

ЛЕКЦИЯ 2

- Модуль 2
- 01.11.2023
- Инкапсуляция
- Структура класса

ЦЕЛИ ЛЕКЦИИ

- Разобраться с реализацией класса и его основных членов в языке C#
- Познакомится с объектной инкапсуляцией



Это изображение, автор: Неизвестный автор, лицензия: CC BY-NC

ПРОГРАММА СЧИТАЕТСЯ ОБЪЕКТНОЙ, ЕСЛИ

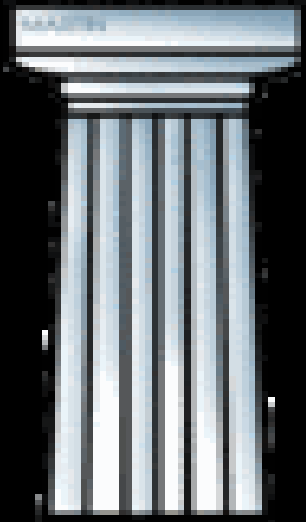
- использует в качестве основных конструктивных элементов объекты, а не алгоритмы
- каждый объект является экземпляром определённого класса
- классы образуют иерархии

Г. Буч, Р.А. Максимчук, М.У. Энгл, Б.Д. Янг, Д. Коналлен, К.А. Хьюстон Объектно-ориентированный анализ и проектирование с примерами приложений. М.: Издательский дом «Вильямс», 2008.

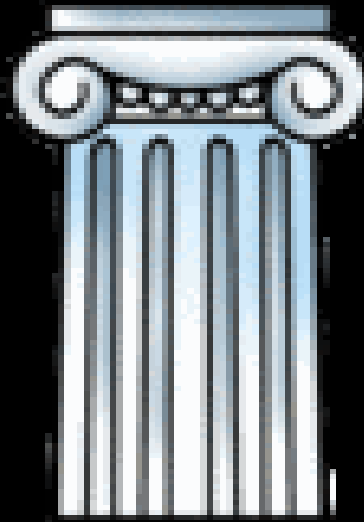
Далее будем подробно разбираться в основных понятиях ООП и знакомиться с реализацией в C#

ТРИ БАЗОВЫХ ПРИНЦИПА ООП

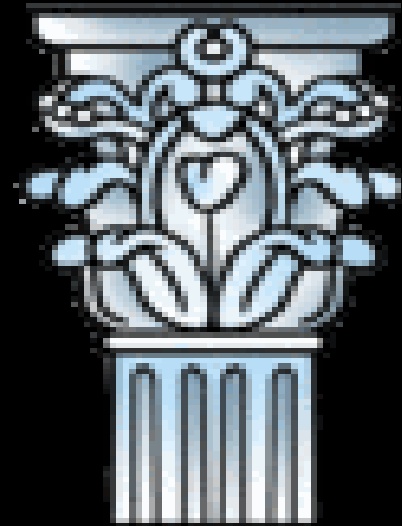
полиморфизм



наследование



инкапсуляция



ИНКАПСУЛЯЦИЯ

- **Инкапсуляция** – сокрытие деталей реализации
- **Объектовая инкапсуляция:**
 - Объединяет инкапсуляцию данных и инкапсуляцию процессов
 - Основана на описании состояния и поведения объекта на уровне интерфейса общего шаблона объектов с идентичным поведением – класса
 - Позволяет управлять видимостью элементов интерфейса
 - Позволяет единообразно сформулировать понятие наследования

В C# у классов и их членов может быть установлен уровень доступа

СПОСОБЫ ОГРАНИЧЕНИЯ ДОСТУПА К ДАННЫМ В C#

Видимость

public, protected internal, protected, internal, private protected, private

Доступ на чтение и запись

readonly, const

Регламент доступа к данным

методы, аксессоры свойств, индексаторы, события

МОДИФИКАТОРЫ ДОСТУПА

Место нахождения вызывающего кода	public	protected internal	protected	internal	private protected	private
Тот же класс	+	+	+	+	+	+
Класс наследник той же сборки	+	+	+	+	+	
Класс не наследник той же сборки	+	+		+		
Класс наследник другой сборки	+	+	+			
Класс не наследник другой сборки	+					

Не строго: сборка – это .dll или .exe, полученный за одну компиляцию одного или нескольких исходных файлов

СИНТАКСИС МОДИФИКАТОРОВ ПРИ ЧЛЕНАХ КЛАССОВ C#

Поля

<модификатор доступа> тип идентификатор

Методы

<модификатор доступа> тип Идентификатор ()

{

...

}

```
class AccessDemo
{
    int F1;           // неявно закрытое поле.
    private int F2;   // явно закрытое поле.
    public int F3;    // открытое поле.

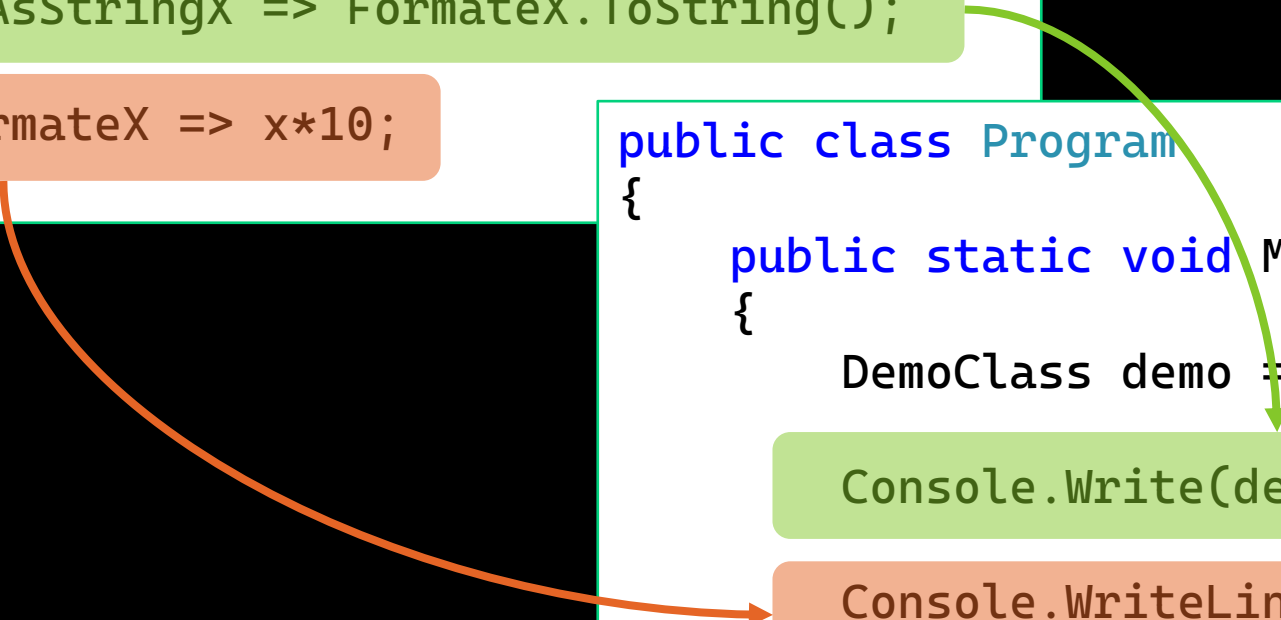
    void DoCalc() // неявно закрытый метод.
    {
        //...
    }

    public int GetVal() // открытый метод.
    {
        //...
    }
}
```


ОТКРЫТЫЕ И ЗАКРЫТЫЕ ЧЛЕНЫ

```
public class DemoClass
{
    int x;
    public string AsStringX => FormateX.ToString();
    private int FormateX => x*10;
}
```

```
public class Program
{
    public static void Main()
    {
        DemoClass demo = new DemoClass();
        Console.Write(demo.AsStringX);
        Console.WriteLine(demo.FormateX);
    }
}
```



ЧТО ВХОДИТ В ТИПЫ ДАННЫХ?

Типы данных содержат в себе функциональные члены и данные

Данные	Функциональные Члены	
Поля	Методы	Операции
Константы	Свойства	Индексаторы
	Конструкторы	События
	Деструкторы (Финализаторы)	

На уровне IL существуют только методы и поля! Остальные конструкции фактически существуют только для упрощения работы программиста

ПОЛЯ



ПОЛЯ

Поля – переменные, определённые на уровне типов

Поля никогда не бывают
неинициализированными, здесь
использовано значение по умолчанию
для double

```
class Point
{
    double _x;
    double _y;

    public override string ToString() => $"({_x:f3}; {_y:f3})";
}
```

Область видимости полей –
весь тип

ИНИЦИАЛИЗАЦИЯ ПОЛЕЙ

Инициализация полей
значениями по умолчанию

```
class MyClass {  
    int F1;  
    string F2;  
    int F3 = 25;  
    string F4 = "abcd";  
    Random rnd = new Random();  
}
```

// Инициализируется 0 - тип значения.
// Инициализируется null - ссылочный тип.
// Инициализируется 25.
// Инициализируется "abcd".
// Инициализируется объектом Random.

Явная инициализация
полей

<тип> <имя поля> = <инициализирующее выражение>;

МЕТОДЫ



МЕТОДЫ КЛАССОВ

Методы – один из основных способов определения поведения типов

Методы в C# не могут объявляться вне типов данных

Минимальный синтаксис объявления методов включает:

- Идентификатор метода
- Тип возвращаемого значения
- Список параметров в круглых скобках
- Тело – блок операторов

СТАТИЧЕСКИЕ И ЭКЗЕМПЛЯРНЫЕ МЕТОДЫ: ПРИМЕР

Класс с экземплярным методом

```
public class Calculator
{
    public int Add(int a, int b) => a + b;
}
```

Класс со статическим методом

```
public static class CalculatorAsStatic
{
    public static int Add(int a, int b) => a + b;
}
```

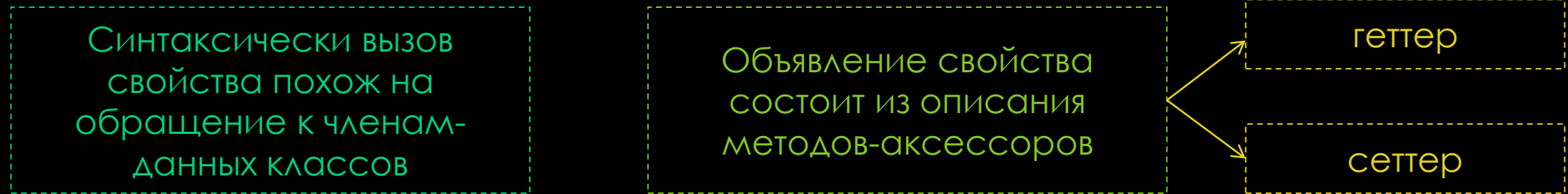
```
class Program
{
    static void Main(string[] args)
    {
        int valueOfA = 10, valueOfB = 20;
        Calculator calculator = new Calculator();
        Console.WriteLine($"{calculator.Add(valueOfA, valueOfB)} ");
        Console.WriteLine($"{CalculatorAsStatic.Add(valueOfA, valueOfB)}");
    }
}
```

СВОЙСТВА



СВОЙСТВА

- **Свойства** – специальный механизм для чтения, записи и вычисления значений закрытых (`private`) полей



```
[Модификаторы] <Тип> <Имя свойства> { get{[Тело]} set{[Тело]} }
```

ГЕТТЕРЫ

```
public class Point
{
    double _x;
    double _y;

    public double X { get { return _x; } }
    public double Y { get { return _y; } }
}
```

get-аксессор
используется для
возврата значения

get обязательно
указывается, чтобы
отметить, что это
описание геттера

При вызове **не указывают явно get:**
`p0.getX()` или `p0.get`

Такой геттер реализует вариант
доступа «только на чтение»

Обращение к геттерам
«как к членам-данным»

```
Point p0 = new Point();
Console.WriteLine($"{p0.X}; {p0.Y}");
```

СЕТТЕРЫ

```
public class Point
```

```
{
```

```
    double _x;  
    double _y;
```

set-аксессор используется для назначения значения скрытому полю

```
    public double X { get { return _x; } set { _x = value; } }  
    public double Y { get { return _y; } set { _y = value; } }  
}
```

Ключевое слово **value** используется для обозначения значения, которое передаётся в свойство для назначения

Пара «getter-сеттер» реализует вариант доступа «чтение и запись»

set обязательно указывается, чтобы отметить, что это описание сеттера

Обращение к сеттерам «как к членам-данных», используем как **l-value**

```
Point p0 = new Point();
```

```
p0.X = 1;
```

```
p0.Y = 2;
```

value в свойстве

```
Console.WriteLine($"{p0.X}; {p0.Y}");
```


ГЕТТЕР ДЛЯ ОДНОКРАТНОГО НАЗНАЧЕНИЯ

```
public class Point
{
    double _x;
    double _y;

    public double X { get { return _x; } }
    public double Y { get { return _y; } }
}
```

init-аксессор используется для назначения значения скрытому полю значения только при создании объекта

```
public double X { get { return _x; } } init { _x = value; } }
public double Y { get { return _y; } } init { _y = value; } }
```

```
Point p0 = new Point();
```

```
p0.X = 1;
```

```
p0.Y = 2;
```

Недопустимо переназначение

```
Console.WriteLine($"{p0.X}; {p0.Y}");
```

В конструктор
ДОПУСТИМО

```
public Point() { }
public Point(double x, double y) => (X, Y) = (x, y);
```

СВОЙСТВО В СИНТАКСИСЕ ВЫРАЖЕНИЯ

```
public class Point
{
```

```
    double _x;
    double _y;
```

```
public double X { get { return _x; } set { _x = value; } }
public double Y { get { return _y; } set { _y = value; } }
```

```
public double X
{
    get => _x;
    set => _x = value;
}
public double Y
{
    get => _y;
    set => _y = value;
}
```

```
public Point() { }
public Point(double x, double y) => (X, Y) = (x, y);
}
```

Свойства, состоящие из
одной строки (выражения)
допустимо упростить

Наши свойства сейчас
данные не валидируют

АВТОРЕАЛИЗУЕМЫЕ СВОЙСТВА

```
double _x;  
double _y;  
  
public double X  
{  
    get => _x;  
    set => _x = value;  
}  
  
public double Y  
{  
    get => _y;  
    set => _y = value;  
}
```

```
public class Point  
{  
    public double X { get; set; }  
    public double Y { get; set; }  
  
    public Point() { }  
    public Point(double x, double y) => (X, Y) = (x, y);  
}
```

Свойства, служащие только для назначения и получения значений (без проверок, вычислений, валидации и проч.) могут быть переписаны в синтаксисе автореализуемых свойств

компилятор сам создаст анонимные закрытые поля и позволит сохранять и получать значения этих полей только через `get` и `set` доступы свойства

*ОБЯЗАТЕЛЬНЫЕ СВОЙСТВА

С# 11 позволяет использовать обязательные (**required**) свойства

```
public required double X { get; set; }  
public required double Y { get; set; }
```

В тело свойства может быть добавлен код для верификации значений

```
public class Square
{
    double _a;
    public double A
    {
        get => _a;
        set => _a = (value > 0) ? value: throw new ArgumentOutOfRangeException();
    }
    public override string ToString() => $"{A}";
}
```

```
Square sq = new();
double a = 0;
do
{
    Console.WriteLine("Enter a side's value: ");
    double.TryParse(Console.ReadLine(), out a);
    try
    {
        sq.A = a;
        break;
    }
    catch (ArgumentOutOfRangeException ex)
    {
        Console.WriteLine(ex.Message);
    }
} while (true);
Console.WriteLine($"{sq.A}");
```

Свойства не обязательно связаны с закрытыми полями один к одному, они могут служить для возврата вычислений

```
Square square = new Square();  
square.A = 1.5;  
Console.WriteLine($"side{square.A}; square: {square.S:f3}");
```

```
public class Square  
{  
    double _a;  
    public double A  
    {  
        get => _a;  
        set => _a = (value > 0) ? value: throw new ArgumentOutOfRangeException();  
    }  
    public double S  
    {  
        get => _a * _a;  
    }  
    public override string ToString() => $"{A}";  
}
```


ИНДЕКСАТОР



ИНДЕКСАТОР

- **Индексатор** предоставляет геттеры и сеттеры для обеспечения контроля над чтением, записью и изменениями значений внутреннего массива

```
public class Data
{
    double[] _data = { 1.3, 1.8, 2.24, 3.75, 4.901 };
    public double this[int i]
    {
        get => (i > 0) ? _data[i] : throw new IndexOutOfRangeException() ;
    }
}
```

```
Data d = new();
Console.Write(d[1]);
```

СИНТАКСИС ИНДЕКСАТОРА

```
<модификатор доступа> <тип данных> this[int index]
{
    get
    {
        // Возвращает значение из массива, доступное по индексу index.
    }
    set
    {
        // Устанавливает значение массива по индексу index.
    }
}
```

ИНДЕКСАТОР ПО ПОЛЯМ КЛАССА

```
public class Point
{
    double _x;
    double _y;

    public double this[int index]
    {
        get => (index) switch
        {
            0 => this._x,
            1 => this._y,
            _ => throw new IndexOutOfRangeException()
        };
    }

    public Point() { }
    public Point(double x, double y) => (_x, _y) = (x, y);
}
```

```
Point p0 = new Point();
Console.WriteLine($"{p0[0]}, {p0[1]}");
```

Индексатор по полям класса, ставим индекс в соответствие полю


МНОГОМЕРНЫЕ ИНДЕКСАТОРЫ

```
public int this[int i, int j]
{
    get
    {
        // Логика геттера.
    }
}
```

Правилами о качестве C# кода крайне не рекомендуется описывать многомерные индексы

СТАТИЧЕСКИЕ И КОНСТАНТНЫЕ ЧЛЕНЫ КЛАССОВ

const
readonly
static



КОНСТАНТЫ

```
public class MyMath
{
    const double MyPi = 3.14;
    const double MyPi2 = 2 * MyPi;
}
const double MyPiDiv2 = MyPi / 2;
```

Константное выражение – это выражение, которое полностью вычисляется на этапе компиляции

Константы могут быть описаны на уровне типа или метода, но не пространства имён

В объявлении константы не допустим модификатор `static`

```
class Program
{
    static void Main(string[] args)
    {
        const double Pi = 3.14;
    }
}
```

СРАВНЕНИЕ КОНТАНТ И ПОЛЕЙ ДЛЯ ЧТЕНИЯ

	инициализация	допускают изменение во время работы программы	Время назначения	Допустимы разные значения при перезапусках программы
const	При объявлении	Нет	Компиляция (compile-time)	Нет
readonly	<ul style="list-style-type: none"> • При объявлении • В конструкторе 	Нет	Время исполнения (run-time)	Да

ПРИМЕР СТАТИЧЕСКИЕ ПОЛЯ

```
public class Dragon
{
    // Золото общее для всех объектов-драконов.
    static int s_Gold = 100;

    public static int GoldAmount
    {
        get => s_Gold;
    }
    public int PersonalGold
    {
        get => s_Gold;
    }
    public void SpendGold(int amount) => s_Gold -= amount;
}
```

```
Dragon dragon1 = new();
Dragon dragon2 = new();
Console.WriteLine($"Общее золото: {Dragon.GoldAmount}");
Console.WriteLine(dragon1.PersonalGold);
Console.WriteLine(dragon2.PersonalGold);
dragon1.SpendGold(80);
Console.WriteLine(dragon1.PersonalGold);
Console.WriteLine(dragon2.PersonalGold);
```

Драконы – твари жадные и каждый
считает всё золото своим

ПРИМЕР СТАТИЧЕСКИХ И ЭКЗЕМПЛЯРНЫХ ПОЛЕЙ

```
public class Dragon
{
    static int s_Gold = 100;
    int _holdGold = 0;
    public static int GoldAmount
    {
        get => s_Gold;
    }
    public int PersonalGold
    {
        get => _holdGold;
        set => _holdGold = value > 0 ? value : throw new ArgumentOutOfRangeException();
    }
    public void SpendGold(int amount) => s_Gold -= amount;
}
```

Личное золото
дракона

При получении золота дракон
прячет его к себе, а не в
общее золото

ИСПОЛЬЗОВАННАЯ ЛИТЕРАТУРА

- Г. Буч, Р.А. Максимчук, М.У. Энгл, Б.Д. Янг, Д. Коналлен, К.А. Хьюстон Объектно-ориентированный анализ и проектирование с примерами приложений. М.: Издательский дом «Вильямс», 2008.
- Модификаторы доступа (Справочник по C#)
(<https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/keywords/access-modifiers>)
- <https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/const>