

**1. Типы данных в языке C#. Разновидности типов данных. Тип данных Object. Упаковка и распаковка. Тип данных Struct.**

Типы значений	Ссылочные типы
<p>Простые типы:</p> <ol style="list-style-type: none"> <li>1. Целочисленный со знаком: sbyte, short, int, long</li> <li>2. Целочисленный без знака: byte, ushort, uint, ulong</li> <li>3. Символы Юникода: char</li> <li>4. Бинарный оператор IEEE с плавающей запятой: float, double</li> <li>5. Десятичное значение с повышенной точностью и плавающей запятой: decimal</li> <li>6. Логическое значение: bool</li> </ol>	<p>Типы классов:</p> <ol style="list-style-type: none"> <li>1. Исходный базовый класс для всех типов: object</li> <li>2. Строки Юникода: string</li> <li>3. Пользовательские типы в формате class C {...}</li> </ol>
	<p>Типы интерфейсов:</p> <ol style="list-style-type: none"> <li>1. Пользовательские типы в формате interface I {...}</li> </ol>
<p>Типы перечислений:</p> <ol style="list-style-type: none"> <li>1. Пользовательские типы в формате enum E {...}</li> </ol>	<p>Типы массивов:</p> <ol style="list-style-type: none"> <li>1. Одно- и многомерные, например, int[] и int[,]</li> </ol>
<p>Типы структур:</p> <ol style="list-style-type: none"> <li>1. Пользовательские типы в формате struct S {...}</li> </ol>	<p>Типы делегатов:</p> <ol style="list-style-type: none"> <li>1. Пользовательские типы в формате delegate int D(...)</li> </ol>
<p>Типы значений, допускающие значение null</p>	<p>Типы кортежей:</p> <ol style="list-style-type: none"> <li>1. Пользовательские типы в формате <code>var unnamed = ("one", "two");</code></li> </ol>

Переменные типа-значения содержат значения, которые сохраняются в области, известной как стек.

Переменные ссылочного типа хранят ссылки на данные, которые сохраняются в области называемой кучей.

**Куча** – хранилище памяти, расположенное в ОЗУ, которое допускает динамическое выделение памяти

*Значение ссылочного типа хранят не сами данные, а ссылку на них, то есть адрес по которому расположены данные, сами данные хранятся в куче, при этом объект ссылочного типа может одновременно именоваться несколькими ссылками.*

## Упаковка и Распаковка

```
int i = 123;  
object o = i; // Упаковка i в o  
o = 456;  
int j = (int)o; // Распаковка o в j
```

- Когда любой значимый тип присваивается к ссылочному типу данных, значение перемещается из области стека в кучу. Эта операция называется упаковкой (упаковка может быть неявной как показано в примере и явной).

*Например: в тип object или в другой любой тип интерфейса, реализующий этот тип.*

*Например: есть структура mystruct и интерфейс который реализует эта структура. Значит мы можем сделать упаковку. Создать переменную типа интерфейс и упаковать в эту переменную структуру.*

- Когда любой ссылочный тип присваивается к значимому типу данных, значение перемещается из области кучи в стек. Это называется распаковкой (распаковка неявная всегда).

## Упаковка

Упаковка используется для хранения типа значения в куче со сборщиком мусора. При упаковке тип значения в куче выделяется экземпляр объекта и выполняется копирование в новый объект.

## Распаковка

Распаковка является явным преобразованием из типа object в тип значения. При распаковке извлекается тип значения из объекта. Операция распаковки состоит из:

1. проверки экземпляра объекта на то, что он является упаковочным значением заданного типа значения;
2. копирование значение из экземпляра в переменную типа значения.

В С# предусмотрен специальный класс `object`, который неявно считается базовым классом для всех остальных классов и типов, включая и типы значений. Иными словами, все остальные типы являются производными от `object`. Это, в частности, означает, что переменная ссылочного типа `object` может ссылаться на объект любого другого типа. Кроме того, переменная типа `object` может ссылаться на любой массив, поскольку в С# массивы реализуются как объекты.

Практическое значение этого в том, что помимо методов и свойств, которые вы определяете, также появляется доступ к множеству общедоступных и защищенных методов-членов, которые определены в классе `object`. Эти методы присутствуют во всех определяемых классах.

Тип данных `Object` может указывать на данные любого типа данных, включая любой экземпляр объекта, распознаваемый приложением. Используйте `object`, если во время компиляции неизвестно, на какой тип данных может указывать переменная.

Все базовые типы неявно наследуются от `Object` и перенимают все базовые методы от `object`. Создаваемые пользователем типы наследуются от `object`.

### Класс `object` как универсальный тип данных

Если `object` является базовым классом для всех остальных типов и упаковка значений простых типов происходит автоматически, то класс `object` можно вполне использовать в качестве "универсального" типа данных. Для примера рассмотрим программу, в которой сначала создается массив типа `object`, элементам которого затем присваиваются значения различных типов данных:

```
namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            var myOS = Environment.OSVersion;
            object[] myArr = { "Строка", 120, 0.345m, 2.34f, myOS, 'Z' };

            foreach (object obj in myArr)
                Console.WriteLine("Элемент \"{0}\" его тип - {1}",obj,obj.GetType());

            Console.ReadLine();
        }
    }
}
```

Как показывает данный пример, по ссылке на объект класса `object` можно обращаться к данным любого типа, поскольку в переменной ссылочного типа `object` допускается хранить ссылку на данные всех остальных типов. Следовательно, в массиве типа `object` из рассматриваемого здесь

примера можно сохранить данные практически любого типа. В развитие этой идеи можно было бы, например, без особого труда создать класс стека со ссылками на объекты класса `object`. Это позволило бы хранить в стеке данные любого типа.

При использовании `var` такое будет невозможно.

---

**Struct** — тип, создаваемый пользователем на основе базовых типов или других типов, создаваемых пользователем.

Отличие структуры от класса:

- Структуры принадлежат к типам значений, классы принадлежат к ссылочным типам;
- При присваивании структур создается полная копия ее объекта, то есть выделяется память для дополнительной структуры. При присваивании классов ссылка на один класс присваивается ссылке на другой класс, и, как результат, обе ссылки указывают на один и тот же объект;
- Все структуры неявно наследуются от класса `System.ValueType`, который унаследован от `object`.
- Структуры не поддерживают наследование. Структура не может наследовать класс или другую структуру. Однако, структура может содержать переменную структурного типа;
- В отличие от классов, в структурах нельзя объявлять конструктор по умолчанию;
- В структуре и в классе доступ по умолчанию `private`;
- В отличие от классов, в структурах члены данных не могут быть объявлены с ключевым словом `protected`, поскольку структуры не поддерживают наследование;
- В отличие от классов, структуры не поддерживают деструкторов;
- Структуры не могут объявляться с ключевыми словами `abstract` и `virtual`;
- Структура может наследоваться только от интерфейса;
- В отличие от класса нельзя инициализировать поля структуры напрямую при их объявлении.

Объявление объекта структуры `User tom;` (Необязательно вызывать конструктор)

## 2. Типы данных `Nullable`. Синтаксис объявления. Принципы работы с переменными `Nullable`.

Значение `NULL` по умолчанию могут принимать только объекты ссылочных типов. Однако в различных ситуациях бывает удобно, чтобы объекты числовых типов данных имели значение `NULL`, то есть были бы не определены. Стандартный пример - работа с базой данных, которая может содержать значения `NULL`. И мы можем заранее не знать, что мы получим из

базы данных - какое-то определенное значение или же NULL. Для этого надо использовать знак вопроса (?) после типа значений.

Запись (?) является упрощенной формой использования структуры `System.Nullable<T>`.

```
1 int? x = null;
2 Console.WriteLine(x.Value); // Ошибка
3 State? state = null;
4 Console.WriteLine(state.Value.Name); // ошибка
```

В этом случае необходимо выполнять проверку на наличие значения с помощью еще одного свойства `HasValue`:

```
1 int? x = null;
2 if(x.HasValue)
3     Console.WriteLine(x.Value);
4 else
5     Console.WriteLine("x is equal to null");
```

Преобразование типов `Nullable`:

- явное преобразование от `T?` к `T`

```
1 int? x1 = null;
2 if(x1.HasValue)
3 {
4     int x2 = (int)x1;
5     Console.WriteLine(x2);
6 }
```

- неявное преобразование от `T` к `T?`

```
1 int x1 = 4;
2 int? x2 = x1;
3 Console.WriteLine(x2);
```

- неявные расширяющие преобразования от `V` к `T?`

```
1 int x1 = 4;
2 long? x2 = x1;
3 Console.WriteLine(x2);
```

### 3. Виды массивов в языке C#. Объявление и инициализация массивов.

#### Основные методы и свойства по работе с массивами.

Виды массивов: одномерный, многомерный, массив массивов.

тип\_переменной[] название\_массива;

`int[] nums = new int[4];` - объявление одномерного массива

`int[,] a2 = new int[10, 5];` - объявление двумерного массива

или

`int[,] nums3 = new int[2, 3] { { 0, 1, 2 }, { 3, 4, 5 } };` (инициализация массива)

`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

Все массивы в C# построены на основе класса **Array** из пространства имен **System**. Этот класс определяет ряд свойств и методов, которые мы можем использовать при работе с массивами. Основные свойства и методы:

- Свойство **Length** возвращает длину массива;
- Свойство **Rank** возвращает размерность массива;
- static int **BinarySearch()** – метод, который осуществляет бинарный поиск в упорядоченном массиве (когда массив делим по полам и отбрасываем одну из частей). Результат – это индекс соответствующего условию поиска элемента или отрицательное число, если нужный элемент в массиве отсутствует;
- Статический метод **Clear()** очищает массив, устанавливая для всех его элементов значение по умолчанию;
- Статический метод **Copy()** копирует часть одного массива в другой массив;
- Статический метод **Exists()** проверяет, содержит ли массив определенный элемент;
- Статический метод **Find()** находит элемент, который удовлетворяет определенному условию;
- Статический метод **FindAll()** находит все элементы, которые удовлетворяют определенному условию;
- Статический метод **IndexOf()** возвращает индекс элемента;
- Статический метод **Resize()** изменяет размер одномерного массива;
- Статический метод **Reverse()** располагает элементы массива в обратном порядке;
- Статический метод **Sort()** сортирует элементы одномерного массива  
метод сортирует элементы массива, по умолчанию осуществляется в порядке возрастания значений. Строки сортируются лексикографически. Используя необязательные параметры можно изменить правила упорядочивания;
- static int **LastIndexOf()** – метод возвращает последний индекс элемента значение которого совпадает с образцом.

#### 4. Типы перечислений. Объявление и использование.

**Тип перечисления** — это тип значения, определенный набором именованных констант применяемого целочисленного типа.

Оператор enum. Далее идет название перечисления, после которого указывается тип перечисления - он обязательно должен представлять целочисленный тип (byte, int, short, long, ushort, int, uint, long и ulong.). Этот тип называется **базовым**. Затем идет список элементов перечисления через запятую.

Синтаксис объявления перечисления:

```
enum имя_перечисления : тип
```

```
{  
    переменная_1,  
    переменная_2,  
    переменная_3  
}
```

Связное значение перечисление будет задаваться либо явно, либо неявно. Если задано неявно, то первый элемент перечисления автоматически принимает значение 0, последующие элементы увеличиваются на 1.

Если перечислению явно не указать тип ( : тип), то по умолчанию будет int.

```
enum Season
```

```
{  
    Spring,  
    Summer,  
    Autumn,  
    Winter  
}
```

Можно также присваивать значения:

```
enum Season
```

```
{  
    Spring=1,  
    Summer=6,  
    Autumn, //здесь 7 автоматически  
    Winter  
}
```

Зачастую переменная перечисления выступает в качестве хранилища состояния, в зависимости от которого производятся некоторые действия.

```
enum Season: byte
```

```
{  
    Spring,  
    Summer,  
    Autumn,  
    Winter  
}
```

**Перечисление необязательно определять внутри класса, можно и вне класса, но в пределах пространства имен.**

## 5. Кортежи в языке C#. Типы коллекций в языке C#. Тип Dictionary<TKey, TValue>. Тип List<T>.

**Кортеж** - это тип данных, являющийся аналогом записи или класса.

Кортежи – это типы, которые определяются с помощью упрощенного синтаксиса.

Кортеж еще один тип данных который может хранить набор значений.

**Объявление картежа:**

```
(string, int, bool) value = ("a", 5, true);
```

или

```
var value = ("a", 5, true);
```

По умолчанию компилятор назначает каждому свойству имя:

ItemX, где X – представляет позицию свойства в кортеже.

Кортежи используют тип ValueTuple. Данный тип создает разные структуры на основе количества свойств для кортежа.

Как запись или класс, кортеж представляет собой набор элементов разных типов, но описывается существенно проще. Кортеж представляет набор значений, заключенных в круглые скобки:

```
var tuple = (5, 10);
```

В данном случае определен кортеж tuple, который имеет два значения: 5 и 10.

В дальнейшем мы можем обращаться к каждому из этих значений через поля с названиями Item[порядковый\_номер\_поля\_в\_кортеже].

Например:

```
static void Main(string[] args)
{
    var tuple = (5, 10);
    Console.WriteLine(tuple.Item1); // 5
    Console.WriteLine(tuple.Item2); // 10
    tuple.Item1 += 26;
    Console.WriteLine(tuple.Item1); // 31
    Console.Read();
}
```

В данном случае тип определяется неявно. Но мы также можем явным образом указать для переменной кортежа тип:

```
(int, int) tuple = (5, 10);
```

Так как кортеж содержит два числа, то в определении типа нам надо указать два числовых типа. Или другой пример определения кортежа:

```
(string, int, double) person = ("Tom", 25, 81.23);
```



Первый элемент кортежа в данном случае представляет строку, второй элемент - тип `int`, а третий - тип `double`.

Мы также можем дать названия полям кортежа:

```
var tuple = (count:5, sum:10);  
Console.WriteLine(tuple.count); // 5  
Console.WriteLine(tuple.sum); // 10
```

Теперь чтобы обратиться к полям кортежа используются их имена, а не названия `Item1` и `Item2`.

Кортежи также можно сравнивать с помощью операторов `==` и `!=`

Кортеж может передаваться в качестве параметра в метод.

Кортежи могут использоваться для возвращения из функции более чем одного значения

---

Деконструкция позволяет извлекать из кортежа все элементы за одну операцию.

```
var tuple = (5, 10, "a");
```

```
(int first, int second, string word) = tuple;
```

или

```
var (first, second, word) = tuple;
```

или

```
int first = 1;
```

```
int second=2;
```

```
string word ="b";
```

```
(first, second, word) = tuple;
```

### **Именованные и неименованные кортежи**

#### **1. Неименованные**

```
(string, int, bool) value = ("a", 5, true);
```

или

```
var value = ("a", 5, true);
```

#### **2. Именованные**

```
(string Name, int Age, bool Key) value = ("a", 5, true);
```

или

```
var value = (Name: "a", Age: 5, Key: true);
```

Теперь доступ возможен, как с использованием имен полей, так и с использованием системы обозначения `ItemX`.

Обращение к неименованному кортежу:

```
string a = value.Item1;
```

```
int b = value.Item2;
```

```
bool c = value.Item3;
```

Обращение к именованному кортежу включает в себя обращение, как к неименованному плюс:

```
string a = value.Name;  
int b = value.Age;  
bool c = value.Key;
```

Кортежу можно присваивать новые значения после объявления:

```
value.Item1 = "R";  
value.Item2 = 10;  
value.Item3 = false;
```

### **Инициализация проекций кортежа**

Если задано явное имя, то оно имеет приоритет.

```
var One = 1;  
var Two = "text";  
var tuple = (Key: One, Name: Two);
```

Key и Name – явные имена. One и Two – неявные имена.

Во всех полях, где не указано явное имя, проецируется применимое неявное имя.

```
string Context = "answer";  
var mixedTuple = (42, Context);
```

```
int a = mixedTuple.Item1;  
string b = mixedTuple.Item2;  
string c = mixedTuple.Context;
```

Существует два условия при которых имена полей кандидата не проецируются в поле картежа:

1. Имя кандидата зарезервировано (ItemX, Rest, ToString и тд).
2. Имя кандидата является публикатором другого имени поля картежа.

### **Кортежи, как тип возвращаемого значения**

```
(int a, string b, bool c) Method()  
{  
    var abc = (5, "text", true);  
    return abc;  
}
```

### **Деконструкция кортежа**

Все элементы кортежа можно распаковать, выполнив операцию деконструкции возвращаемого методом кортежа.

Возьмем в качестве примера функция выше:

```
(int number, string str, bool flag) = Method();
```

или

```
var(number, str, flag) = Method();
```

или

```
(int number, var str, bool flag) = Method();
```

Затем мы можем использовать объявленные и проинициализированные переменные.

```
Console.Write(number);
```

```
Console.Write(str);
```

```
Console.Write(flag);
```

**Коллекции** – один из способов группировки данных.

Коллекции находятся к пространству имен System в классе Collections.

Класс	Описание
Dictionary<TKey,TValue>	Предоставляет коллекцию пар «ключ-значение», которые упорядочены по ключу. Позволяет быстро получать доступ к элементу по ключу. Можно также быстро производить добавление и удаление также как и в List<T>.
List<T>	Представляет список объектов, доступных по индексу. Предоставляет методы для поиска по списку, его сортировки и изменения. Методы, которыми мы пользовались: Remove, Add, Sort

Коллекции предоставляют более гибкий способ работы с группами объектов. В отличие от массивов, коллекция, с которой вы работаете, может расти или уменьшаться динамически при необходимости. Некоторые коллекции допускают назначение ключа любому объекту, который добавляется в коллекцию, чтобы в дальнейшем можно было быстро извлечь связанный с ключом объект из коллекции. Коллекция является классом.

Типы коллекций:

1. System.Collections.Generic (обобщенные или типизированные классы коллекций);
2. System.Collections.Specialized (специальные классы коллекций);
3. System.Collections.Concurrent.

1. Универсальная коллекция применяется в том случае, если все элементы в коллекции имеют одинаковый тип данных.

(остальное не проходили, не стала писать)

- **void Add(T item):** добавление нового элемента в список
- **void AddRange(ICollection collection):** добавление в список коллекции или массива
- **int BinarySearch(T item):** бинарный поиск элемента в списке. Если элемент найден, то элемента в коллекции. При этом список должен быть отсортирован.
- **int IndexOf(T item):** возвращает индекс первого вхождения элемента в списке
- **void Insert(int index, T item):** вставляет элемент item в списке на позицию index
- **bool Remove(T item):** удаляет элемент item из списка, и если удаление прошло успешно
- **void RemoveAt(int index):** удаление элемента по указанному индексу index
- **void Sort():** сортировка списка

## 6. Язык запросов к источнику данных LINQ. LINQ to Objects. Методы расширения. Отложенное и немедленное выполнение.

LINQ (Language-Integrated Query) представляет простой и удобный язык запросов к источнику данных.

Или

Язык интегрированных запросов (LINQ) представляет собой набор языковых и платформенных средств для описания структурированных, безопасных в отношении типов запросов к локальным коллекциям объектов и удаленным источникам данных.

LINQ to Objects: применяется для работы с массивами и коллекциями.

```
string[] teams = {"Бавария", "Боруссия", "Реал Мадрид", "Манчестер Сити",
"ПСЖ", "Барселона"};
var selectedTeams = from t in teams // определяем каждый объект из teams как t
                     where t.ToUpper().StartsWith("Б") //фильтрация по критерию
                     orderby t // упорядочиваем по возрастанию
                     select t; // выбираем объект
```

```
foreach (string s in selectedTeams)
    Console.WriteLine(s); // Бавария, Барселона, Боруссия
```

Простейшее определение запроса LINQ выглядит следующим образом:

```
from переменная in набор_объектов
select переменная;
```

Кроме стандартного синтаксиса from .. in .. select для создания запроса LINQ мы можем применять специальные методы расширения, которые определены для интерфейса IEnumerable. Как правило, эти методы реализуют ту же функциональность, что и операторы LINQ типа where или orderby.

Например:

```
string[] teams = { "Бавария", "Боруссия", "Реал Мадрид", "Манчестер Сити",  
"ПСЖ", "Барселона" };
```

```
var selectedTeams = teams.Where(t=>t.ToUpper().StartsWith("Б")).OrderBy(t =>  
t);
```

```
foreach (string s in selectedTeams)  
    Console.WriteLine(s);
```

Запрос `teams.Where(t=>t.ToUpper().StartsWith("Б")).OrderBy(t => t)` будет аналогичен предыдущему. Он состоит из цепочки методов `Where` и `OrderBy`. В качестве аргумента эти методы принимают делегат или лямбда-выражение.

Список используемых методов расширения LINQ:

- **Select**: определяет проекцию выбранных значений
- **Where**: определяет фильтр выборки
- **OrderBy**: упорядочивает элементы по возрастанию
- **OrderByDescending**: упорядочивает элементы по убыванию
- **ThenBy**: задает дополнительные критерии для упорядочивания элементов возрастанию
- **Join**: соединяет две коллекции по определенному признаку
- **GroupBy**: группирует элементы по ключу
- **Reverse**: располагает элементы в обратном порядке
- **All**: определяет, все ли элементы коллекции удовлетворяют определенному условию
- **Contains**: определяет, содержит ли коллекция определенный элемент и т.д

При отложенном выполнении LINQ-выражение не выполняется, пока не будет произведена итерация или перебор по выборке. То есть, в примере выше мы написали LINQ выражение (`from t in teams...`) (просто сформировали запрос и, фактически, он выполнится когда мы используем далее `foreach`) и там нет метода, который возвращает результат (`Count`, `ToList`) => это отложенное выполнение. Если бы там был метод `Count` или `ToList`, то запрос выполнялся бы сразу и это было бы немедленное выполнение. Если мы используем методы `Count` и `ToList`, то неявно вызывается `foreach`.

Важная особенность большинства операций запросов – выполнение не при конструировании, а во время перечисления.

Отложенное выполнение поддерживают все стандартные операции запросов со

следующими исключениями:

- операции, которые возвращают одиночный элемент или скалярное значение

(First,Count)

- операции преобразования ToArray, ToList, ToDictionary, ToLookup

При отложенном выполнении LINQ-выражение не выполняется, пока не будет произведена итерация или перебор по выборке. Рассмотрим отложенное выполнение:

```
1 string[] teams = {"Бавария", "Боруссия", "Реал Мадрид", "Манчестер Сити", "ПСЖ", "Барселона"};
2
3 var selectedTeams = from t in teams where t.ToUpper().StartsWith("Б") orderby t select t;
4
5 // выполнение LINQ-запроса
6 foreach (string s in selectedTeams)
7     Console.WriteLine(s);
```

То есть фактическое выполнение запроса происходит не в строке определения: `var selectedTeams = from t...`, а при переборе в цикле `foreach`.

## Немедленное выполнение запроса

С помощью ряда методов мы можем применить немедленное выполнение запроса. Это методы, которые возвращают одно атомарное значение или один элемент. Например, `Count()`, `Average()`, `First()` / `FirstOrDefault()`, `Min()`, `Max()` и т.д. Например, метод `Count()` возвращает числовое значение, которое представляет количество элементов в полученной последовательности. А метод `First()` возвращает первый элемент последовательности. Но чтобы выполнить эти методы, вначале надо получить саму последовательность, то есть результат запроса, и пройти по ней циклом `foreach`, который вызывается неявно внутри структуры запроса.

## 7. Оператор «using». Директива «region». Пространства имен в языке C#.

Директива **using** - применяется для определения или разрешения использования типов как пространств имен.

```
using System.IO;
```

### Оператор using

```
public class Person : IDisposable
{
    public string Name { get; set; }
    public void Dispose()
    {
        Console.WriteLine("Disposed");
    }
}

using (Person p = new Person { Name = "Tom" })
{
    // переменная p доступна только в блоке using
    Console.WriteLine($"Некоторые действия с объектом Person.
Получим его имя: {p.Name}");
}
Console.WriteLine("Конец метода Test");

using (StreamWriter sw = new StreamWriter("file.txt"))
{
    sw.WriteLine("Hello");
    //sw.close();
}
```

1. **Оператор using** - задает область видимости, вне которой объект недоступен.
2. В блоке **using(созданный объект){...}** объект доступен только для чтения, изменить и переназначить его невозможно.
3. Все переменные объявленные в блоке **using** действуют только в блоке **using(...){...}**
4. Использовать оператор **using** можно только для тех объектов класса, чей класс наследуется от интерфейса **IDisposable**. (есть функция **Dispose()**);



Блок `using() {...}` по сути компилятором превращается в блок `try {...} finally{...}`. И служит, как я понял, для обработки ошибок, то есть он выполняется не зависимо от исхода работы программы.

Использование инструкции `using` обеспечивает вызов `Dispose` (или `DisposeAsync`), даже если в блоке `using` возникает исключение.

`Dispose` выполняет определенные приложением задачи, связанные с освобождением или сбросом неуправляемых ресурсов. То есть, этот метод используется, чтобы закрыть или освободить неуправляемые ресурсы, такие как файлы, потоки и дескрипторы, удерживаемые экземпляром класса, реализующего этот интерфейс

Представь, что ты работаешь с файлом, ты его открыла, пишешь туда что-то, и возникла ошибка, которую ты никак не отлавливала, файл твой не сохраняется и все летит к чертям. А мы помним, что файловые потоки нужно закрывать, так вот, конструкция `using` это делает все сама, закрывает поток и в случае удачного и в случае неудачного работы блока кода, завершает она поток корректно.

Такой подход крайне важен для эффективного управления памятью.

---

### **Директива `using` используется в следующих трех целях**

- Для разрешения использования типов в пространстве имен, чтобы не нужно было квалифицировать использование типа в этом пространстве имен:  
`using System.Text;`
- Для разрешения доступа к статическим членам и вложенным типам без необходимости квалифицировать доступ с помощью имени типа.  
`using static System.Math;`
- Чтобы создать псевдоним для пространства имен или типа. Это называется *директивой `using static`*  
`using Project = PC.MyCompany.Project;`

Область директивы `using` ограничена файлом, в котором она находится.

Директива `using` может отображаться:


- В начале файла исходного кода перед определением пространств имен и типов.
- В пространстве имен перед любыми пространствами имен и типами, объявленными в этом пространстве имен.


Директива **#region** позволяет указать блок кода, который можно разворачивать и сворачивать.

Все определяемые классы и структуры, как правило, не существуют сами по себе, а заключаются в специальные контейнеры - пространства имен. Пространство имен определяется с помощью ключевого слова `namespace`, после которого идет название

C#	 Копировать
<pre>System.Console.WriteLine("Hello World!");</pre>	

[System](#) является пространством имен, а [Console](#) — это класс в нем. Можно использовать ключевое слово `using`, и тогда полное имя не потребуется. См. следующий пример:

C#	 Копировать
<pre>using System;</pre>	

C#	 Копировать
<pre>Console.WriteLine("Hello");</pre>	

Можно объявить собственное пространство имен, что поможет вам контролировать область имен классов и методов в более крупных проектах.

Пространства имен имеют следующие свойства:

- Они помогают упорядочивать проекты с большим объемом кода.
- Они разделяются оператором `.` (точка)
- Директива `using` позволяет не указывать название пространства имен для каждого класса.
- Пространство имен `global` является корневым: `global::System` всегда будет ссылаться на пространство имен `System` в `.NET`.

## 8. Анонимные типы. Ключевое слово «var». Оператор `foreach`.

Анонимные типы позволяют создать объект с некоторым набором свойств без определения класса. Анонимный тип определяется с помощью ключевого слова `var` и инициализатора объектов:

```
var user = new { Name = "Tom", Age = 34 };  
Console.WriteLine(user.Name);
```

В данном случае `user` - объект анонимного типа, у которого определены два свойства `Name` и `Age`. И мы также можем использовать его свойства, как и у обычных объектов классов. Однако тут есть ограничение - свойства анонимных типов доступны только для чтения.

При этом во время компиляции компилятор сам будет создавать для него имя типа и использовать это имя при обращении к объекту. Нередко анонимные типы имеют имя наподобие "`<>f__AnonymousType0'2`".

Для исполняющей среды CLR анонимные типы будут также, как и классы, представлять ссылочный тип.

---

Ключевое слово **`var`** ссылается на тип неявным способом. Это псевдоним любого типа. Реальный тип определит компилятор C#.

Использование **`var`** никак не ухудшает производительность.

---

Оператор цикла `foreach` служит для перебора элементов коллекции. К коллекциям относятся массивы, списки `List` и пользовательские классы коллекций. В данном операторе не нужно создавать переменную-счетчик для доступа к элементам коллекции, в отличие от других циклов. Оператор `foreach` имеет следующую структуру:

```
foreach ([тип] [переменная] in [коллекция])
{
    //тело цикла
}
```

Оператор в основном используется для чтения данных из коллекции (при добавлении будет ошибка)

## 9. Основные принципы Объектно-ориентированного программирования. Примеры в разрезе языка программирования C#.

Объектно-ориентированное программирование (ООП) — методология программирования, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определённого класса, а классы образуют иерархию наследования.

## Принципы ООП:

А) Инкапсуляция означает, что группа связанных свойств, методов и других членов рассматривается как единый элемент или объект. Инкапсуляция описывает способность языка скрывать излишние детали реализации от пользователя. Благодаря модификатором доступа `public`, `protected`, `private`, `internal` можно скрывать методы, свойства, поля класса от других частей программы.

Б) *Инкапсуляция* – это заточение свойств, методов, полей внутри какого-то одного класса. При всем при этом это всё позволяет нам обеспечивать модификаторы доступа(4-6). Их 6. Но достаточно назвать 4.

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.

Мы к элементу класса можем обращаться из любого места нашей программы.

- **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.

Мы не можем обращаться к элементу класса извне этого класса.

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

Мы можем обращаться к элементу класса из этого класса и внутри класса наследника.

- **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`).

Очень похож на публик.

Мы можем обращаться к элементу класса только в пределах своей сборки.

- **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.

**А)** Наследование описывает возможность создания новых классов на основе существующих классов. Наследование отражает возможность языка разрешать построение определений новых классов на основе определений существующих классов.

**Б)** *Наследование – это необходимость расширить класс который имеет свои методы свойства поля при работе с какими-то объектами(функциональность) для каких то своих нужд без переписывания самого класса. От двух классов одновременно нельзя наследоваться.(запрещено множественное наследование)*

**А)** Полиморфизм означает, что можно иметь несколько взаимозаменяемых классов, даже если каждый класс реализует одни и те же свойства или методы разными способами.

**Б)** *Полиморфизм обозначает способность трактовать связанные объекты в сходной манере. В частности, данный принцип ООП позволяет базовому классу определить набор методов, которые будут доступны всем классам наследникам и которые можно переопределить в классах наследниках. Когда возникает полиморфизм объявленный тип объекта перестает соответствовать своему типу во время выполнения.*

**А)** Абстракция – это использование только тех характеристик объекта, которые с достаточной точностью представляют его в данной системе.

**Б)** *Абстракция – это абстрагирование от каких-то свойств объекта при написании какого-то класса.*

Что это значит?

Если описываешь какого-то человека в своей программе не обязательно писать какой у него вес или рост, если я являюсь сотрудником деканата. Важно то как его зовут и какого он года рождения предположим. *То есть Я от других свойств и характеристик абстрагируюсь.*

## **10. Принцип ООП «Абстракция». Классы и объекты. Элементы класса. Статические классы и элементы класса.**

Абстракция (как поняла это я) – отделение реализации функции от ее описания (интерфейс и реализация интерфейса или абстрактные классы).

Из википедии: Абстрагирование означает выделение значимой информации и исключение из рассмотрения незначимой. В ООП рассматривают лишь абстракцию данных (нередко называя её просто «абстракцией»),

подразумеваю набор наиболее значимых характеристик объекта, доступных остальной программе.

Класс - самодостаточный фрагмент кода, обладающий набором свойств и методов для описания некоторого объекта реального мира. Объект представляет экземпляр класса.

Static (для элемента класса) означает то, что этот элемент является «общим» для всех объектов класса (если меняется в одном, то в другом тоже изменится) также такими могут быть свойства и методы. Все что помечено static вызывается от имени класса, а то что не статическое вызывается от имени объекта.

В статическом классе все поля, свойства являются статическими (должны быть статическими). У статического класса не может быть объектов.

Статический конструктор вызывается только один раз. Статический конструктор вызывается раньше обычного. Память под статический элемент выделяется даже если не создано ни одного объекта класса.

Статические методы могут обращаться только к статическим членам класса.

Члены класса могут быть статическими членами или членами экземпляра(у каждого объекта свои). Статические члены принадлежат классу в целом, а члены экземпляра принадлежат конкретным объектам (экземплярам классов).

Ниже перечислены виды членов, которые могут содержаться в классе.

- Константы
  - Константные значения, связанные с классом.
- Поля
  - Переменные класса.
- Методы
  - Вычисления и действия, которые может выполнять класс.
- Свойства
  - Действия, связанные с чтением и записью именованных свойств класса.
- Индексаторы
  - Действия, реализующие индексирование экземпляров класса, чтобы обращаться к ним как к массиву.
- События
  - Уведомления, которые могут быть созданы этим классом.
- Операторы
  - Поддерживаемые классом операторы преобразования и выражения.
- Конструкторы
  - Действия, необходимые для инициализации экземпляров класса или класса в целом.

- Методы завершения
  - Действия, выполняемые перед окончательным удалением экземпляров класса.
- Типы
  - Вложенные типы, объявленные в классе.

В С# показательным примером статического класса является класс `Math`, который применяется для различных математических операций.

**Статический** класс в основном такой же, как и нестатический класс, но имеется одно отличие: нельзя создавать экземпляры статического класса. Другими словами, нельзя использовать оператор [new](#) для создания переменной типа класса. Поскольку нет переменной экземпляра, доступ к членам статического класса осуществляется с использованием самого имени класса. Например, если есть статический класс, называемый `UtilityClass`, имеющий открытый статический метод с именем `MethodA`, вызов метода выполняется, как показано в следующем примере:

```
UtilityClass.MethodA();
```

Статический класс может использоваться как обычный контейнер для наборов методов, работающих на входных параметрах, и не должен возвращать или устанавливать каких-либо внутренних полей экземпляра

Ниже приведены основные возможности статического класса.

- Содержит только статические члены.
- Создавать его экземпляры нельзя.
- Является запечатанным.
- Не может содержать [конструкторы экземпляров](#).

Преимущество применения статических классов заключается в том, что компилятор может проверить отсутствие случайно добавленных членов экземпляров. Таким образом, компилятор гарантирует невозможность создания экземпляров таких классов.

Статические классы запечатаны, поэтому их нельзя наследовать. Они не могут наследовать ни от каких классов, кроме [Object](#).

Нестатический класс может содержать статические методы, поля, свойства или события. Статический член вызывается для класса даже в том случае, если не создан экземпляр класса. Доступ к статическому члену всегда выполняется по имени класса, а не экземпляра. Существует только одна копия статического члена, независимо от того, сколько создано экземпляров класса. Статические методы и свойства не могут обращаться к нестатическим полям и событиям в их содержащем типе, и они не могут обращаться к переменной экземпляра объекта, если он не передается явно в параметре метода.

С# не поддерживает статические локальные переменные

Статические члены инициализируются перед первым доступом к статическому члену и перед вызовом статического конструктора, если

таковой имеется. Для доступа к члену статического класса следует использовать имя класса, а не имя переменной. Если класс содержит статические поля, должен быть указан статический конструктор, который инициализирует эти поля при загрузке класса.

```
public class Automobile
{
    public static int NumberOfWheels = 4;
    public static int SizeOfGasTank
    {
        get
        {
            return 15;
        }
    }
    public static void Drive() { }
    public static event EventType RunOutOfGas;

    // Other non-static fields and properties...
}
```

-----

```
Automobile.Drive();
int i = Automobile.NumberOfWheels;
```

## 11. Свойства класса. Члены, воплощающие выражение. Автоматически реализуемое свойство.

Свойства (get и set) обеспечивают простой доступ к полям классов и структур, позволяют узнать их значение или выполнить их установку.

[модификатор\_доступа] возвращаемый\_тип произвольное\_название

```
{
    // код свойства
}
```

Пример:

```
class Person
{
    private string name;

    public string Name
    {
        get{ return name;}
    }
}
```



```

        set{name = value;}
    }
}

```

Если свойство определяют только блок `get`, то такое свойство доступно только для чтения - мы можем получить его значение, но не установить. И, наоборот, если свойство имеет только блок `set`, тогда это свойство доступно только для записи - можно только установить значение, но нельзя получить.

- 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. Например, если свойство имеет модификатор `public`, то блок `set/get` может иметь только модификаторы `protected internal`, `internal`, `protected`, `private`

Токен => поддерживается в двух формах: в виде лямбда-оператора и в виде разделителя имени члена и реализации члена в определении тела выражения. Используется для удобочитаемости.

Оператор => перегрузить нельзя.

`member => expression; //expression` — любое допустимое выражение.

```
public override string ToString() => $"{fname} {lname}".Trim();
```

сокращенная версия вот этого:

```

public override string ToString()
{
    return $"{fname} {lname}".Trim();
}

```

```

public async Task<Department> GetById(Guid id)
{
    return await _dbContext.Department.FirstOrDefaultAsync(g => g.Id == id);
}

```

`public string Name { get; set; } = "Jane"` // автоматически реализуемое свойство сокращенная запись вот этого:

```

string Name { get {return Name;}
set {Name = name;}}

```

## 12. Понятие «Отражение» (Reflection).

**Рефлексия** представляет собой процесс выявления типов во время выполнения приложения. Каждое приложение содержит набор используемых классов, интерфейсов, а также их методов, свойств и прочих кирпичиков, из которых складывается приложение. И рефлексия как раз и позволяет определить все эти составные элементы приложения.

**Рефлексия** позволяет: перечислять члены типа, создавать новые экземпляры объекта, запускать на выполнение члены объекта, извлекать информацию о типе, извлекать информацию о сборке, исследовать пользовательские атрибуты, примененные к типу, создавать и компилировать новые сборки.

**Type** является корневым классом для функциональных возможностей рефлексии и основным способом доступа к метаданным. С помощью членов класса **Type** можно

получить сведения об объявленных в типе элементах: конструкторах, методах, полях, свойствах и событиях класса, а также о модуле и сборке, в которых развернут данный класс

```
class MyClass
```

```
{  
}
```

```
class Program
```

```
{
```

```
    static void Main()
```

```
    {
```

```
        var my = new MyClass();
```

```
        Type type;
```

```
        // Способы получения экземпляра класса Type.
```

```
        // 1.
```

```
        type = my.GetType();
```

```
        Console.WriteLine("1й способ: " + type);
```

```
        // 2.
```

```
        type = Type.GetType("TypeTest.MyClass"); // Полное
```

квалифицированное имя типа в строковом представлении.

```
        Console.WriteLine("2й способ: " + type);
```

```
        // 3.
```

```
        type = typeof(MyClass);
```

```
        Console.WriteLine("3й способ: " + type);
```

```
        // Delay.
```

```
        Console.ReadKey();
```

```
}
```

Получаем разную информацию о классе с помощью методов класса `Type`:

```
Type t = cl.GetType();
Console.WriteLine("Полное Имя:          {0}", t.FullName);
Console.WriteLine("Базовый класс:       {0}", t.BaseType);
Console.WriteLine("Абстрактный:        {0}", t.IsAbstract);
Console.WriteLine("Это COM объект:      {0}", t.IsCOMObject);
Console.WriteLine("Запрещено наследование: {0}", t.IsSealed);
Console.WriteLine("Это class:          {0}", t.IsClass);
```

### 13. Разделяемые элементы кода. Ключевое слово «partial».

Можно разделить определение класса, структуры, интерфейса или метода между двумя или более исходными файлами. Каждый исходный файл содержит часть определения класса или метода, а во время компиляции приложения все части объединяются.

При работе над большими проектами распределение класса между различными файлами позволяет нескольким программистам работать с ним одновременно.

Чтобы разделить определение класса, используйте модификатор ключевого слова `partial`, как показано ниже:

```
public partial class Employee
{
    public void DoWork()
    {
    }
}
```

```
public partial class Employee
{
    public void GoToLunch()
    {
    }
}
```

Ключевое слово `partial` указывает, что другие части класса, структуры или интерфейса могут быть определены в пространстве имен. Все части должны использовать ключевое слово `partial`. Для формирования окончательного типа все части должны быть доступны во время компиляции. Все части должны иметь одинаковые модификаторы доступа, например `public`, `private` и т. д. Если какая-либо из частей объявлена абстрактной, то весь тип будет считаться абстрактным. Если какая-либо из частей объявлена запечатанной,

то весь тип будет считаться запечатанным. Если какая-либо из частей объявляет базовый тип, то весь тип будет наследовать данный класс.

Модификатор `partial` недоступен в объявлениях делегатов или перечислений

Модификатор `partial` должен находиться непосредственно перед ключевыми словами `class`, `struct` или `interface`.

#### 14. Понятия «Куча» и «Сборщик мусора». Конструктор и финализатор класса. Виды конструкторов.

**Куча** — это хранилище памяти, также расположенное в ОЗУ, которое допускает динамическое выделение памяти и не работает по принципу стека: это просто склад для ваших переменных. Когда вы выделяете в куче участок памяти для хранения переменной, к ней можно обратиться не только в потоке, но и во всем приложении.

**Сборщик мусора** .NET управляет выделением и освобождением памяти для приложения. При каждом создании объекта среда CLR выделяет память для объекта из управляемой кучи. Пока в управляемой куче есть доступное адресное пространство, среда выполнения продолжает выделять пространство для новых объектов. Тем не менее ресурсы памяти не безграничны. В конечном счете сборщику мусора необходимо выполнить сбор, чтобы освободить память. Механизм оптимизации сборщика мусора определяет наилучшее время для выполнения сбора, основываясь на выполненных операциях выделения памяти. Когда сборщик мусора выполняет сборку, он проверяет наличие объектов в управляемой куче, которые больше не используются приложением, а затем выполняет необходимые операции, чтобы освободить память

---

**Конструкторы** — это методы классов, выполняемые автоматически при создании объекта заданного типа. Обычно конструкторы выполняют инициализацию членов данных нового объекта. Конструктор можно запустить только один раз при создании класса. Кроме того, код конструктора всегда выполняется раньше всех остальных частей кода в классе. Следует отметить, что так же, как и для других методов, можно создать несколько перегрузок конструктора.

**Финализатор** — это метод класса, который автоматически вызывается средой исполнения в промежутке времени между моментом, когда объект этого класса опознаётся сборщиком мусора как неиспользуемый, и моментом удаления объекта (освобождения занимаемой им памяти).

---

Виды конструкторов:

1. Конструктор без параметров
2. Конструктор с параметрами
3. Конструктор по умолчанию
4. Конструктор копирования

5. Статический конструктор (не имеет параметров и модификатор доступа, иниц. статич. элемент класса)

15. Принцип ООП «Наследование». Организация наследования в языке C#.

Возможность организации множественного наследования в языке C#.

Механизмы запрета наследования от класса.

*Наследование* позволяет определить дочерний класс, который использует (наследует), расширяет или изменяет возможности родительского класса. Класс, члены которого наследуются, называется *базовым классом*. Класс, который наследует члены базового класса, называется *производным классом*.

C# и .NET поддерживают только *одиночное наследование*.

Не все члены базового класса наследуются производными классами. Следующие члены не наследуются.

- [Статические конструкторы](#), которые инициализируют статические данные класса.
- [Конструкторы экземпляров](#), которые вызываются для создания нового экземпляра класса. Каждый класс должен определять собственные конструкторы.
- [Методы завершения](#), которые вызываются сборщиком мусора среды выполнения для уничтожения экземпляров класса.

Производные классы могут также *переопределять* унаследованные члены, то есть предоставлять альтернативную реализацию. Переопределить можно только те члены, которые в базовом классе отмечены ключевым словом [virtual](#) (виртуальный).

Члены базового класса, отмеченные ключевым словом [abstract](#) (абстрактный), обязательно должны переопределяться в производных классах.

Наследование применяется только для классов и интерфейсов. Другие категории типов (структуры, делегаты и перечисления) не поддерживают наследование.

---

Ключевое слово [sealed](#) позволяет предотвратить наследование класса или определенных членов класса, помеченных ранее как [virtual](#).

«Запечатанный» класс

[public sealed class D](#)

```
{  
    // Class members here.  
}
```

Запечатанный класс не может использоваться в качестве базового класса. Поэтому он также не может быть абстрактным классом. Запечатанные классы предотвращают наследование. Поскольку их нельзя использовать в качестве базовых классов, определенная оптимизация во время выполнения позволяет несколько ускорить вызов членов запечатанных классов.

## 16. Ссылки на объекты. Преобразование типов данных. Явное и неявное преобразование типов данных в языке C#.

Могут принимать значение null.

При передаче параметров по ссылке перед параметрами используется модификатор ref:

```
static void Main(string[] args)  
{  
    int x = 10;  
    int y = 15;  
    Addition(ref x, y); // вызов метода  
    Console.WriteLine(x); // 25  
  
    Console.ReadLine();  
}  
// параметр x передается по ссылке  
static void Addition(ref int x, int y)  
{  
    x += y;  
}
```

По сути, как и в случае с ключевым словом ref, ключевое слово out применяется для передачи аргументов по ссылке. Однако в отличие от ref для переменных, которые передаются с ключевым словом out, не требуется инициализация. И кроме того, вызываемый метод должен обязательно присвоить им значение.

---

Преобразование типов данных, это приведение одного типа к другому. Например, приведение целочисленной переменной к числу с плавающей точной, или преобразование числа в строку. В C# выделяют два варианта преобразования типов:

- **Неявное преобразование типов.** Это, так называемое безопасное преобразование типов в C#. Например, преобразование из

типа **float** (более «маленький» тип) в тип **double** (более «большой» тип). При таком преобразовании никакая информация не «потеряется», что при обратном преобразовании вполне возможно.

```
double a = 3;
```

```
float c = 2;
```

```
a = c;
```

- **Явное преобразование типов.** Такое преобразование выполняется программистом с прямым указанием типа, к которому нужно привести переменную. Для такого преобразования требуется наличие оператора преобразования.

```
int intData = (int)doubleData;
```

Или

```
int intData = Convert.ToInt32(doubleData);
```

## 17. Принцип ООП «Инкапсуляция». Модификаторы доступа в языке C#.

Ключевое слово «readonly».

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.

---

Ключевое слово `readonly` — это модификатор, который может использоваться в четырех контекстах:

- В **объявлении поля** `readonly` указывает на то, что присвоение значения полю может происходить только при объявлении или в конструкторе этого класса. Полю только для чтения можно несколько раз назначить значения в объявлении поля и в конструкторе.

- В определении типа `readonly struct` объект `readonly` указывает на то, что тип структуры является неизменяемым.
- В **определении элемента** `readonly` указывает, что элемент `struct` не изменяет внутреннее состояние структуры.
- В **возврате метода** `ref readonly` модификатор `readonly` указывает, что метод возвращает ссылку, и записи для этой ссылки не допускаются.

```
class Age
{
    readonly int year;
    Age(int year)
    {
        this.year = year;
    }
    void ChangeYear()
    {
        //year = 1967; // Compile error if uncommented.
    }
}
```

Можно присвоить значение полю `readonly` только в следующих контекстах:

- Когда переменная инициализируется в объявлении, например:

```
public readonly int y = 5;
```

- В конструкторе экземпляра класса, содержащего объявление поля экземпляра.
- В статическом конструкторе класса, содержащего объявление статического поля.

Можно ключевым словом `readonly` пометить структуру => все поля в структуре тоже должны быть помечены ключевым словом `readonly`

## 18. Принцип ООП «Полиморфизм». Примеры полиморфных реакций.

**Полиморфизм** – свойство, которое позволяет использовать одно и тоже имя функции для решения двух и более схожих, но технически разных задач.

**Полиморфизм** – возможность замещения методов объекта родителя методами объекта-потомка, имеющих то же имя.

Пример с двумя функциями `print` с разными параметрами.

Полиморфизм – один интерфейс со множеством реализаций.



## 19. Виртуальные члены класса. Соккрытие членов базового класса новыми членами (ключевые слова «new» и «override»). Примеры использования.

### Переопределение метода «ToString()».

Метод, при определении которого присутствует слово `virtual`, называется виртуальным. Каждый класс - наследник может иметь собственную версию виртуального метода, называется это переопределением и обозначается словом `override`. Переопределенные методы обеспечивают поддержку полиморфизма. Полиморфизм позволяет определять в базовом классе методы, которые будут общими для всех наследников, но каждый наследник, в случае необходимости, может иметь их собственные реализации.

Естественно, что интерфейсы виртуального метода и всех его версий должны полностью совпадать. Таким образом, применение виртуальных методов позволяет фиксировать интерфейс метода и потом разработать под этот интерфейс новые реализации. Виртуальными могут быть и свойства, и индексы.

Нельзя переопределить или объявить виртуальным статический метод.

```
class Shape
```

```
{
    protected int a, h;
    public Shape(int x, int y)
    {
        a = x;
        h = y;
    }
    public virtual void Show_area()
    { // вводится виртуальный метод
        Console.WriteLine("Площадь будет определен позже");
    }
}
```

```
class Tri : Shape
```

```
{
    int s;
    public Tri(int x, int y) : base(x, y)
    { }
    public override void Show_area()
    { //первое переопределение виртуального метода
        s = a * h / 2;
        Console.WriteLine("Площадь треугольника= " + s);
    }
}
```

```
class Class1
```

```
{
```

```

static void Main(string[] args)
{
    Shape q = new Shape(10, 30); // Площадь будет определен позже
    q.Show_area();

    Tri z1 = new Tri(5, 12); // Площадь треугольника= 30

    z1.Show_area();

    Console.ReadLine();
}
}

```

---

Если вы хотите, чтобы производный класс имел член с тем же именем, что и член в базовом классе, можно использовать ключевое слово new, чтобы скрыть член базового класса. Ключевое слово new вставляется перед типом возвращаемого значения замещающего члена класса. Примером является следующий код:

```

public class BaseClass
{
    public void DoWork() { WorkField++; }
    public int WorkField;
}

public class DerivedClass : BaseClass
{
    public new void DoWork() { WorkField++; }
    public new int WorkField;
}

```

Доступ к скрытым членам базового класса можно осуществлять из клиентского кода приведением экземпляра производного класса к экземпляру базового класса. Пример:

```

DerivedClass B = new DerivedClass();
B.DoWork(); // Calls the new method.

```

```

BaseClass A = (BaseClass)B;
A.DoWork(); // Calls the old method.

```

```

// ((MyBase)covid).Covid19; обращение к ф-ям, св-вам и т.д. класса отца
// при использовании override ((MyBase)covid).Covid19; теряет смысл, используется
последняя ф-я, св-во и т.д.;
// чтобы это работало нужно переопределять ф-ю, св-во и т.д. с помощью оператора new

```

---

Чтобы переопределить метод ToString в классе или структуре, выполните указанные ниже действия.

1. Объявите метод ToString со следующими модификаторами и типом возвращаемого значения:

```
public override string ToString(){}
```

## 20. Абстрактные методы и классы. Процедуры создания и реализации.

### Примеры использования.

В C# существует возможность введения в базовом классе методов, а их реализацию оставить «на потом». Такие методы называют абстрактными. Абстрактный метод автоматически является и виртуальным, но писать это нельзя. Класс, в котором имеется хотя бы один абстрактный метод, тоже называется абстрактным и такой класс может служить только в качестве базового класса. Создать объекты абстрактных классов невозможно, потому что там нет реализации абстрактных методов. Чтобы класс – наследник абстрактного класса не был в свою очередь абстрактным (хотя и это не запрещено), там должны содержаться переопределения всех наследованных абстрактных методов. Абстрактные члены классов не должны иметь модификатор private

Абстрактными могут быть:

- Методы
- Свойства
- Индексаторы
- События

Abstract должен стоять и в заголовке класса, и в заголовке метода, иначе ошибка

```
abstract class Shape
{
    public abstract int GetArea();
}
```

```
class Square : Shape
{
    int side;
```

```
    public Square(int n) => side = n;
```

```
    // GetArea method is required to avoid a compile-time error.
    public override int GetArea() => side * side;
```

```
static void Main()
{
    var sq = new Square(12);
    Console.WriteLine($"Area of the square = {sq.GetArea()}");
}
}
// Output: Area of the square = 144
```

Абстрактные классы предоставляют следующие возможности:

- Создавать экземпляры абстрактного класса нельзя.
- Абстрактный класс может содержать абстрактные методы и методы доступа.
- Изменить абстрактный класс с модификатором **sealed** нельзя, так как два этих модификатора имеют взаимоисключающие значения. Модификатор sealed запрещает наследование класса, в то время как модификатор abstract указывает, что класс обязан иметь производные классы.
- Неабстрактный класс, производный от абстрактного класса, должен включать фактические реализации всех наследуемых абстрактных методов и методов доступа.

## 21. Интерфейсы. Процедуры создания и реализации интерфейсов. Элементы интерфейса. Интерфейсные ссылки. Примеры использования.

При практическом программировании иногда возникает необходимость определения действий, которые должны выполняться классом, но без уточнения способов их выполнения. Один способ достижения этого был рассмотрен выше – это абстрактные методы. Абстрактный класс содержит интерфейс метода, но не содержит его реализации. В С# этот подход расширен введением интерфейсов, которые содержат только интерфейсы методов, без их реализации. Таким образом, можно полностью отделить интерфейс класса от его реализации. Описание интерфейса:

```
interface имя_интерфейса
{
    Тип_возвращаемого_значения имя_1(параметры);
    Тип_возвращаемого_значения имя_1(параметры);
    ...
}
```

Абстрактный пример: Интерфейс, в котором есть метод Летать. Летать могут и птицы и самолеты, но реализация этого интерфейса будет разной, т.к. птица летает с помощью крыльев, а самолет с помощью двигателей.

Реализация интерфейса должна находиться в классе – наследнике интерфейса. Ограничение единственного предка в С# на интерфейсы не распространяется: класс может иметь только один класс – предок и сколько

удовно интерфейсов – предков. Однако класс должен содержать реализации всех методов, которые содержатся в наследованном интерфейсе. Кроме этого, класс может содержать и собственные методы по общим правилам.

```
interface IControl
```

```
{  
    void Paint();  
}
```

```
interface IDataBound
```

```
{  
    void Bind(int b);  
}
```

```
public class EditBox: IControl, IDataBound //класс реализует одновременно два  
интерфейса
```

```
{  
    public void Paint() { }  
    public void Bind(int b) { }  
}
```

С интерфейсами связана еще одна интересная возможность: интерфейсные ссылки. Можно объявить переменную типа интерфейс, которая затем может ссылаться на любой класс – наследник этого интерфейса. Через эту ссылку можно получить доступ только к методам в классе, которые описаны и в интерфейсе.

```
static void Main(string[] args)
```

```
{  
    EditBox xxx = new EditBox();  
    IControl peremennaya; // Создадим ссылку на интерфейс  
    peremennaya = xxx;  
    peremennaya.Paint();  
}
```

Интерфейс не может иметь поля данных. В интерфейсе все методы и свойства являются абстрактными. Можно наследоваться от нескольких интерфейсов, а от нескольких абстрактных классов нельзя.

**22. Тип «Делегат». Процедуры объявления и создания экземпляра делегата. Многоадресный делегат. Последовательность выполнения методов, поставленных в соответствие делегату. Использование делегата Func<T, TResult>.**

Делегат — это объект, который может ссылаться на метод. То есть делегаты - это указатели на методы и с помощью делегатов мы можем вызвать данные методы.

Во время выполнения программы один и тот же делегат можно использовать для вызова различных методов, просто заменив метод, на который ссылается этот делегат. Таким образом, метод, который будет вызван делегатом, определяется не в период компиляции программы, а во время ее работы.

`delegate` тип\_возвращаемого\_значения  
`имя_делегата`(список\_формальных\_параметров);

Использование делегата:

Имя\_делегата указатель\_на\_делегат;

Указатель\_на\_делегат = `new` `имя_делегата`(имя\_функции);

Или

Имя\_делегата тип\_делегата = Имя\_функции;

```
delegate void delegate1();
public class Shape
{
    protected int a, h;
    public Shape(int x, int y)
    {
        a = x;
        h = y;
    }
    public virtual void Show_area()
    { // вводится виртуальный метод
        Console.WriteLine("Площадь будет определен позже");
    }
}
class Tri : Shape
{
    int s;
    public Tri(int x, int y) : base(x, y)
    { }
    public override void Show_area()
    { //первое переопределение виртуального метода
        s = a * h / 2;
        Console.WriteLine("Площадь треугольника= " + s);
    }
}
```

```

class Program
{
    static void Main(string[] args)
    {
        Shape q = new Shape(1, 2);
        delegate1 f1 = new delegate1(q.Show_area);
        // объявим переменную типа делегат f1 и ставим ей в соответствие
        // функцию Show_area из объекта q класса Tri

        Tri xxx = new Tri(10, 20);

        delegate1 f2 = new delegate1(xxx.Show_area);
        // объявим переменную типа делегат f2 и ставим ей в соответствие
        // функцию Show_area из объекта r класса Square
        //
        f1(); //идентичен вызову q.Show_area(); // Площадь будет определен
    позже
    f2(); // идентичен вызову r.Show_area();// Площадь треугольника
    =100
        Console.ReadLine();
    }
}

```

---

События объявляются в классе с помощью ключевого слова event, после которого указывается тип делегата, который представляет

событие:

```

1 | delegate void AccountHandler(string message);
2 | event AccountHandler Notify;

```

В данном случае вначале определяется делегат AccountHandler, который принимает один параметр типа string. Затем с помощью ключевого слова event определяется событие с именем Notify, которое представляет делегат AccountHandler. Название для события может быть произвольным, но в любом случае оно должно представлять некоторый делегат.

Определив событие, мы можем его вызвать в программе как метод, используя имя события:

```

1 | Notify("Произошло действие");

```

Поскольку событие Notify представляет делегат AccountHandler, который принимает один параметр типа string - строку, то при вызове события нам надо передать в него строку.

---

### Многоадресный делегат

Многоадресный делегат содержит список присвоенных ему делегатов. При вызове многоадресного делегата поочередно вызываются представленные в его списке делегаты. Объединять можно только делегаты одного типа.

```

delegate int Deleg(ref string st); //объявим делегат
class Class1
{
    public static int met1(ref string x)
    {
        Console.WriteLine("I am Metod 1");
        x += " 11111";
        return 5;
    }
    public static int met2(ref string y)
    {
        Console.WriteLine("I am Metod 2");
        y += " AAAAAA";
        return 55;
    }
    static void Main(string[] args)
    {
        Deleg d1;
        int k;
        string r = "*****";
        Deleg d2 = new Deleg(met1);
        // связываем делегат и функцию
        Deleg d3 = new Deleg(met2);
        d1 += d2; // присоединим первый
        d1 += d3; // добавим второй делегат
        k = d1(ref r);
        Console.WriteLine(r);
        Console.WriteLine("k=" + k);
        Console.ReadLine();
    }
}

```

В качестве ответа получим:

```

I am Metod 1
I am Metod 2
***** 11111 AAAAAA
K=55

```

делегат

Как видно из примера, в случае, когда делегат имеет возвращаемое значение (в нашем случае int), значением многоадресного делегата будет значение, возвращенное последней функцией (в нашем случае met2).

Подведем итоги: делегаты расширяют знакомые нам средства программирования в двух случаях:

- Делегаты как указатели на функцию, что позволяет использовать функции в качестве формальных/фактических параметров других функций.
- Многоадресные делегаты, таким образом получим возможность одним вызовом обеспечить выполнение ряда функций. Использование делегата в роли псевдонима функции может иногда уменьшить объем наших записей, но не расширяет наши возможности.



Делегат `Func<T, TResult>` возвращает результат действия и может принимать параметры. `Func<in T1, in T2,...in T16, out TResult>()` может принимать до 16 параметров.

Данный делегат также часто используется в качестве параметра в методах:

```
static void Main(string[] args)
{
    Func<int, int> retFunc = Factorial;
    int n1 = GetInt(6, retFunc);
    Console.WriteLine(n1); // 720

    int n2 = GetInt(6, x=> x *x);
    Console.WriteLine(n2); // 36

    Console.Read();
}

static int GetInt(int x1, Func<int, int> retF)
{
    int result = 0;
    if (x1 > 0)
        result = retF(x1);
    return result;
}

static int Factorial(int x)
{
    int result = 1;
    for (int i = 1; i <= x; i++)
    {
        result *= i;
    }
    return result;
}
```

## 23. Универсальные шаблоны (Generics). Принципы использования. Примеры. Отличия от использования процедуры упаковки-распаковки (Boxing-Unboxing).

Благодаря универсальным шаблонам вы можете создавать классы и методы с типами, спецификация которых отложена до момента объявления и создания экземпляров в клиентском коде. Как пример, ниже показан класс с параметром `T` универсального типа. Этот класс может использоваться в другом клиентском коде, не требуя ресурсов и не создавая рисков, связанных с операциями приведения и упаковки-преобразования в среде выполнения.

// Declare the generic class.

```
public class GenericList<T>
{
    public void Add(T input) { }
}
```

Общие сведения об универсальных шаблонах:

- Используйте универсальные типы, чтобы получить максимально широкие возможности многократного использования кода, обеспечения безопасности типов и повышения производительности.
- Чаще всего универсальные шаблоны используются для создания классов коллекций.
- Вы можете создавать собственные универсальные интерфейсы, классы, методы, события и делегаты.
- Сведения о типах, используемых в универсальном типе данных, можно получить во время выполнения с помощью отражения (рефлексии).

Если нам необходима гибкость (можно переопределять несколько раз) в типе, то мы можем использовать тип `object`, который в свою очередь во время компиляции упаковывается и распаковывается в нужный тип, но это очень сильно влияет на производительность, а можем использовать `Generic`, который не использует упаковку и распаковку (исп. универсальный шаблон), соответственно производительность не падает.

## 24. Перегрузка операторов. Синтаксис перегрузки. Перегрузка противоположных операций. Примеры. Недопустимость перегрузки некоторых операторов.

Перегрузка позволяет указать пользовательскую реализацию операции, если один или оба операнда принадлежат этому типу.

Для объявления оператора используйте ключевое слово `operator`. Объявление оператора должно соответствовать следующим правилам:

- Оно должно включать `public` и модификатор `static`.
- У унарного оператора один входной параметр. У бинарного оператора два входных параметра. В каждом случае хотя бы один параметр должен иметь тип `T` или `T?`, где `T` — тип, который содержит объявление оператора.

```
class Counter
```

```
{
    public int Value { get; set; }

    public static Counter operator +(Counter c1, Counter c2)
    {
        return new Counter { Value = c1.Value + c2.Value };
    }
}
```

```

public static bool operator >(Counter c1, Counter c2)
{
    return c1.Value > c2.Value;
}

}

class Program
{
    static void main(){
        Counter c1 = new Counter { Value = 23 };
        Counter c2 = new Counter { Value = 45 };
        Counter c3 = c1 + c2;
    }
}

```

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

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

Нельзя перегрузить тернарный оператор и оператор присваивания, ключевые слова, =>

## 25. Индексатор. Синтаксис реализации индексатора. Примеры использования.

Индексаторы позволяют индексировать объекты и обращаться к данным по индексу. Фактически с помощью индексаторов мы можем работать с объектами как с массивами. По форме они напоминают свойства со стандартными блоками get и set, которые возвращают и присваивают значение.

Индексатор можно перегружать, так же его можно сделать виртуальным и абстрактным.

Формальное определение индексатора:

возвращаемый\_тип this [Тип параметр1, ...]

```

{
    get { ... }
    set { ... }
}

```

В отличие от свойств индексатор не имеет названия. Вместо него указывается ключевое слово `this`, после которого в квадратных скобках идут параметры. Индексатор должен иметь как минимум один параметр. Посмотрим на примере. Допустим, у нас есть класс `Person`, который представляет человека, и класс `People`, который представляет группу людей. Используем индексаторы для определения класса `People`:

```
class Person
{
    public string Name { get; set; }
}
class People
{
    Person[] data;
    public People()
    {
        data = new Person[5];
    }
    // индексатор
    public Person this[int index]
    {
        get
        {
            return data[index];
        }
        set
        {
            data[index] = value;
        }
    }
}
```

Конструкция `public Person this[int index]` и представляет индексатор. Здесь определяем, во-первых, тип возвращаемого или присваиваемого объекта, то есть тип `Person`. Во-вторых, определяем через параметр `int index` способ доступа к элементам.

По сути все объекты `Person` хранятся в классе в массиве `data`. Для получения их по индексу в индексаторе определен блок `get`. Поскольку индексатор имеет тип `Person`, то в блоке `get` нам надо вернуть объект этого типа с помощью оператора `return`. Здесь мы можем определить разнообразную логику. В данном случае просто возвращаем объект из массива `data`.

В блоке `set` получаем через параметр `value` переданный объект `Person` и сохраняем его в массив по индексу.

После этого мы можем работать с объектом `People` как с набором объектов `Person`

```
class Program
```

```

{
static void Main(string[] args)
{
    People people = new People();
    people[0] = new Person { Name = "Tom" };
    people[1] = new Person { Name = "Bob" };

    Person tom = people[0];
    Console.WriteLine(tom.Name);

    Console.ReadKey();
}
}

```

`type_index` – тип индекса (позиции) индексатора. Как правило это тип `int`. Однако, допускается наличие других типов (например `double`, `char`);

## 26. Исключительные ситуации. Примеры исключительных ситуаций.

### Процедура обработки исключительных ситуаций. Основные блоки

Исключительная ситуация – это нарушение нормального хода выполнения программы в результате ошибки.

Элементарные примеры: деление на нуль, выход индекса за границы

```

try
{
    /* Блок кода, подлежащий проверке на наличие ошибок*/
}
catch(Исключительная_ситуация_1 exOb1)
{
    /* Обработчик для Исключительная_ситуация_1 */
}
catch
{
    /* Обработчик для не идентифицированных исключительных ситуаций*/
}
finally
{
    /* Часть программы, выполняющаяся всегда*/
}

```

Наиболее распространенные исключительные ситуации приведены в таблице.

Исключение	Значение
<i>ArrayTypeMismatchException</i>	Тип сохраняемого значения несовместим с типом массива
<i>DivideByZeroException</i>	Попытка деления на ноль
<i>IndexOutOfRangeException</i>	Индекс массива оказался вне диапазона
<i>InvalidCastException</i>	Неверно выполнено динамическое приведение типов
<i>OutOfMemoryException</i>	Обращение к оператору <b>new</b> оказалось неудачным из-за недостаточного объема свободной памяти
<i>OverflowException</i>	Имеет место арифметическое переполнение
<i>NullReferenceException</i>	Была сделана попытка использовать нулевую ссылку, т.е. ссылку, которая не указывает ни на какой объект
<i>StackoverflowException</i>	Переполнение стека

## 27. Асинхронное программирование. Определение асинхронных частей кода.

### Типы вызовов асинхронных операций.

Асинхронность позволяет вынести отдельные задачи из основного потока в специальные асинхронные методы или блоки кода. Особенно это актуально в графических программах, где продолжительные задачи могут блокировать интерфейс пользователя. И чтобы этого не произошло, нужно задействовать асинхронность.

Асинхронный метод обладает следующими признаками:

- В заголовке метода используется модификатор `async`
- Метод содержит одно или несколько выражений `await`
- В качестве возвращаемого типа используется один из следующих:
  - `void`
  - `Task`
  - `Task<T>`
  - `ValueTask<T>`

`Task` – пока все асинхронные операции не закончат свое выполнение он «удерживает» результат (перед тем как «вывести результат» мы должны дождаться пока закончатся все асинхронные операции)

слово `async`, которое указывается в определении метода, не делает автоматически метод асинхронным. Оно лишь указывает, что данный метод может содержать одно или несколько выражений `await`.

Рассмотрим пример асинхронного метода:

```
class Program
{
    static void Factorial()
    {
        int result = 1;
        for (int i = 1; i <= 6; i++)
        {
```

```

        result *= i;
    }
    Thread.Sleep(8000);
    Console.WriteLine($"Факториал равен {result}");
}
// определение асинхронного метода
static async void FactorialAsync()
{
    Console.WriteLine("Начало метода FactorialAsync"); // выполняется синхронно
    await Task.Run(() => Factorial()); // выполняется асинхронно
    Console.WriteLine("Конец метода FactorialAsync");
}

static void Main(string[] args)
{
    FactorialAsync(); // вызов асинхронного метода

    Console.WriteLine("Введите число: ");
    int n = Int32.Parse(Console.ReadLine());
    Console.WriteLine($"Квадрат числа равен {n * n}");

    Console.Read();
}

```

Здесь прежде всего определен обычный метод подсчета факториала. Для имитации долгой работы в нем используется задержка на 8 секунд с помощью метода `Thread.Sleep()`. Условно это некоторый метод, который выполняет некоторую работу продолжительное время. Но для упрощения понимания он просто подсчитывает факториал числа 6.

Также здесь определен асинхронный метод `FactorialAsync()`. Асинхронным он является потому, что имеет в определении перед возвращаемым типом модификатор `async`, его возвращаемым типом является `void`, и в теле метода определено выражение `await`.

Выражение `await` определяет задачу, которая будет выполняться асинхронно. В данном случае подобная задача представляет выполнение функции факториала:

```
await Task.Run(()=>Factorial());
```

## 28. Технология ASP.NET Core. Описание технологии. Структура проекта. Кросс-платформенность. Менеджер пакетов Nuget.

Платформа ASP.NET Core представляет технологию от компании Microsoft, предназначенную для создания различного рода веб-приложений.

Паттерны –MVVM, MVC, MVP.

---

Структура проекта:

- appsettings.json: файл конфигурации проекта в формате json
- Program.cs: главный файл приложения, с которого и начинается его выполнение. Код этого файла настраивает и запускает веб-хост, в рамках которого разворачивается приложение
- Startup.cs: файл, который определяет класс Startup и который содержит логику обработки входящих запросов

---

Core означает кроссплатформенность.

Nuget – это одно из расширений Visual Studio, которое позволяет с легкостью устанавливать, обновлять и удалять библиотеки (сборки), компоненты, инструменты.

(Для установки доп библиотек)

## 29. Концепция паттерна MVC. Основные элементы. Принципы разделения обязанностей разработчиков согласно паттерну.

Концепция паттерна (шаблона) MVC (model - view - controller) предполагает разделение приложения на три компонента:

- **Контроллер** (controller) представляет класс, обеспечивающий связь между пользователем и системой, представлением и хранилищем данных. Он получает вводимые пользователем данные и обрабатывает их. И в зависимости от результатов обработки отправляет пользователю определенный вывод, например, в виде представления.
- **Представление** (view) - это собственно визуальная часть или пользовательский интерфейс приложения. Как правило, html-страница, которую пользователь видит, зайдя на сайт.
- **Модель** (model) представляет класс, описывающий логику используемых данных. *(хранит состояние объекта и может храниться в базе данных)* Допустим есть база данных в которой хранится Юзер. **Модель** может разделяться. Либо в запросе либо как класс с доп состоянием.





Благодаря этому реализуется концепция разделение ответственности, в связи с чем легче построить работу над отдельными компонентами. Кроме того, вследствие этого приложение обладает лучшей тестируемостью. И если нам, допустим, важна визуальная часть или фронтэнд, то мы можем тестировать представление независимо от контроллера. Либо мы можем сосредоточиться на бэкэнде и тестировать контроллер.

### 30. Представление (View) в паттерне MVC. Разметка страницы. Razor-код. Шаблон Layout. Частичное представление. Создание форм в представлениях. Использование HTML-Helpers.

Представление (View) отвечает за отображение данных модели пользователю, реагируя на изменения модели. Все, что видит пользователь, генерируется видом. Представление не обрабатывает введенные данные пользователя. Представления - файлы с расширением cshtml, которые содержат код пользовательского интерфейса в основном на языке html.

---

Языком Razor по умолчанию является HTML. Razor поддерживает C# и использует символ @ для перехода с HTML на C#. Razor вычисляет выражения C# и отображает их в выходных данных HTML.

---

Частичное представление - кусочек html кода, который встраивается в вашу страницу. Если есть кусочек кода, который должен часто повторяться, то в этом случае вот это вот частичное представление удобно использовать, например, какая-то менюшка.

Layout page - это тоже самое частичное представление (возможно чужое), но в которое будет встраиваться ваше представление которые вы будете писать. Это, например, грубо говоря, "хедер" и "футер". Благодаря шаблон Layout все представления веб-приложения могут иметь единообразный стиль.

---

Часто возникает необходимость в использовании одних и тех же фрагментов дескрипторов Razor и разметки HTML в разных местах приложения. Вместо дублирования содержимого можно применять частичные представления. Это

отдельные файлы представлений, содержащие фрагменты дескрипторов и разметки, которые могут быть включены в другие представления.

---

(Дескрипторы - <HTML>, <body>, <head>)

Частичное представление используется за счет вызова вспомогательного метода `Html.Partial()` внутри другого представления.

---

Тег **<form>** устанавливает форму на веб-странице. Форма предназначена для обмена данными между пользователем и сервером.

```
<form action="URL">
```

```
...
```

```
</form>
```

---

HTML Helper — это всего лишь метод, который возвращает строку. Строка может представлять любой тип содержимого, который вы хотите.

```
@using (Html.BeginForm("Метод", "Контроллер")) { }
```

Это форма, которая по нажатию на кнопку отправляет все введенные данные запросом POST на адрес /Контроллер/Метод

Платформа ASP.NET MVC включает в себя следующий набор стандартных HTML Помощников (это не полный список):

- `Html.ActionLink()`
- `Html.BeginForm()`
- `Html.CheckBox()`

### 31. Контроллер (Controller) в паттерне MVC. Основная концепция. Логика обработки запросов. Правила наименования и создания контроллеров.

Контроллер является центральным компонентом в архитектуре MVC. Контроллер получает ввод пользователя, обрабатывает его и посылает обратно результат обработки, например, в виде представления.

Логика обработки запросов: запрос поступает из представления и в зависимости от ситуации отправляется в модель или в представление.

При использовании контроллеров существуют некоторые условности. Так, по соглашениям об именовании названия контроллеров должны оканчиваться на суффикс "Controller", остальная же часть до этого суффикса считается именем контроллера.

Логика работы вне контроллера – реализовать интерфейс в отдельном классе, а в контроллере пользоваться только объектом интерфейса.

Так же для вынесения логики работы используется Manager. Создается папка Manager, внутри нее создается интерфейс и класс, который его реализует.

Manager что-то делает с нашими данными. То есть в контроллере есть метод `ShowAll` далее этот метод вызывает наш `manager`, который получает эти

адреса и возвращает их в контроллер, потом контроллер кидает их в представление.

### 32. Маршрутизация в проекте MVC. Обработчик маршрута. Сегменты запроса. Метод «Configure» класса «Startup».

Маршрутизация в MVC это процесс соответствия входящего запроса и обработчика запроса. Обработчиками запросов в MVC обычно выступают действия контроллера.

```
template: "{controller=Department}/{action=ShowDepartment}/{id?}";
```

При дефолтном значении мы попадем в контроллер(сегмент) Студент в метод(сегмент) StudentPage из этого следует, что наш первый view будет StudentPage

При использовании параметров в шаблоне URL мы можем их помечать как необязательные с помощью знака вопроса.

При отображении различного рода запросов к серверу мы будем через слеш конкретно говорить о том, что у нас указывается имя сервера (IIS Express), потом слеш и прописывается название контроллера потом еще слеш и прописывается действие, которое нам необходимо.

В ASP.NET MVC все определения маршрутов находятся в файле **RouteConfig.cs**, который располагается в проекте в папке **App\_Start**. Класс **Startup** является входной точкой в приложение ASP.NET Core. Этот класс производит конфигурацию приложения, настраивает сервисы, которые приложение будет использовать, устанавливает компоненты для обработки запроса

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
```

Метод **Configure** устанавливает, как приложение будет обрабатывать запрос. Этот метод является обязательным. Для установки компонентов, которые обрабатывают запрос, используются методы объекта **IApplicationBuilder**. Объект **IApplicationBuilder** является обязательным параметром для метода **Configure**.

Кроме того, метод нередко принимает еще один необязательный параметр - объект **IWebHostEnvironment**, который позволяет получить информацию о среде, в которой запускается приложение, и взаимодействовать с ней.

```
{  
    app.UseDeveloperExceptionPage();
```

Страница с ошибками (перенесли из удалённого кода)

```
    app.UseStatusCodePages();//использование стандартных статус-кодов
```

Те ошибки, которые может генерировать сервис при обработке различных запросов

```
    app.UseStaticFiles();
```

использование статических файлов (из NuGet)

```
app.UseMvcWithDefaultRoute();
использование технологий MVC с дефолтной маршрутизацией, указания
стартовой страницы и правила маршрутизации
app.UseMvc(routes =>
{
    routes.MapRoute(
        name: "default",
        template:
            "{controller=Department}/{action=ShowDepartment}/{id?}");
});
```

### 33. Внедрение зависимостей (Dependency Injection). Вынесение логики работы «вне» контроллера.

Dependency injection (DI) или внедрение зависимостей представляет механизм, который позволяет сделать взаимодействующие в приложении объекты слабосвязанными. Такие объекты связаны между собой через абстракции, например, через интерфейсы, что делает всю систему более гибкой, более адаптируемой и расширяемой.

```
services.AddTransient<IDepartmentManager, DepartmentManager>();
```

Логика работы вне контроллера – реализовать интерфейс в отдельном классе, а в контроллере пользоваться только объектом интерфейса.

Внедрение зависимостей происходит в классе Startup в методе Configure.Services – в этом методе происходят связка нашего менеджера с интерфейсом, который реализует наш менеджер.

### 34. Введение в Базы Данных. Реляционные базы данных. Построение диаграммы «Сущность-связь» (Entity-Relationship Diagram). Виды связей между сущностями.

База данных – организованная коллекция данных, которая обычно хранится в компьютерной системе и доступна тоже оттуда. Когда базы данных более сложные, они часто обрабатываются с использованием специальных формальных методов проектирования и моделирования. Путаются понятия «база данных» и «СУБД».

СУБД – такое программное обеспечение, которое взаимодействует с конечными пользователями, с какими-то приложениями или с самой базой данных для сбора и анализа данных. Сама база данных – грубо говоря какая-то структура, которая содержит набор определенных таблиц, связи между этими таблицами и правила взаимодействия. СУБД предоставляет возможность обращаться к базе данных и проводить определенные действия. Основные действия, которые могут производиться с базой данных:

С – создавать (create)

R – считывать (read)

U-изменять (update)

D- удалять (delete)

---

Контекст – часть базы данных, которой мы будем пользоваться.

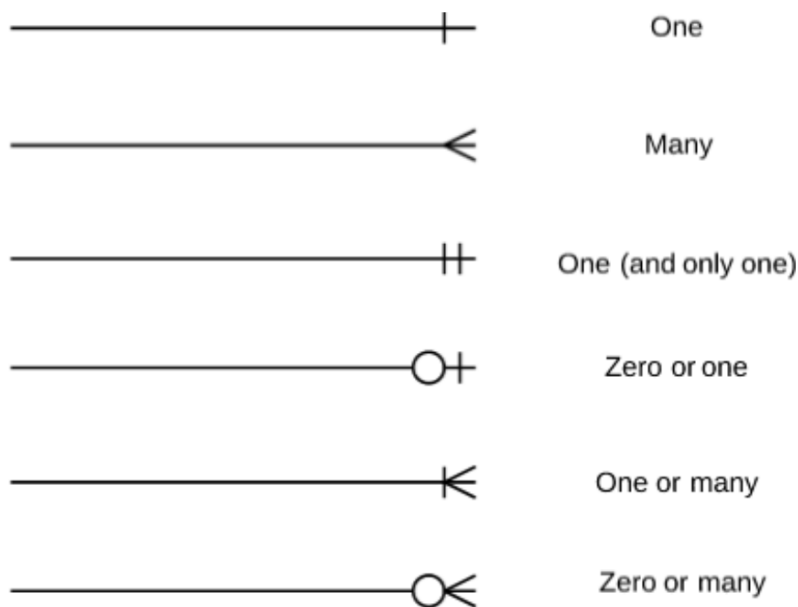
---

**Реляционная база данных** — это совокупность взаимосвязанных таблиц, каждая из которых содержит информацию об объектах определенного типа

---

Схема «сущность-связь» (также ERD или ER-диаграмма) — это разновидность блок-схемы, где показано, как разные «сущности» (люди, объекты, концепции и так далее) связаны между собой внутри системы. ER-диаграммы чаще всего применяются для проектирования и отладки реляционных баз данных в сфере образования, исследования и разработки программного обеспечения и информационных систем для бизнеса.

---



(One to one – был класс студент, решили разбить таблицу студент (имя и фамилия) на две таблицы. Одна фамилия соответствует одному имени.

Zero or many – один заказ может содержать несколько товаров, но может и вообще их не иметь)

### 35. Фреймворк Entity Framework / EF Core. Описание. Подходы к проектированию.

**Entity Framework** - технология для работы с данными.

Технология Entity Framework – кроссплатформенное приложение, которое позволяет обеспечивать доступ к данным через определенную СУБД.

Используем версию Entity Framework Core, которая уже имеет в своем наборе доп библиотеки для ASP.NET Core.

В нашем программном коде мы не будем писать запросы к базе данных (CRUD), а эта технология позволит нам сформировать запросы самостоятельно, а нам нужно будет лишь пользоваться основными средствами C#.

EF поддерживает несколько СУБД.

Стандартно поддерживает Microsoft SQL Server.

Центральной концепцией Entity Framework является понятие сущности.

Сущность представляет набор данных, ассоциированных с определенным объектом. Поэтому данная технология предполагает работу не с таблицами, а с объектами и их наборами.

Отличительной чертой Entity Framework является использование запросов LINQ для выборки данных из БД.

В Tools содержится специальная технология, которая позволяет проводить миграции.

Миграции – технология, которая позволяет обновлять базу данных при помощи командной строки нугет пакетов. Через JSON мы можем передавать определенные объекты, которые хранятся в спец формате который поддерживает JS.

---

Entity Framework предполагает три возможных способа взаимодействия с базой данных:

- **Database first:** Entity Framework создает набор классов, которые отражают модель конкретной базы данных
- **Model first:** сначала разработчик создает модель базы данных, по которой затем Entity Framework создает реальную базу данных на сервере.
- **Code first:** разработчик создает класс модели данных, которые будут храниться в бд, а затем Entity Framework по этой модели генерирует базу данных и ее таблицы

(Code First - сначала пишется код, а потом по нему создается база данных и ее таблицы.

Model First - сначала делается модель, а потом по ней создается база данных.

Database First - во многом похож на Model First и подходит для тех случаев, когда разработчик уже имеет готовую базу данных.)

### 36. Модель (Model) в паттерне MVC. Создание сущностей Базы Данных. Определение Context.

**Модель** - это данные и правила, которые используются для работы с данными, которые представляют концепцию управления приложением. В любом приложении вся структура моделируется как данные, которые обрабатываются определённым образом. Пользователь для приложения -

данные, которые должны быть обработаны в соответствии с правилами. Модель реагирует на команды контроллера, изменяя своё состояние. Модель — этот компонент отвечает за данные, а также определяет структуру приложения.

---

*Атрибут* описывает данные о сущности, которые нужно сохранить. У каждой сущности ноль или более атрибутов, описывающих ее, и каждый атрибут описывает в точности одну сущность.

Мы работаем по третьей нормальной форме. У каждой сущности должен быть некий id. Он может быть как int, так и специальным типом данных (guid).

Тип данных Guid – уникальный идентификатор, который представляет из себя набор шестнадцатеричных значений, которое будет уникальным полем, по которому мы сможем связывать наши объекты в случае необходимости для создания зависимостей. (Связать студент с предметом или что-то такое)

Key – атрибут, который позволяет сказать, что id будет уникальным ключом для данного поля.

Required: данный атрибут указывает, что свойство должно быть обязательно установлено, обязательно должно иметь какое-либо значение.

Шаги:

1. Выявить и смоделировать сущности.
2. Выявить и смоделировать связи между сущностями.
3. Выявить и смоделировать атрибуты.
4. Указать уникальный идентификатор для каждой сущности.
5. Провести нормализацию.

Говорят, что сущность находится в первой нормальной форме, когда все ее атрибуты имеют единственное значение.

Говорят, что сущность находится во второй нормальной форме, если она уже находится в первой НФ, и каждый неидентифицирующий атрибут зависит от всего уникального идентификатора сущности.

Сущность находится в третьей нормальной форме, если она уже находится во второй нормальной форме и ни один неидентифицирующий атрибут не зависит от каких-либо других неидентифицирующих атрибутов.

Нормализация – приведение сущности к какому-либо виду, описанному выше.



```

[Table("tblWorkers")]
Ссылка: 33
public class Worker : IUniqueIdentifiable
{
    [Required]
    [Column("Id")]
    Ссылка: 68
    public Guid Id { get; set; }

    [Required]
    [Column("Name")]
    [MaxLength(50)]
    Ссылка: 8
    public string Name { get; set; }

    [Required]
    [Column("Last Name")]
    [MaxLength(50)]
    Ссылка: 7
    public string LastName { get; set; }

    [Required]
    [Column("Date of employment")] //дата приема на работу
    Ссылка: 3
    public string DateOfEmployment { get; set; }

    [Required]
    [Column("PositionId")]
    Ссылка: 3
    public Guid PositionId { get; set; } // связываем работника с должностью
    [ForeignKey(nameof(PositionId))]
    Ссылка: 25
    public Position Position { get; set; }
}

```

Чтобы взаимодействовать с базой данных через Entity Framework нам нужен контекст данных - класс, унаследованный от класса **Microsoft.EntityFrameworkCore.DbContext**.

Каскадное удаление убирается в миграциях (в кратком описании

OnDelete:ReferentialAction.NoAction вместо

OnDelete:ReferentialAction.Cascade)

Контекст - класс, который будет использоваться для подключения к базе данных

```

namespace OOP_Лабораторная_4_5.Storage
{
    Ссылка: 23
    public class PersonnelDepartmentDataContext : DbContext
    {
        Ссылка: 0
        public PersonnelDepartmentDataContext(DbContextOptions<PersonnelDepartmentDataContext> options) : base(options)
        {
        }

        Ссылка: 12
        public DbSet<Worker> Worker { get; set; }
        Ссылка: 8
        public DbSet<Position> Position { get; set; }
        Ссылка: 8
        public DbSet<Department> Department { get; set; }
        Ссылка: 5
    }
}

```



Свойство DbSet представляет собой коллекцию объектов, которая сопоставляется с определенной таблицей в базе данных. При этом по умолчанию название свойства должно соответствовать множественному числу названию модели в соответствии с правилами английского языка. То есть User - название класса модели представляет единственное число, а Users - множественное число.

```
public class DepartmentManager : IDepartmentManager
{
    private readonly PersonnelDepartmentDataContext _dbContext;

    Ссылка: 0
    public DepartmentManager(PersonnelDepartmentDataContext dbContext)
    {
        _dbContext = dbContext; //получаем контекст
    }
}
```

Для того, чтобы «доставать» данные мы в лабораторной в менеджере создавали объект класса контекста и уже работали с ним с помощью LINQ (доставали всякую информацию из базы данных, фильтровали её)

### 37. LINQ to Entity. Трансляция в SQL-выражение.

LINQ to Entities предлагает простой и интуитивно понятный подход для получения данных с помощью выражений, которые по форме близки выражениям языка SQL

Хотя при работе с базой данных мы оперируем запросами LINQ, но база данных понимает только запросы на языке SQL. Поэтому между LINQ to Entities и базой данных есть проводник, который позволяет им взаимодействовать. Этим проводником является провайдер EntityClient.

#### Пример

В следующем примере метод [Select](#) используется для возврата всех строк из таблицы Product и отображения названий продуктов.

```
C# Копировать

using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    IQueryable<Product> productsQuery = from product in context.Products
                                        select product;

    Console.WriteLine("Product Names:");
    foreach (var prod in productsQuery)
    {
        Console.WriteLine(prod.Name);
    }
}
```

```
Ссылка: 2
public async Task<IReadOnlyCollection<Department>> GetAll() //преобразовываем в список, чтобы вывести
{
    var query = _dbContext.Department.AsNoTracking();
    var entities = await query.ToListAsync();
    return entities;
}
```

```
Ссылка: 2
public async Task<Department> RemoveDepartment(Guid id, CreateOrUpdateDepartment request)
{
    var entity = await _dbContext.Department.FirstOrDefault(g => g.Id == id);
    _dbContext.Remove(entity);
    await _dbContext.SaveChangesAsync();
    return entity;
}
```

```
Ссылка: 2
public async Task<IReadOnlyCollection<Department>> Search(string Name)
{
    var query = await _dbContext.Department.Where(p => EF.Functions.Like(p.Name, Name))
        .ToListAsync();

    return query;
}
```

---

Одной из целей LINQ является разрешение разработчикам программировать на своем родном языке программирования. Помните, что LINQ — это язык интегрированных запросов.