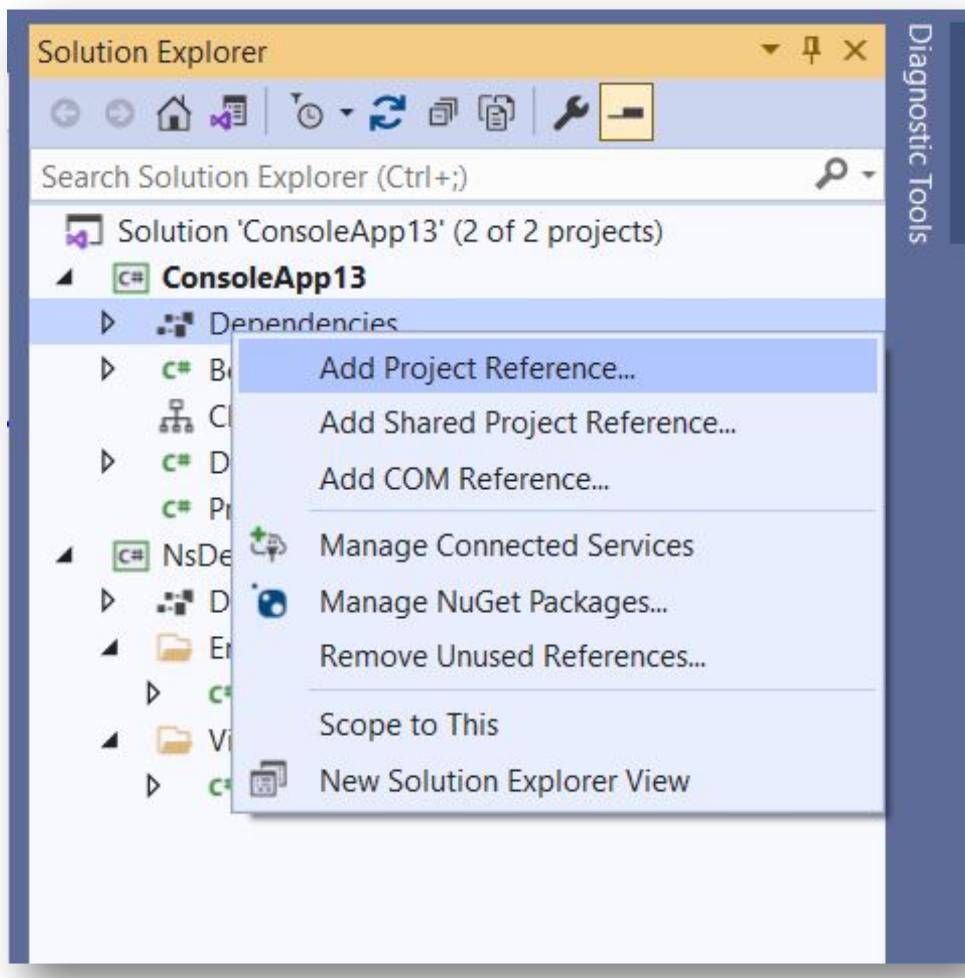
```
using System;

namespace NsDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

# Пространства имен

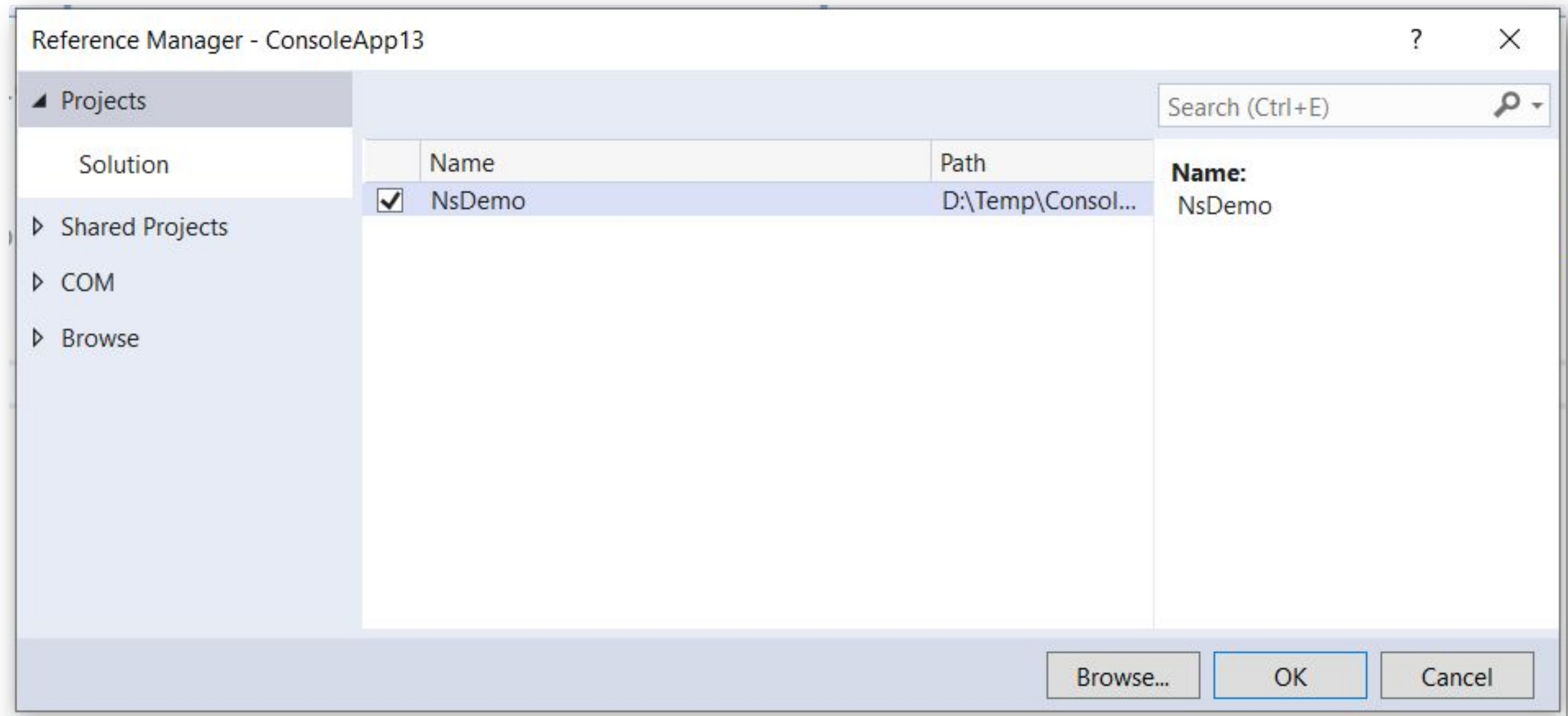
```
namespace NsDemo.Entities
{
    public class Person { }
}
```

```
namespace NsDemo.ViewModels
{
    public class Person { }
}
```

# Пространства имен



# Пространства имен



# Пространства имен

```
var person1 = new NsDemo.Entities.Person();

var person2 = new NsDemo.ViewModels.Person();
```

# Пространства имен (псевдоним / alias)

```
using ent = NsDemo.Entities;
using vm = NsDemo.ViewModels;

var person1 = new ent::Person();

var person2 = new vm::Person();
```

# Пространства имен (псевдоним **global**)

global относится к глобальному пространству имен, его можно использовать для решения проблем, связанных с переопределением типов.

# Пространства имен (псевдоним **global**)

```
class Demo
{
    class System
    { }

    System sys;
    public Demo()
    {
        sys = new System();
        global::System.Console.WriteLine("...");
    }
}
```

# Введение в язык C#

## ВЫРАЖЕНИЯ, ОПЕРАТОРЫ

Простейшими выражениями C# являются литералы (например, целые и реальные числа) и имена переменных. Их можно объединить в сложные выражения с помощью операторов.

# Основные виды операторов в C#

<u>Оператор-выражение</u>. Под выражением может пониматься вызов метода, присваивание, а также допустимые комбинации операндов и операций. Оператор-выражение завершается символом ;

<u>Операторы управления</u> ходом выполнения программы, такие как оператор условного перехода или операторы циклов.

<u>Блок операторов</u>. Блок – это набор операторов, обрамленных фигурными скобками – { и }. Блоки использует там, где синтаксис языка требует одного оператора.

<u>Операторы объявлений</u> пользовательских типов, элементов типов и локальных переменных и констант.

# Выражения и операции

| Прио-ритет | Категория | Выражение | Описание |
|---|---|---|---|
| 1. | Первичные | **x.m** | Доступ к элементу типа |
| | | **x(…)** | Вызов методов и делегатов |
| | | **x[…]** | Доступ к элемену массива и индексатора |
| | | **x++** | Постинкремент |
| | | **x--** | Постдекремент |
| | | **new T(…)** | Создание объекта или делегата |
| | | **new T[…]** | Создание массива |
| | | **typeof(T)** | Получение для типа **T** объекта **System.Type** |
| | | **checked(x)** | Вычисление в контролируемом контексте |
| | | **unchecked(x)** | Вычисление в неконтролируемом контексте |
| 2. | Унарные | **+x** | Идентичность |
| | | **-x** | Отрицание |
| | | **!x** | Логическое отрицание |
| | | **~x** | Битовое отрицание |
| | | **++x** | Пре-инкремент |
| | | **--x** | Пре-декремент |
| | | **(T)x** | Явное преобразование **x** к типу **T** |

# Выражения и операции

| | | | |
|---|---|---|---|
| 3. | Умножение | **x * y** | Умножение |
| | | **x / y** | Деление |
| | | **x % y** | Вычисление остатка |
| 4. | Сложение | **x + y** | Сложение, конкатенация строк |
| | | **x – y** | Вычитание |
| 5. | Сдвиг | **x << y** | Битовый сдвиг влево |
| | | **x >> y** | Битовый сдвиг вправо |
| 6. | Отношение и проверка типов | **x < y** | Меньше |
| | | **x > y** | Больше |
| | | **x <= y** | Меньше или равно |
| | | **x >= y** | Больше или равно |
| | | **x is T** | Возвращает **true**, если тип **x** это **T** |
| | | **x as T** | Возвращает **x**, приведенный к типу **T**, или **null** |

# Выражения и операции

| | | | |
|---|---|---|---|
| 7. | Равенство | **x == y** | Равно |
| | | **x != y** | Не равно |
| 8. | Логическое AND | **x & y** | Целочисленное битовое AND, логическое AND |
| 9. | Логическое XOR | **x ^ y** | Целочисленное битовое XOR, логическое XOR |
| 10. | Логическое OR | **x \| y** | Целочисленное битовое OR, логическое OR |
| 11. | Сокращенное AND | **x && y** | Вычисляется **y**, только если **x = true** |
| 12. | Сокращенное OR | **x \|\| y** | Вычисляется **y**, только если **x = false** |
| 13. | Условие | **x ? y : z** | Если **x = true**, вычисляется **y**, иначе **z** |
| 14. | Присваивание | **x = y** | Присваивание |
| | | **x op= y** | Составное присваивание, поддерживаются ***= /= %= += -= <<= >>= &= ^= \|=** |

# Преобразование типов в выражениях

Преобразование типов выполняется на основе правил продвижения по "типовой” лестнице.

Правило продвижения типов действует только при вычислении выражения.

# Преобразование типов для бинарных операций

- ЕСЛИ один операнд имеет тип decimal, ТО и второй "возводится в ранг", т.е. "в тип" decimal (но если второй операнд имеет тип float или double, результат будет ошибочным).
- ЕСЛИ один операнд имеет тип double, ТО и второй преобразуется в значение типа double.
- ЕСЛИ один операнд имеет тип float, ТО и второй преобразуется в значение типа float.
- ЕСЛИ один операнд имеет тип ulong, ТО и второй преобразуется в значение типа ulong (но если второй операнд имеет тип sbyte, short, int или long, результат будет ошибочным).
- ЕСЛИ один операнд имеет тип long, ТО и второй преобразуется в значение типа long.
- ЕСЛИ один операнд имеет тип uint, а второй имеет тип sbyte, short или int, ТО оба операнда преобразуются в значения типа long.
- ЕСЛИ один операнд имеет тип uint, ТО и второй преобразуется в значение типа uint.

# Преобразование типов для бинарных операций

```
int a = 10;
int b = 3;

Console.WriteLine(a/b);
Console.WriteLine((double)a/b);
```

# Введение в язык C#

## УПРАВЛЕНИЕ ПОТОКОМ ВЫПОЛНЕНИЯ ПРОГРАММЫ

# Операторы выбора

```
if (<условие>)
    <блок1>
[else
    <блок2>]


switch (<выражение>)
{
   case: … break;
   …


}
```

<выражение> должно иметь целый числовой тип, символьный или строковый тип.

# Операторы цикла

```
for ([<инициализатор>]; [<условие>]; [<итератор>]) <блок>

while (<условие>) <блок>

do
    <блок>
while (<условие>);
```

# Тернарный оператор

```
if (a > b)
        message = "a больше, чем b";
else
        message = "b больше, чем a";
```

```
message = a > b
                ? "a больше, чем b"
                : "b больше, чем a";
```

# Значение null

# Значение null

Одно из отличий ссылочных типов от типов значений состоит в том, что переменные ссылочных типов могут принимать значение null.

```
object o = null;
string s = null;
```

Если переменным ссылочного типа не присваивается значение, то им дается значение по умолчанию - значение null. Фактически оно говорит об отсутствии значения как такового.

# Nullable типы значений

```
double? pi = 3.14;
char? letter = 'a';

int m2 = 10;
int? m = m2;

bool? flag = null;

// Массив nullable типов значений:
int?[] arr = new int?[10];
```

# Nullable типы значений (проверка)

```
int? c = 7;
if (c != null)
{
    Console.WriteLine($"c is {c.Value}");
}
else
{
    Console.WriteLine("c does not have a value");
}
// Результат:
// c is 7
```

# Nullable типы значений (проверка)

```
int? b = 42;
if (b is int valueOfb)
{
    Console.WriteLine($"b is {valueOfb}");
}
else
{
    Console.WriteLine("a does not have a value");
}
// Результат:
// b is 42
```

# Nullable типы значений (проверка)

```
int? b = 10;
if (b.HasValue)
{
    Console.WriteLine($"b is {b.Value}");
}
else
{
    Console.WriteLine("b does not have a value");
}
// Результат:
// b is 10
```

# Оператор ??

Оператор ?? называется оператором **null-объединения**.

Он применяется для установки значений по умолчанию для типов, которые допускают значение null.

Оператор ?? возвращает левый операнд, если этот операнд не равен null. Иначе возвращается правый операнд.

При этом левый операнд должен принимать null.

```
object x = null;
object y = x ?? 100;  // равно 100, так как x равен null

object z = 200;
object t = z ?? 44; // равно 200, так как z не равен null
```

# Оператор ??

Но мы не можем написать следующим образом:

```
int x = 44;
int y = x ?? 100;
```

Здесь переменная `x` представляет значимый тип `int` и не может принимать значение `null`, поэтому в качестве левого операнда в операции `??` она использоваться не может.

# Оператор условного null

Иногда при работе с объектами, которые принимают значение `null`, мы можем столкнуться с ошибкой: мы пытаемся обратиться к объекту, а этот объект равен `null`.

# Оператор условного null

Объект User содержит ссылку на объект Phone, а объект Phone содержит ссылку на объект Company, поэтому теоретически мы можем получить из объекта User название компании:

```
class User
{
    ссылок: 0
    public Phone Phone { get; set; }
}

ссылка: 1
class Phone
{
    ссылок: 0
    public Company Company { get; set; }
}

ссылка: 1
class Company
{
    ссылок: 0
    public string Name { get; set; }
}
```

```
User user = new User();
Console.WriteLine(user.Phone.Company.Name);
```

В данном случае свойство Phone не определено, будет по умолчанию иметь значение null.

Поэтому мы столкнемся с исключением NullReferenceException.

# Оператор условного null

Чтобы избежать этой ошибки мы могли бы использовать условную конструкцию для проверки на null:

```
User user = new User();

if (user != null)
{
    if (user.Phone != null)
    {
        if (user.Phone.Company != null)
        {
            string companyName = user.Phone.Company.Name;
            Console.WriteLine(companyName);
        }
    }
}
```

# Оператор условного null

```
User user = new User();

if (user != null && user.Phone != null && user.Phone.Company != null)
{
    string companyName = user.Phone.Company.Name;
    Console.WriteLine(companyName);
}
```

`user.Phone!=null` и так далее.
Конструкция намного проще, но все равно получается довольно большой.

# Оператор условного null

```
User user = new User();

string companyName = user?.Phone?.Company?.Name;
```

Выражение `?.` и представляет оператор условного `null`.
Здесь последовательно проверяется равен ли объект `user` и вложенные объекты значению `null`.
Если же на каком-то этапе один из объектов окажется равным `null`, то `companyName` будет иметь значение по умолчанию, то есть `null`.

# Оператор условного null и оператор ??

```
User user = new User();
string companyName = user?.Phone?.Company?.Name ?? "не установлено";
Console.WriteLine(companyName);
```

# Введение в язык C#

## МАССИВЫ. ПЕРЕЧИСЛЕНИЯ. КОЛЛЕКЦИИ.

# Массив

Массив - набор элементов одного и того же типа, объединенных общим именем.

C#-массивы относятся к ссылочным типам данных, реализованы как объекты.

Имя массива является ссылкой на область кучи (динамической памяти), в которой последовательно размещается набор элементов определенного типа.

Выделение памяти под элементы происходит на этапе инициализации массива.

# Одномерные массивы

Одномерный массив – это фиксированное количество элементов одного и того же типа, объединенных общим именем, где каждый элемент имеет свой номер.

1. Объявляется ссылочная переменная на массив

2. Выделяется память под требуемое количество элементов базового типа, и ссылочной переменной присваивается адрес нулевого элемента в массиве.

# Объявление одномерного массива

| *Форма записи* | *Пояснения* |
|---|---|
| базовый_тип [] имя__массива;<br>*Например*:<br>int [] a; | Описана ссылка на одномерный массив, которая в дальнейшем может быть использована:<br>1) для адресации на уже существующий массив;<br>2) передачи массива в метод в качестве параметра<br>3) отсроченного выделения памяти под элементы массива. |
| базовый_тип [] имя__массива = new базовый_тип [размер];<br>*Например*:<br>int []a=new int [10]; | Объявлен одномерный массив заданного типа и выделена память под одномерный массив указанной размерности. Адрес данной области памяти записан в ссылочную переменную. Элементы массива равны нулю.<br>***Замечание***. В C# элементам массива присваиваются начальные значения по умолчанию в зависимости от базового типа.<br>Для арифметических типов – нули,<br>для ссылочных типов – null, для символов - пробел. |
| базовый_тип [] имя__массива={список инициализации};<br>*Например*:<br>int []a={0, 1, 2, 3}; | Выделена память под одномерный массив, размерность которого соответствует количеству элементов в списке инициализации. Адрес этой области памяти записан в ссылочную переменную. Значение элементов массива соответствует списку инициализации. |

```
int[] myArray = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
int i;
for (i = 0; i < 10; ++i)
    Console.WriteLine(myArray[i]);
```

```
int[] myArray = new int[10];
for (int i = 0; i < 10; i++)
    myArray[i] = i * i;
for (int i = 0; i < 10; i++)
    Console.WriteLine(myArray[i]);
```

# Существующей ссылке на одномерный массив присваивается ссылка на новый массив

```
int[] myArray = { 0, 1, 2, 3, 4, 5 };
int i;
for (i = 0; i < 10; i++)
    Console.Write(" " + myArray[i]);
Console.WriteLine("\nНовый массив: ");
```

1.переменная myArray ссылалась на 6-ти элементный массив.
2.переменной myArray была присвоена ссылка на новый 10-элементный массив, в результате чего исходный массив оказался неиспользуемым, т.к. на него теперь не ссылается ни один объект.
3.он автоматически будет удален сборщиком мусора.

```
myArray = new int[] { 99, 10, 100, 18, 78, 23, 163, 9, 87, 49 };
for (i = 0; i < 10; i++)
    Console.Write(" " + myArray[i]);
```

# Массивы и исключения

```
int[] myArray = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
int i;
try
{
    for (i = 0; i <= 10; i++) Console.WriteLine(myArray[i]);
}
catch (IndexOutOfRangeException)
{
    Console.WriteLine("Exception: Выход за границу диапазона");
}
```

```
0
1
2
3
4
5
6
7
8
9
Exception: Выход за границу диапазона
```

# Массив как параметр

Так как имя массива фактически является ссылкой, то он передается в метод по ссылке

Все изменения элементов массива, являющегося формальным параметром, отразятся на элементах соответствующего массива, являющимся фактическим параметром.

```
class Program
   {
      static void Print(int n, int[] a)  //n – размерность массива, а – ссылка на массив
       {
          for (int i = 0; i < n; i++) Console.Write("{0} ", a[i]);
          Console.WriteLine();          }

       static void Change(int n, int[] a)
       {
          for (int i = 0; i < n; i++)
             if (a[i] > 0) a[i] = 0;    // изменяются элементы массива
       }

       static void Main()
       {
          int[] myArray = { 0, -1, -2, 3, 4, 5, -6, -7, 8, -9 };
          Print(10, myArray);
          Change(10, myArray);
          Print(10, myArray);          }
    }
```

```
0 -1 -2 3 4 5 -6 -7 8 -9
0 -1 -2 0 0 0 -6 -7 0 -9
Для продолжения нажмите любую клавишу
```

# Массив как объект

| *Элемент* | *Вид* | *Описание* |
|---|---|---|
| Length | свойство | Количество элементов массива (по всем размерностям) |
| BinarySearch | статический метод | Двоичный поиск в отсортированном массиве |
| Clear | статический метод | Присваивание элементам массива значений по умолчанию |
| Copy | статический метод | Копирование заданного диапазона элементов одного массива в другой |
| CopyTo | экземплярный метод | Копирование всех элементов текущего одномерного массива в другой массив |
| GetValue | экземплярный метод | Получение значения элемента массива |
| IndexOf | статический метод | Поиск первого вхождения элемента в одномерный массив |
| LastIndexOf | статический метод | Поиск последнего вхождения элемента в одномерный массив |
| Reverse | статический метод | Изменение порядка следования элементов на обратный |
| SetValue | экземплярный метод | Установка значения элемента массива |
| Sort | статический метод | Упорядочивание элементов одномерного массива |

Вызов статических методов происходит через обращение к имени класса

Например:

/*Обращение к статическому методу Sort класса Array и передача данному методу в качестве параметра объект myArray - экземпляр класса Array*/

***Array.Sort(myArray)***

Обращение к свойству или вызов экземплярного метода производится через обращение к экземпляру класса

Например:

***myArray.Length***

или

***myArray.GetValue(i)***

```
try
{
    int[] MyArray;
    Console.Write("Введите размерность массива: ");
    int n = int.Parse(Console.ReadLine());
    MyArray = new int[n];
    for (int i = 0; i < MyArray.Length; ++i)
    {
        Console.Write("a[{0}]=", i);
        MyArray[i] = int.Parse(Console.ReadLine());
    }
    PrintArray("исходный массив:", MyArray);
    Array.Sort(MyArray);
    PrintArray("массив отсортирован по возрастанию", MyArray);
    Array.Reverse(MyArray);
    PrintArray("массив отсортирован по убыванию", MyArray);
}
catch (FormatException)
{
    Console.WriteLine("неверный формат ввода данных");
}
catch (OverflowException)
{ Console.WriteLine("переполнение"); }
catch (OutOfMemoryException)
{
    Console.WriteLine("недостаточно памяти для создания нового объекта");
}

static void PrintArray(string a, int[] mas)
{
    Console.WriteLine(a);
    for (int i = 0; i < mas.Length; i++) Console.Write("{0} ", mas[i]);
    Console.WriteLine();
}
```

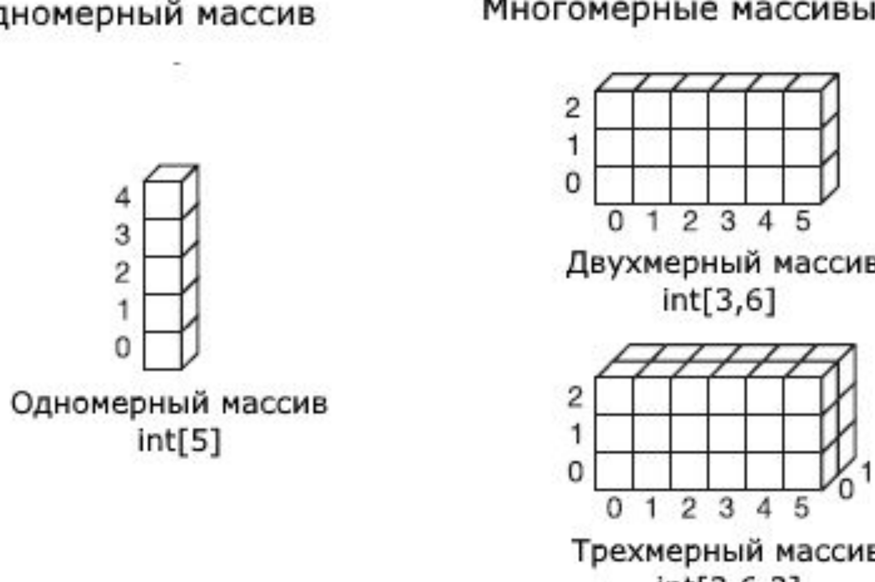
```
Введите размерность массива: i
неверный формат ввода данных
```

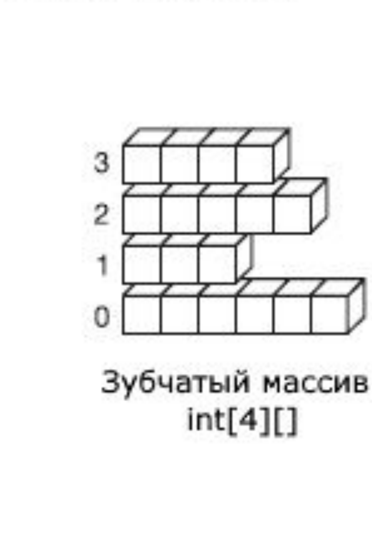
```
Введите размерность массива: 1000000000000000
переполнение
```

```
Введите размерность массива: 7
a[0]=1
a[1]=5
a[2]=66
a[3]=-8
a[4]=0
a[5]=-88
a[6]=9
исходный массив:
1 5 66 -8 0 -88 9
массив отсортирован по возрастанию
-88 -8 0 1 5 9 66
массив отсортирован по убыванию
66 9 5 1 0 -8 -88
Для продолжения нажмите любую клавишу . . .
```

# Многомерные массивы

С C# поддерживается 2 вида многомерных массивов: прямоугольные и зубчатые (рваные).



# Многомерные массивы

**тип [,] имя__массива;**

**тип [,] имя__массива = new тип [размер1, размер2];**

**тип [,] имя__массива={{элементы 1-ой строки}, … , {элементы n-ой строки}};**

**тип [,] имя__массива= new тип [,]{{элементы 1-ой строки}, … ,{элементы n-ой строки}};**

Например:

***int [,] a;***

***int [,] a= new int [3, 4];***

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

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

# Рваные массивы

*тип[ ] [ ] имя = new тип[размер][ ];*

| jagged[0][0] | jagged[0][1] | jagged[0][2] | jagged[0][3] | |
|---|---|---|---|---|
| jagged[1][0] | jagged[1][1] | jagged[1][2] | | |
| jagged[2][0] | jagged[2][1] | jagged[2][2] | jagged[2][3] | jagged[2][4] |

```
int [ ] [ ] jagged = new int [ 3 ] [ ] ;
jagged [0] = new int [ 4 ] ;
jagged [1] = new int [ 3 ] ;
jagged [2] = new int [ 5 ] ;
```

Так как каждая строка ступенчатого массива фактически является одномерным массивом, то с каждой строкой можно работать как с экземпляром класса Array.
Это является преимуществом ступенчатых массивов перед двумерными массивами.

# Оператор foreach

Применяется для перебора элементов в специальным образом организованной группе данных, в том числе и в массиве.
Удобство заключается в том, что не требуется определять количество элементов в группе и выполнять перебор по индексу –просто указываем на необходимость перебрать все элементы группы.

***foreach (<тип> <имя> in <группа>) <тело цикла>***

где *имя* определяет локальную по отношению к циклу переменную, которая будет по очереди принимать все значения из указанной *группы*, а *тип* соответствует базовому типу элементов *группы*.

Ограничение: с его помощью можно только просматривать значения элементов в группе данных, но нельзя их изменять.

# Оператор foreach

**1) для работы с одномерными массивами:**

```
 static void PrintArray(string a, int [] mas)
        {
            Console.WriteLine(a);
            foreach (int x in mas)Console.Write("{0} ", x);
            Console.WriteLine();        }
```

**2) для работы с двумерными массивами:**

```
static int Sum (int [,] mas)
        {
            int s=0;
            foreach (int x in mas) s += x;
            return s;        }
```

**3) для работы со ступенчатыми массивами:**

```
static void PrintArray3(string a, int[][] mas)
            {
                Console.WriteLine(a);
                for (int i = 0; i < mas.Length; i++)
                {
                    foreach (int x in mas[i]) Console.Write("{0} ", x);
                    Console.WriteLine();                }            }
```

# Массивы и коллекции

Для хранения набора элементов одного типа используется массив. Но у массива есть свои недостатки.

После **создания массива его размеры должны оставаться неизменными**, поэтому *дописывать новые элементы в конец уже существующего массива нельзя* — требуется создавать новый массив.

При удалении элементов остаются пустые места.

Ключ в массиве – только индекс.

# Массивы и коллекции

**Коллекции** — это объекты, в которых могут содержаться наборы других объектов и которые обладают функциональными возможностями для организации доступа к ним.

# Коллекции данных

## СПИСКИ

# Списки

Для динамических списков в .NET Framework предусмотрен обобщенный класс **List<T>**.

Этот класс реализует интерфейсы IList, ICollection, IEnumerable, IList<T>, ICollection<T> и IEnumerable<T>.

# Списки

Конструктор по умолчанию List<T> создает пустой список. Как только элементы начинают добавляться в список, его емкость увеличивается до 4 элементов.

При добавлении пятого элемента размер списка изменяется так, чтобы уместить 8 элементов. Если же и этого недостаточно, список вновь расширяется, на этот раз до 16 элементов. При каждом расширении емкость списка удваивается.

# Списки

```
var intList = new List<int>();
```

# Списки

**Присваивать** значения коллекциям можно с помощью инициализаторов коллекций. Синтаксис инициализаторов коллекций подобен инициализаторам массивов.

```
var intList = new List<int>() { 1, 2 };

var stringList =
        new List<string>() { "one", "two" };
```

# Списки

**Добавлять** элементы в список можно методом Add()
Обобщенный параметрический тип определяет тип первого параметра метода Add ().

```
var intList = new List<int>();
intList.Add(22);
intList.Add(33);
var stringList = new List<string>();
stringList.Add("one");
stringList.Add("two");
```

# Списки

Метод **AddRange()** класса List<T>, можно добавить множество элементов в коллекцию за один прием. Метод AddRange () принимает объект типа **IEnumerable<T>**, так что допускается передавать массив.

# Списки

Для вставки элементов в определенную позицию коллекции служит метод **Insert()**:

```
var intList =
            new List<int> { 22,34,52,37,35};
intList.Insert(1, 33);
```

Если указывается индекс, превышающий количество элементов в коллекции, генерируется исключение типа **ArgumentOutOfRangeException**.

# Списки

Метод **InsertRange()** предоставляет возможность вставки множества элементов, подобно тому, как это делает метод AddRange()

# Списки

Все классы, реализующие интерфейсы IList и IList<T>, предоставляют индексатор, так что к элементам можно обращаться с использованием индексатора, передавая ему номер элемента. Первый элемент доступен по индексу 0.

```
var intList =
     new List<int> { 22,34,52,37,35};
intList.Insert(1, 33);
var item = intList[1];
```

# Списки

Существуют различные способы поиска элементов в коллекции. Можно получить индекс найденного элемента или сам найденный элемент.

- IndexOf(),
- LastIndexOf(),
- FindIndex(),
- FindLastIndex(),
- Find() и FindLast ().

Для проверки существования элемента класс List<T> предлагает метод Exists().

# Списки

Класс List<T> позволяет сортировать свои элементы с помощью метода Sort(), в котором реализован алгоритм быстрой сортировки.

# Списки

Для использования доступно несколько перегрузок метода Sort().

Аргументы, которые могут ему передаваться — это делегат Comparison<T>, обобщенный интерфейс IComparer<T> и диапазон вместе с обобщенным интерфейсом IComparer<T>

# Списки

```
var intList = new List<int> { 22,34,52,37,1};
intList.Sort();
foreach(var item in intList)
    Console.WriteLine(item);
```

# Списки

```
public class Book
    {
        #region Свойства
        public int Id { get; set; }
        public string Name { get; set; }
        public int Pages { get; set; }
        #endregion
    }
```

# Списки (сортировка, вариант 1)

```
public class Book : IComparable
    {
        #region Свойства
        public int Id { get; set; }
        public string Name { get; set; }
        public int Pages { get; set; }
        #endregion
    }
```

# Списки (сортировка, вариант 2)

```
class BookComparer : Comparer<Book>
    {
        public override int Compare(Book x, Book y)
        {
            return x.Pages.CompareTo(y.Pages);
        }
    }



var books = new List<Book>();
books.Sort(new BookComparer());
```

# Списки (сортировка, вариант 3)

```
var books = new List<Book>();
books.Sort((b1,b2)=>b1.Pages.CompareTo(b2));
```

# Коллекции данных

## ОЧЕРЕДЬ

# Очередь (Queue)

Очередь (queue) — это коллекция, в которой элементы обрабатываются по схеме "первый вошел, первый вышел" (first in, first out — FIFO). Элемент, вставленный в очередь первым, первым же и читается.

# Очередь (Queue)

Очередь реализуется с помощью класса **Queue<T>** из пространства имен **System.Collections.Generic**.

# Очередь (Queue)

Внутри класс Queue<T> использует массив типа T, который реализует интерфейсы **IEnumerable<T>** и **ICollection**, но не ICollection<T>.

Интерфейс ICollection<T> не реализован, поскольку он определяет методы Add () и Remove (), которые не должны быть доступны для очереди.

# Очередь (Queue)

Очередь позволяет добавлять элементы, при этом элемент помещается в конец очереди (методом **Enqueue()**), а также получать элементы из головы очереди (методом **Dequeue()**)

# Коллекции данных

## СТЕК

# Стек (Stack)

Стек (stack) — это контейнер, работающий по принципу "последний вошел, первый вышел" (last in, first out — LIFO).

# Стек (Stack)

Класс Stack<T> предоставляет следующие методы:

- Push () добавляет элемент,
- Pop () - получает элемент, добавленный последним.

Подобно классу Queue<T>, класс Stack<T> реализует интерфейсы IEnumerable<T> и ICollection.

# Коллекции данных

## СВЯЗАННЫЙ СПИСОК

# Связанный список (LinkedList)

Класс **LinkedList<T>** представляет собой двухсвязный список, в котором каждый элемент ссылается на следующий и предыдущий.

Класс **LinkedList<T>** наследуется от интерфейсов ICollection<T>, Icollection, IEnumerable<T>, IEnumerable

# Связанный список (LinkedList)

Преимущество связного списка проявляется в том, что операция вставки элемента в середину выполняется очень быстро. При этом только ссылки Next (следующий) предыдущего элемента и Previous (предыдущий) следующего элемента должны быть изменены так, чтобы указывать на вставляемый элемент.

В классе List<T> при вставке нового элемента все последующие должны быть сдвинуты.

# Связанный список (LinkedList)

Все элементы связных списков доступны лишь друг за другом. Поэтому для нахождения элемента, находящегося в середине или конце списка, требуется довольно много времени.

# Связанный список (LinkedList)

LinkedList<T> содержит элементы типа **LinkedListNode<T>**.

Класс LinkedListNode<T> определяет свойства List, Next, Previous и Value.

# Связанный список (LinkedList)

- Свойство **List** возвращает объект LinkedList<T>, ассоциированный с узлом.
- Свойства **Next** и **Previous** предназначены для итераций по списку и для доступа к следующему и предыдущему элементам.
- Свойство **Value** типа T возвращает элемент, ассоциированный с узлом.

# Коллекции данных

## СОРТИРОВАННЫЙ СПИСОК

# Сортированный список (SortedList)

Класс **SortedList<TKey, TValue>** сортирует элементы на основе значения ключа.

# Сортированный список (SortedList)

Конструктор по умолчанию создает пустой список.

Применяя перегруженные конструкторы, можно указать **емкость списка**, а также передать объект, который реализует интерфейс **IComparer<TKey>**, используемый для сортировки элементов в списке.

# Сортированный список (SortedList)

С помощью оператора foreach можно выполнить итерацию по списку.

Элементы, возвращенные перечислителем, имеют тип **KeyValuePair<TKey, TValue>**, который содержит как ключ, так и значение. Ключ доступен через свойство Key, а значение - через свойство Value.

# Сортированный список (SortedList)

Свойства **Keys** и **Values** списка SortedList позволяют обращаться сразу ко всем ключам и значениям.

Свойство Values возвращает **IList<TValue>**
Свойство Keys — **IList<TKey>**

Эти свойства можно использовать вместе с foreach.

# Коллекции данных

## СЛОВАРИ

# Словари (Dictionary)

Словарь (dictionary) представляет собой сложную структуру данных, позволяющую обеспечить доступ к элементам по ключу.

Главное свойство словарей — быстрый поиск на основе ключей. Можно также свободно добавлять и удалять элементы, подобно тому, как это делается в List<T>, но без накладных расходов производительности, связанных с необходимостью смещения последующих элементов в памяти.

# Словари (Dictionary)

Главный класс, который можно использовать — это **Dictionary<TKey, TValue>**

# Словари (Dictionary)

Класс **SortedDictionary<TKey, TValue>** представляет дерево бинарного поиска, в котором все элементы отсортированы на основе ключа.

Тип ключа должен реализовать интерфейс **IComparable<TKey>**.

Если тип ключа не сортируемый, компаратор можно также создать, реализовав **IComparer<TKey>** и указав его в качестве аргумента конструктора сортированного словаря.

# Словари (Dictionary)

SortedList(TKey,TValue) использует меньше памяти, чем SortedDictionary (TKey, TValue).

SortedDictionary(TKey,TValue) имеет более быстрые операции вставки и удаления для несортированных данных.

Если список заполняется сразу из отсортированных данных, SortedList (TKey, TValue) работает быстрее, чем SortedDictionary (TKey, TValue).

# Коллекции данных

## МНОЖЕСТВА

# Множества

Коллекция, содержащая **только отличающиеся** элементы, называется ***множеством (set)***. В составе .NET имеются два множества — **HashSet<T>** и **SortedSet<T>**.

# Множества

Класс HashSet<T> содержит неупорядоченный список различающихся элементов, а в SortedSet<T> элементы упорядочены.

# Множества

Метод **Add** добавляет объектв во множество. Метод возвращает:

**true** – если объект добавлен в коллекцию

**false** – если тако объект оже есть в коллекции

# Множества

Множества также предоставляют методы для создания объединения нескольких множеств, пересечения множеств и определения, является ли одно множество надмножеством или подмножеством другого.

# Введение в язык C#

## КОРТЕЖИ

# Кортежи (с версии C# 7.0)

Кортежи предоставляют удобный способ для работы с набором значений.
Кортеж представляет набор значений, заключенных в круглые скобки:

```
var tuple1 = (2, 11);
(int, string) tuple2 = (2, "Hello");

Console.WriteLine(tuple1.Item1);
Console.WriteLine(tuple2.Item2);
```

# Кортежи (с версии C# 7.0)

```
var person = (Age:2, Name:"Bob");
Console.WriteLine($"{person.Name} -{person.Age} лет");


(string Name, int Age) person = ("Bob", 22);
Console.WriteLine($"{person.Name} -{person.Age} лет");



var (Name, Age) = ("Bob", 22);
Console.WriteLine($"{Name} -  {Age} лет");
```