**1.**

**► Что такое пространство имён?**

Обычно определяемые классы и другие типы в .NET не существуют сами по себе, а заключаются в специальные контейнеры - **пространства имен**. Пространства имен позволяют организовать код программы в логические блоки, поволяют объединить и отделить от остального кода некоторую функциональность, которая связана некоторой общей идеей или которая выполняет определенную задачу.

Для определения пространства имен применяется ключевое слово namespace, после которого идет название название пространства имен:

namespace имя\_пространства\_имен

{

    // содержимое пространства имен

}

**Пространство имен** — это не более чем группа типов данных, но дающая тот эффект, что имена всех типов данных в пределах пространства имен автоматически снабжаются префиксом - названием пространства имен. Пространства имен можно вкладывать друг в друга.

**► Как подключить?**

Подключить пространство имен можно с помощью директивы **using**:

using System;

using System.Collections.Generic;

using System.Linq;

**► Пространства имен уровня файла?**

Объявления пространств имен с областью действия файла позволяют объявить, что *все типы в файле находятся в одном пространстве имен*. Объявления пространств имен с областью действия файла доступны с C# 10.

Добавим в проект класс, где напишем:

namespace Base;

class Person

{

    string name;

    public Person(string name) => this.name = name;

    public void Print() => Console.WriteLine($"Имя: {name} ");

}

Первой строчкой мы задаем пространство имен уровня файла, как видно дальнейший код не берется в блок, а просто ставится точка с запятой после объявления пространства.

Директива namespace Base в начале файла указывает, что содержимое файла будет представлять пространство имен Base.

Теперь подключим это пространство имен в файле Program.cs:

using Base; // подключение пространства имен Base

Person tom = new("Tom");

tom.Print();

**2.**

**► Область видимости.**

**Область видимости**, или контекст переменной — это часть кода, в пределах которого доступна данная переменная.

**► Какие контексты переменных существуют?**

▪ **Контекст класса**. Переменные, определенные на уровне класса, доступны в любом методе этого класса. Их еще называют глобальными переменными или полями.

▪ **Контекст метода**. Переменные, определенные на уровне метода, являются локальными и доступны только в рамках данного метода. В других методах они недоступны.

▪ **Контекст блока кода**. Переменные, определенные на уровне блока кода, также являются локальными и доступны только в рамках данного блока. Вне своего блока кода они не доступны. **3.**

**► Все типы можно разделить на?**

**Ссылочные** и **значимые**.

*Значимые типы хранятся в стеке, а ссылочные в куче.*

*Кучу* можно представить как неупорядоченный набор разнородных объектов.

При создании объекта ссылочного типа *в кучу помещаются сами данные*, а *в стеке ─ ссылка на адрес*, под которым они в этой куче хранятся.

**► Какие типы относятся к ссылочным?**

Ссылочные типы:

• Тип object

• Тип string

• Классы (class)

• Интерфейсы (interface)

• Делегаты (delegate)  
**4.**

**► Что такое многомерный массив?**

Многомерный массив – это массив, который имеет два и более измерений. Для доступа к элементу многомерного массива используется комбинация из двух и более индексов.

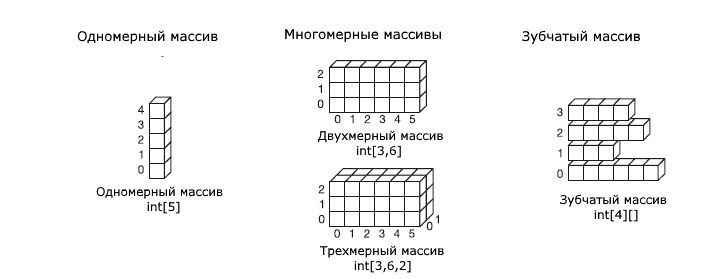
Общая форма объявления многомерного массива следующая:

*тип*[,...,] *имя\_массива* = new *тип*[*размер1*, *размер2*, ..., *размерN*];

где

* *тип* – непосредственно тип элементов массива;
* *размер1*, *размер2*, *размерN* – размерность каждого измерения массива. Значение N означает, что массив N-мерный.

*Двумерный массив* ─ массив массивов одинакового размера.



**► Чем отличается от ступенчатого?**

**Ступенчатый массив** представляет собой массив массивов, в котором *длина каждого массива может быть разной*.

Ступенчатые массивы объявляются с помощью ряда квадратных скобок, в которых указывается их размерность. Например, для объявления двумерного ступенчатого массива служит следующая общая форма:

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

**5.**

**► Что такое класс?**

**Класс** представляет собой *шаблон, по которому определяется форма объекта*. В нем указываются данные и код, который будет оперировать этими данными. В C# используется спецификация класса для построения объектов, которые являются экземплярами класса. Следовательно, класс, по существу, представляет собой ряд схематических описаний способа построения объекта. При этом очень важно подчеркнуть, что класс является логической абстракцией. Физическое представление класса появится в оперативной памяти лишь после того, как будет создан объект этого класса.

Классы и структуры — это, по сути, шаблоны, по которым можно создавать объекты. Каждый объект содержит данные и методы, манипулирующие этими данными.

**► К какому типу относится?**

К ссылочному.

**► Какие классы бывают?**

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

**→ Частичные классы** объявляются с модификатором partial и предоставляют возможность иметь несколько файлов с определением одного и того же класса, и при компиляции все эти определения будут скомпилированы в одно.

**→ Абстрактные** объявляются с модификатором abstract и является шаблоном для других классов, при это *объекты абстрактного класса создавать нельзя*, от такого класса только могут наследовать другие классы.

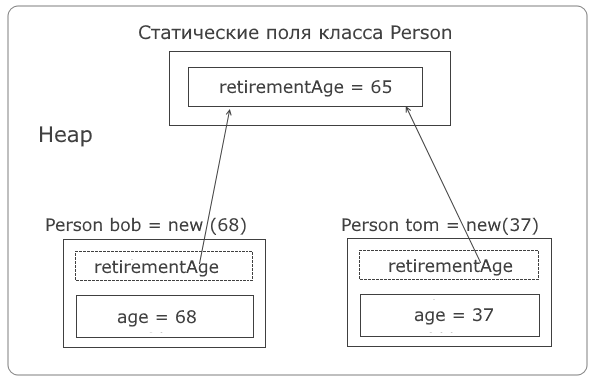
Например, в реальности не существует геометрической фигуры как таковой. Есть круг, прямоугольник, квадрат, но просто фигуры нет. Однако же и круг, и прямоугольник имеют что-то общее и являются фигурами. И для описания подобных сущностей, которые не имеют конкретного воплощения, предназначены абстрактные классы.

Класс, содержащий один или больше абстрактных методов, должен быть также объявлен как абстрактный.

**6.**

**► Как работают статические поля?**

Статические поля хранят состояние всего класса / структуры. Статическое поле определяется как и обычное, только перед типом поля указывается ключевое слово static. На уровне памяти для статических полей будет создаваться участок в памяти, который будет общим для всех объектов класса.



**► Как вызвать?**

Для обращения к таким членам класса необязательно использовать объект.

То есть вместо: *obj\_name.field\_name*, можно писать: *class\_name.field\_name*. **7.**

**► Что такое конструктор?**

Кроме обычных методов в классах используются также и специальные методы, которые называются конструкторами. **Конструкторы** вызываются при создании нового объекта данного класса.

Имя конструктора совпадает с его типом, то есть с именем класса, в котором он создается.

**► Для чего нужен?**

Конструкторы выполняют инициализацию объекта.

**► Какие конструкторы бывают?**

**→ По умолчанию**

Это конструктор дефолтно существующий во всех классах. При необходимости *может быть переопределен*. По определению такой конструктор *никогда не принимает аргументов*. После размещения нового объекта в памяти конструктор по умолчанию *гарантирует установку всех полей в соответствующие стандартные значения* (0 для int, null для string и т.д.).

**→ Закрытые**

Закрытый конструктор — это особый конструктор экземпляров. Такой конструктор объявляется с модификатором **private** и *запрещает определенный шаблон создания объектов*.

Если в классе один или несколько закрытых конструкторов и ни одного открытого конструктора, то прочие классы (за исключением вложенных классов) не смогут создавать экземпляры этого класса.

Пример:

class NLog

{

// Private Constructor:

private NLog() { }

public static double e = Math.E; //2.71828...

}

В данном примере объявление пустого конструктора закрытым запрещает автоматическое создание конструктора без параметров. Обратите внимание, что если не использовать с конструктором модификатор доступа, то по умолчанию он все равно будет закрытым. Однако обычно используется модификатор private, чтобы явно обозначить невозможность создания экземпляров этого класса.

**→ Статические**

*Статические конструкторы объявляются с ключевым словом* ***static*** *имеют следующие отличительные черты*:

→ Статические конструкторы *не должны иметь модификатор доступа* и *не принимают параметров*

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

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

→ Статические конструкторы обычно используются для инициализации статических данных, либо же *выполняют действия, которые требуется выполнить только один раз*

Определим статический конструктор:

Console.WriteLine(Person.RetirementAge);

class Person

{

static int retirementAge;

public static int RetirementAge => retirementAge;

static Person()

{

if (DateTime.Now.Year == 2022)

retirementAge = 65;

else

retirementAge = 67;

}

}

В данном случае с помощью встроенной структуры DateTime получаем текущий год. Для этого используется свойство DateTime.Now.Year. если он равен 2022, устанавливаем один пенсионный возраст. При другом значении текущего года устанавливается другое значение пенсионного возраста.  
**8.**

**► Что такое свойства в классе?**

Кроме обычных методов в языке C# предусмотрены специальные методы доступа, которые называют **свойства**. Они обеспечивают простой доступ к полям классов и структур, позволяют узнать их значение или выполнить их установку.

**► Как создать?**

Стандартное описание свойства имеет следующий синтаксис:

[модификаторы] тип\_свойства название\_свойства

{

get { действия, выполняемые при *получении* значения свойства}

set { действия, выполняемые при *установке* значения свойства}

}

Вначале определения свойства могут идти различные модификаторы, в частности, модификаторы доступа. Затем указывается тип свойства, после которого идет название свойства. Полное определение свойства содержит два блока: get и set.

В блоке **get** выполняются действия по получению значения свойства. В этом блоке с помощью оператора return возвращаем некоторое значение.

В блоке **set** устанавливается значение свойства. В этом блоке с помощью параметра value мы можем получить значение, которое передано свойству.

Блоки get и set еще называются **акссесорами** или методами доступа (к значению свойства), а также геттером и сеттером.

То есть, по сути, свойство ничего не хранит, оно выступает в роли посредника между внешним кодом и переменной.

Автоматические свойства имеют сокращенное объявление:

[модификаторы] тип\_свойства название\_свойства{ get; set; }

**9.**

**► Как выглядит общее определение метода?**

[модификаторы] тип\_возвращаемого\_значения название\_метода ([параметры (по шаблону: тип название)])

{

// тело метода

}

Модификаторы и параметры необязательны.

**► Как вызывать метод?**

название\_метода (значения\_для\_параметров\_метода);

Примеры:

//методы

void SayHello() {}

int Sum (int a, float b) {}

//вызовы

SayHello();

Sum(5, 2f);

Поскольку первый метод не принимает никаких параметров, то после названия метода идут пустые скобки.

**10.**

**► Какие методы бывают?**

→ **Статические методы** объявляются со словом static и определяют общее для всех объектов поведение, которое не зависит от конкретного объекта. Для обращения к статическим методам также применяется имя класса / структуры:

Person bob = new(68);

Person.CheckRetirementStatus(bob);

class Person

{

public int Age { get; set; }

static int retirementAge = 65;

public Person(int age) => Age = age;

public static void CheckRetirementStatus(Person person)

{

if (person.Age >= retirementAge)

Console.WriteLine("Уже на пенсии");

else

Console.WriteLine($"Сколько лет осталось до пенсии: {retirementAge - person.Age}") ;

}

}

→ **Абстрактные методы** – методы, объявляемые со словом abstract в абстрактном классе, не содержащие тела, а лишь сигнатуру.

abstract class Transport

{

public abstract void Move();

}

// класс корабля

class Ship : Transport

{

// мы должны реализовать все абстрактные методы и свойства базового класса

public override void Move()

{

Console.WriteLine("Корабль плывет");

}

}

→ **Виртуальные методы** – это методы, объявляемые с ключевым словом virtual и доступные для переопределения в классах наследниках.

А чтобы переопределить метод в классе-наследнике, этот метод определяется с модификатором override. Переопределенный метод в классе-наследнике должен иметь тот же набор параметров, что и виртуальный метод в базовом классе.

**class Person**

{

public string Name { get; set; }

public Person(string name)

{

Name = name;

}

public **virtual** void Print()

{

Console.WriteLine(Name);

}

}

**class Employee : Person**

{

public string Company { get; set; }

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

{

Company = company;

}

public **override** void Print()

{

Console.WriteLine($"{Name} работает в {Company}");

}

}

→ **Частичные методы** определяются с ключевым словом **partial** в частичном классе. Причем *определение частичного метода без тела метода находится в одном частичном классе*, а *реализация* этого же метода – *в другом* частичном классе.

Например:

Часть частичного метода в одном файле:

public partial class Person

{

partial void Read();

}

В другом (с реализацией метода):

public partial class Person

{

partial void Read()

{

Console.WriteLine("I am reading a book");

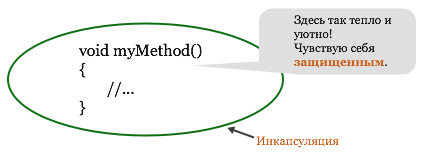
}

}

**► Где могут быть созданы (каждый из видов) и вызваны?  
11.**

**► Что такое структура?**

**Тип структуры** представляет собой *тип значения*, который может инкапсулировать данные и связанные функции (то есть объединять их в блок кода). Для определения типа структуры используется ключевое слово **struct**.



**► Отличия от класса? (поля, конструктор, наследование, методы)**

→ *Не поддерживают* ***наследование*** (не могут наследовать и не могут сами быть наследуемыми).

→ Как и класс, структура *может определять* ***конструктор***.

Но в отличие от класса нам *не обязательно вызывать конструктор для создания объекта структуры*. Однако если мы создаем объект структуры таким образом, то обязательно надо проинициализировать все поля (глобальные переменные) структуры перед получением их значений или перед вызовом методов структуры.

Также мы можем использовать для создания структуры конструктор без параметров, который есть в структуре по умолчанию и при вызове которого полям структуры будет присвоено значение по умолчанию (например, для числовых типов это число 0):

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

Также *мы можем определить свои конструкторы*. Важно учитывать, что **если мы определяем конструктор в структуре, то он должен инициализировать все поля структуры**.

struct User

{

public string *name*;

public int *age*;

public User(string name, int age)

{

*this.name* = name;

*this.age* = age;

}

public void DisplayInfo()

{

Console.WriteLine($"Name: {name} Age: {age}");

}

Вы *не можете объявить конструктор без параметров* в версиях C# до 10. Каждый тип структуры уже предоставляет неявный конструктор без параметров, который создает значение по умолчанию типа.

Примечание

Начиная с C# версии 10 вы можете объявить конструктор без параметров в типе структуры.

public readonly struct Measurement

{

public Measurement()

{

Value = double.NaN;

Description = "Undefined";

}

public Measurement(double value, string description)

{

Value = value;

Description = description;

}

public double Value { get; init; }

public string Description { get; init; }

}

→ *Нельзя инициализировать* ***поля структуры*** *напрямую при их объявлении*, например, следующим образом:

struct User

{

public string name = "Sam"; // ! Ошибка

public int age = 23; // ! Ошибка

public void DisplayInfo()

{

Console.WriteLine($"Name: {name} Age: {age}");

}

}

Однако *можно инициализировать статическое поле, или поле константы, или статическое свойство при его объявлении*.

Примечание

Начиная с C# версии 10 вы можете инициализировать поле экземпляра или свойство при его объявлении.

struct Person

{

// инициализация полей значениями по умолчанию - доступна с C#10

public string name = "Tom";

public int age = 1;

public void Print() =>Console.WriteLine($"Имя: {name} Возраст: {age}");

}

Однако даже в этом случае, несмотря на значения по умолчанию, необходимо вызывать конструктор, если мы хотим из использовать.