ЛЕКЦИЯ 1-2

- Модуль 2
- 05.11.2024
- Основные концепции объектно-ориентированного программирования
- Инкапсуляция
- Структура класса

ЦЕЛИ ЛЕКЦИИ

- Познакомится с общеархитектурными принципами проектирования
- Познакомится с понятием объекта и класса
- Разобраться с созданием объектов

ГРАФИК КОНТРОЛЕЙ, МОДУЛЬ 2

Модуль 2									
	Форма контроля						Повторные сдачи		
Подгруппа	ЛР_2_1	ЛР_2_2	ЛР_2_3	П_2_1	Т_2	КР_2	П_2_1	Т_2	КР_2 для болевших
Максименкова (244-1)	07.ноя	14.ноя	28.ноя	11.11-22.11	03.дек	12.дек	03.12-14.12	10.дек	на неделе 16.12-20.12
Максименкова (244-2)	08.ноя	15.ноя	29.ноя	11.11-22.11	03.дек	13.дек	03.12-14.12	10.дек	на неделе 16.12-20.12
Бегичева (248-2)	06.ноя	13.ноя	27.ноя	11.11-22.11	03.дек	11.дек	03.12-14.12	10.дек	на неделе 16.12-20.12
Бегичева (248-1)	08.ноя	15.ноя	29.ноя	11.11-22.11	03.дек	13.дек	03.12-14.12	10.дек	на неделе 16.12-20.12
Резуник (247-1)	07.ноя	14.ноя	28.ноя	11.11-22.11	03.дек	12.дек	03.12-14.12	10.дек	на неделе 16.12-20.12
Резуник (247-2)	09.ноя	16.ноя	30.ноя	11.11-22.11	03.дек	14.дек	03.12-14.12	10.дек	на неделе 16.12-20.12
Лесовская (246-1)	07.ноя	14.ноя	28.ноя	11.11-22.11	03.дек	12.дек	03.12-14.12	10.дек	на неделе 16.12-20.12
Лесовская (245-1)	09.ноя	16.ноя	30.ноя	11.11-22.11	03.дек	14.дек	03.12-14.12	10.дек	на неделе 16.12-20.12
Лесовская (245-2)	09.ноя	16.ноя	30.ноя	11.11-22.11	03.дек	14.дек	03.12-14.12	10.дек	на неделе 16.12-20.12
Чапкин (2410-2)	06.ноя	13.ноя	27.ноя	11.11-22.11	03.дек	11.дек	03.12-14.12	10.дек	на неделе 16.12-20.12
Чапкин (2410-1)	07.ноя	14.ноя	28.ноя	11.11-22.11	03.дек	12.дек	03.12-14.12	10.дек	на неделе 16.12-20.12
Солодченко (249-1)	09.ноя	16.ноя	30.ноя	11.11-22.11	03.дек	14.дек	03.12-14.12	10.дек	на неделе 16.12-20.12
Солодченко (249-2)	09.ноя	16.ноя	30.ноя	11.11-22.11	03.дек	14.дек	03.12-14.12	10.дек	на неделе 16.12-20.12
Лесовская (246-2)	08.ноя	15.ноя	29.ноя	11.11-22.11	03.дек	13.дек	03.12-14.12	10.дек	на неделе 16.12-20.12

Темы контролей предварительные и могут варьироваться в зависимости от скорости подачи и освоения материала

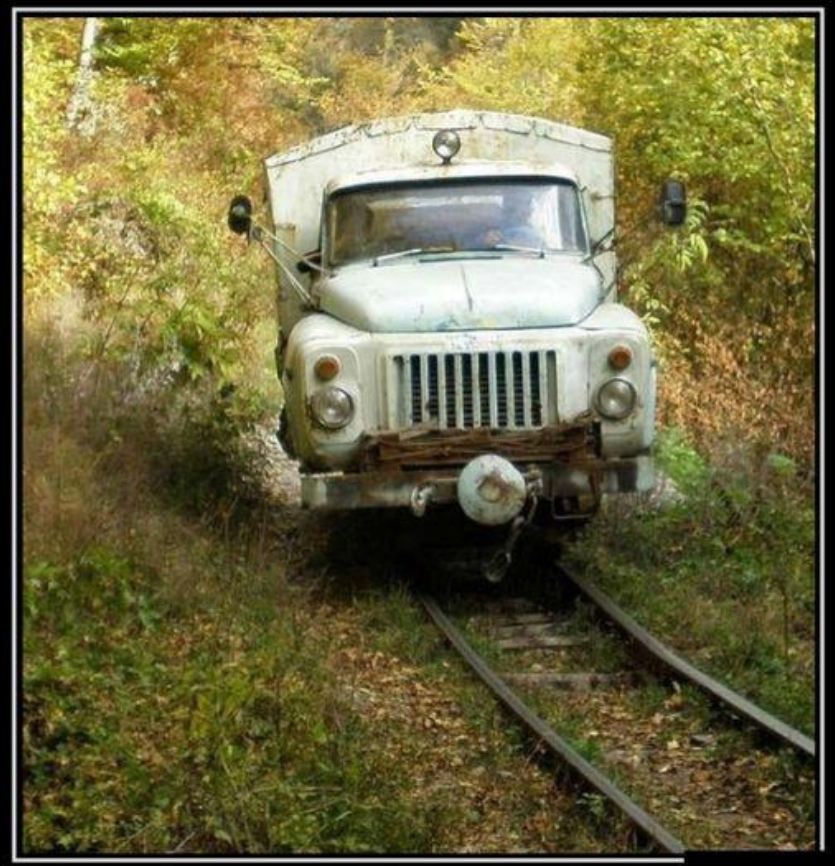
ОБЩЕАРХИТЕКТУРНЫЕ ПРИНЦИПЫ

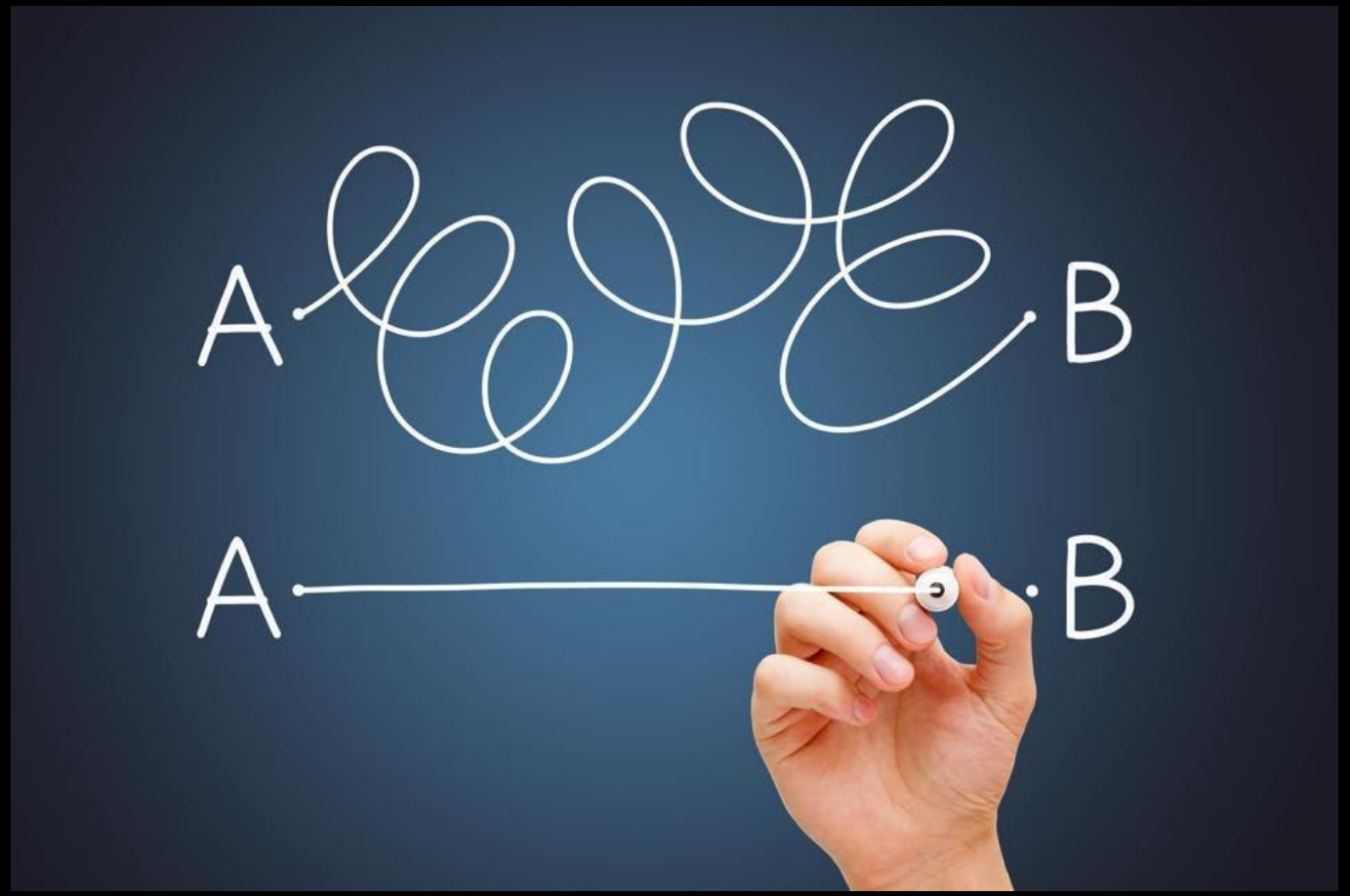
KISS, YAGNI, DRY

и

Бритва Оккама

KEEP IT SHORT AND SIMPLE





YOU AIN'T GONNA NEED IT

В хозяйстве пригодится

by Wall-ery

WRITE EVERYTHING TWICE OR WE ENJOY TYPING

- Stay **dry**, don't be **wet**

БРИТВА ОККАМА

Не следует множить сущее без необходимости

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

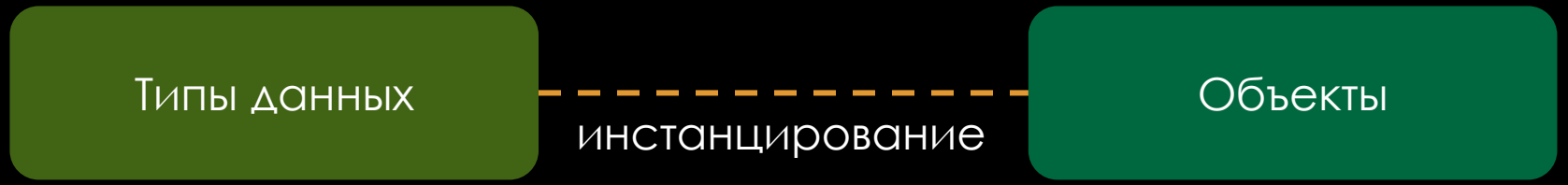
Основные определения

Базовые принципы

ОО-ОБЪЕКТНО-ОРИЕНТИРОВАННЫЙ

- **Объектно-ориентированный подход** [object-oriented approach] – подход к решению проблем, предполагающий объектовую декомпозицию проблемы
- **Объектно-ориентированное программирование** [object-oriented programming] – это парадигма программирования, основанная на представлении программы в виде совокупности взаимодействующих объектов, каждый из которых является экземпляром некоторого класса, входящего в иерархию наследования

ОБЪЕКТНО-ОРИЕНТИРОВАННАЯ ПАРАДИГМА В C#




```
ṣʧsîŋĝ ṣʧs   ŋêx ṣʧsîŋĝ  ǎ   ,_
```

```
îŋʧ   ǎs   ŋêx îŋʧ ¸
```

ПРЕИМУЩЕСТВА ИСПОЛЬЗОВАНИЯ ОО ПАРАДИГМЫ

Объекты достаточно естественны для восприятия человеком

Объекты могут содержать внутренне устройство, закрытое от внешнего мира

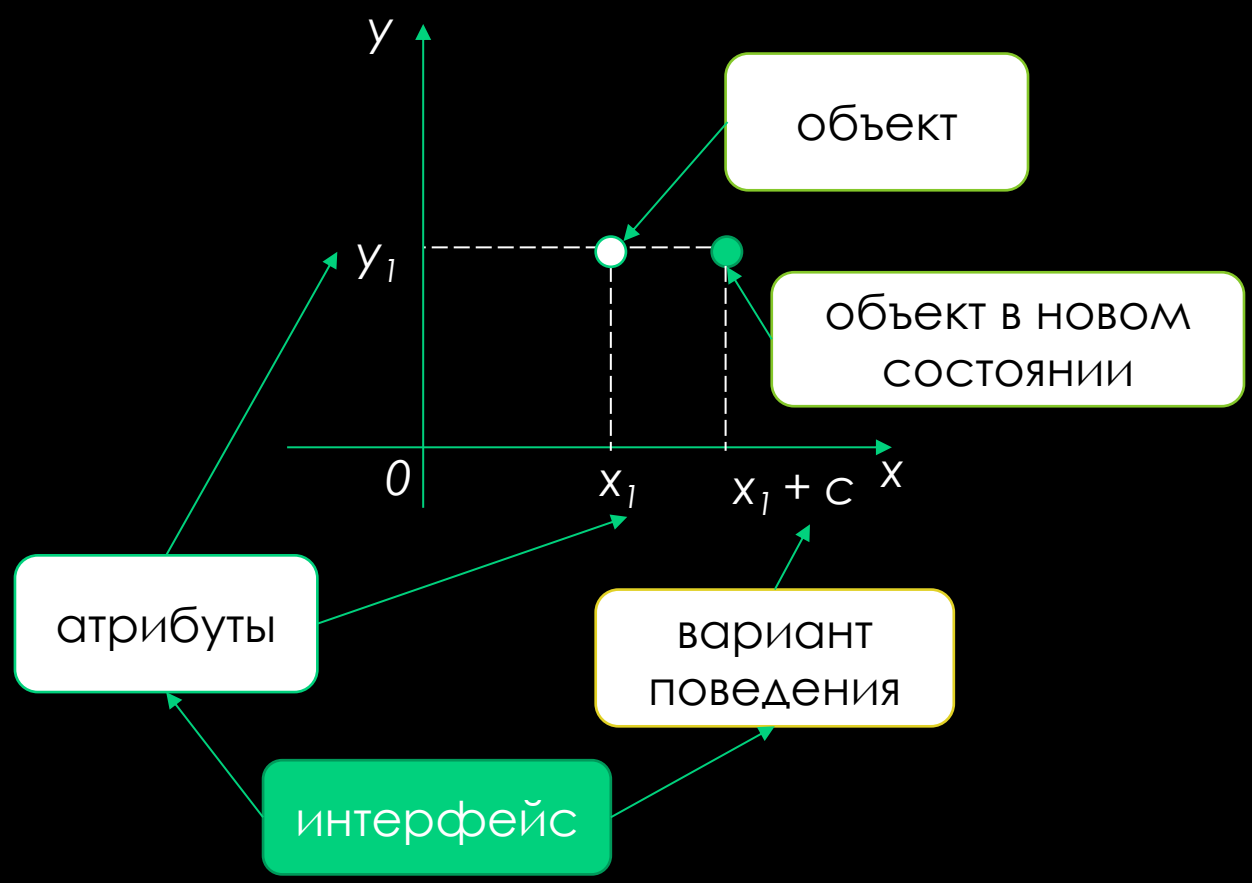
Можно представлять одни типы как подтипы других

Подтип всегда можно использовать там, где ожидается базовый тип

ОПРЕДЕЛЕНИЯ, СВЯЗАННЫЕ С ОБЪЕКТОМ

- **Объект** [object] – произвольная сущность, для которой достаточно легко определить границы, состояние и поведение
  - **Состояние объекта** определяется совокупностью значений атрибутов
    Атрибут [attribute] – свойство объекта
  - **Поведение объекта** определяется совокупностью методов
    - **Метод** [method] – вариант поведения объекта
  - **Интерфейс** [interface] **объекта** – внешнее представление объекта, то есть всего его атрибуты и методы
    - Описание интерфейса задаёт **границы объекта** самим фактом включения или не включения тех или иных атрибутов или методов

РАЗБИРАЕМСЯ С ТЕРМИНАМИ



СОЗДАНИЕ ОБЪЕКТОВ

Синтаксис создания объектов в C#:

```
<Тип> <Идентификатор ссылки> = new <Тип>([Параметры]);
```

```
string str = new string('a', 15);
```

```
string str = new ('a',15);
```

```
<Тип> <Идентификатор ссылки> = new ([Параметры]); // (начиная с C# 9.0)
```

ОПРЕДЕЛЕНИЯ, СВЯЗАННЫЕ С КЛАССОМ

- **Класс** [class] – описание однотипных объектов, то есть их интерфейса
- Объект, удовлетворяющий этому шаблону, называется **экземпляром** [instance] **класса**

Класс:
- Точка на вещественной плоскости

Состояние:
- координата по оси X
- координата по оси Y

Поведение:
- получить координату по оси X
- получить координату по оси Y
- изменить координату по оси X на вещественную константу C

КЛАСС, КАК ТИП ДАННЫХ

Класс – это тип данных

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

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

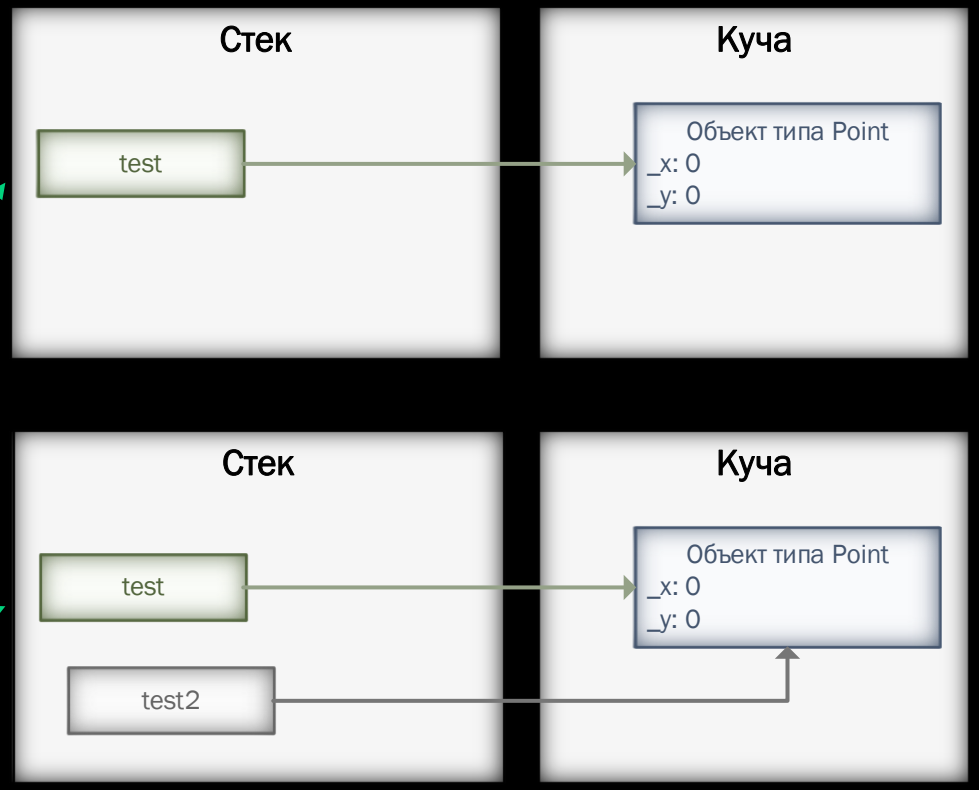
Состояние

Поведение

```
Point test = new Point();
Console.WriteLine(test);
```

```
Point test2 = test;
```

Класс – это ссылочный тип данных



По умолчанию **internal**

По умолчанию **private**

Свойства доступа к полям

В интерфейс входит только отмеченное **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 void MoveX(double c) => _x += c;

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

Класс:
- Точка на вещественной плоскости

Состояние:
- координата по оси X
- координата по оси Y

Поведение:
- получить координату по оси X
- получить координату по оси Y
- изменить координату по оси X на вещественную константу C

МОДИФИКАЦИЯ НА ОСНОВЕ АВТОРЕАЛИЗУЕМЫХ СВОЙСТВ

```
class Point
{
    public double X { get;set; }
    public double Y { get;set; }

    public void MoveX(double c) => X += c;

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

Поля теперь неявно спрятаны за автореализуемые свойства

Теперь обращаемся к свойствам, а не к полям

СОЗДАНИЕ ОБЪЕКТОВ КЛАССА

Конструктор

Операция new

ОПЕРАЦИЯ NEW

- **вычисляет количество байтов**, необходимых для хранения всех экземплярных полей типа и всех его базовых типов, включая System.Object
- **выделяет память для объекта с резервированием необходимого для данного типа количества байтов в управляемой куче**
  - Выделенные байты инициализируются нулями
- **инициализирует указатель на объект-тип и индекс блока синхронизации**
- **вызывает конструктор экземпляра типа с параметрами, указанными при вызове new**
- **возвращает ссылку на вновь созданный объект**

ПРИМЕРЫ ИСПОЛЬЗОВАНИЯ ОПЕРАЦИИ NEW

```
Point p1 = new Point();
```

```
Point p2 = new();
```

Имя типа можно опустить, т.к. известен целевой тип конструктора

```
Point p3 = new() { X = 1.2, Y = 4.3 };
```

Имя типа можно опустить, т.к. известен целевой тип конструктора и использовать список инициализации

```
Point p2 = new(X: 1.2, Y: 4.3);
```

Такой вариант в нашем случае не сработает, т.к. требуется конструктор с двумя параметрами

КОНСТРУКТОРЫ КЛАССОВ

Конструктор – специальный функциональный член, позволяющий создавать объекты данного типа. Конструкторы позволяют описать обязательные параметры, необходимые для создания экземпляров

Синтаксис описания конструктора:

```
[Модификаторы] <Идентификатор Типа> ([Параметры]) {[Тело]}
```

При наличии хотя бы одного явно определённого конструктора, конструктор по умолчанию не генерируется

ОСОБЕННОСТИ КОНСТРУКТОРОВ

- Имя обязано совпадать с именем типа
- **Нет явно указанного типа возвращаемого значения**, они всегда (неявно) возвращают ссылку на созданный объект
- Могут быть перегружены, аналогично методам
- При отсутствии явных определений конструктора, к**омпилятор генерирует конструктор без параметров**, инициализирующий поля значениями их типов по умолчанию

```
class Point
{
    public double X { get;set; }
    public double Y { get;set; }

    public void MoveX(double c) => X += c;

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

```
Point p1 = new Point();
```

ЭКЗЕМПЛЯРЫ КЛАССА

```
class Point
{
    public double X { get;set; }
    public double Y { get;set; }

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

    public void MoveX(double c) => X += c;

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

Добавили конструкторы

Теперь это допустимо

```
Point p0= new Point();
Point p1 = new Point(1, 0);
Point p2 = new (x: 1.2, y: 4.3);
Point p3 = new() { X = 1.2, Y = 4.3 };
```

ССЫЛКА THIS

- Используется объектом для ссылки на себя
- Позволяют избегать путаницы при совпадающих именах параметров и членов классов
- В описании индексатора (об этом немного позже)

```
public Point(double x, double y) => (this.X, this.Y) = (x, y);
```

```
public Point(double x, double y) : this() => (this.X, this.Y) = (x, y);
```

КОНСТРУКТОРЫ И МОДИФИКАТОР READONLY

```
class Point
{
    readonly char _separator = ';';
    public double X { get; set; }
    public double Y { get; set; }

    public Point() => _separator = ':';

    public Point(double x, double y) (X, Y) = (x, y);
    public void MoveX(double c) => X += c;

    public override string ToString() => $"({X:f3}{_separator}{Y:f3})";
}
```

readonly-поле получает значение только при объявлении или в конструкторе

Теперь для точек, созданных конструтором по умолчанию сепаратор будет :

ДЕСТРУКТОР

- **Деструктор** (метод завершения) – специальный метод класса, который вызывается автоматически при уничтожении объекта сборщиком мусора

```
class Point
{
    readonly char _separator = ';';
    public double X { get; set; }
    public double Y { get; set; }

    public Point() => _separator = ':';

    public Point(double x, double y) => (this.X, this.Y) = (x, y);
    public void MoveX(double c) => X += c;

    public override string ToString() => $"({X:f3}{_separator}{Y:f3})";
    ~Point() { }
}
```

Код для освобождения ресурсов

ОСОБЕННОСТИ ДЕСТРУКТОРА

- Деструкторы применяются только в классах, **в структурах их определить невозможно**
- Имя деструктора совпадает с именем класса
- При описании деструктора его имя предваряется знаком **~**
- Параметры у деструктора отсутствуют
- **Деструкторы не перегружаются, то есть в классе может быть только один деструктор**
- Деструктор в классе может быть только один
- Напрямую деструктор вызвать нельзя, он запускается только автоматически при наличии

СРАВНЕНИЕ КОНСТРУКТОРОВ И ДЕСТРУКТОРОВ

Для чего	**Кто**	**Когда и как вызывается**
Экземпляр	Конструктор	Однократно вызывается для создания каждого объекта класса (одно создание объекта – один вызов конструктора)
	Деструктор	Вызывается для каждого объекта класса, в некоторый момент времени после того, как в программе с объектом потеряна связь (ссылки отсутствуют)

КЛАССЫ – НАСЛЕДНИКИ OBJECT

Переопределение методов базового типа (немного о полиморфизме)

Ссылки с типом Object

КЛАССЫ C# НАСЛЕДНИКИ SYSTEM.OBJECT

Имя	Описание
Открытые (public)	
Equals(Object)	Возвращает true, если два объект-параметр совпадает с текущим.
GetHashCode()	Возвращает хеш-код для значения текущего объекта.
GetType()	Возвращает объект Type для текущего экземпляра.
ToString()	По умолчанию возвращает полное имя типа.
Защищённые (protected)	
MemberwiseClone()	Создаёт новый экземпляр и присваивает его полям значения соответствующих полей объекта this.
Finalize()	Позволяет объекту выполнить набор действий или освободить ресурсы до того, как он будет удалён сборщиком мусора.

ПЕРЕОПРЕДЕЛЕНИЕ TOSTRING()

Переопределение ToString() из класса Point

```
public override string ToString() => $"({X:f3}{_separator}{Y:f3})";
```

Вызов ToString() для объектов класса Point

```
Point p0 = new Point();
Console.WriteLine(p0.ToString());

Point p1 = new Point(1, 0);
Console.WriteLine(p1.ToString());
```

```
(0,000:0,000)
(1,000;0,000)
```

```
Point p2 = new(x: 1.2, y: 4.3);
Console.WriteLine(p2.ToString());

Point p3 = new() { X = 1.2, Y = 4.3 };
Console.WriteLine(p3.ToString());
```

```
(1,200;4,300)
(1,200:4,300)
```

ПРИМЕР ССЫЛКИ С ТИПОМ OBJECT

```
public class Cat
{
    string _name;

    public Cat(string n) => _name = n;

    public override string ToString() => $"Cat:: {_name}";
}

public class Dog
{
    string _name;

    public Dog(string n) => _name = n;

    public override string ToString() => $"Dog:: {_name}";
}
```

Классы **Cat** и **Dog** не объединены общим предком, кроме **Object**

```
Cat c = new Cat("Мурка");
Dog d = new Dog("Шарик");

Object[] pets = { c, d };
foreach (Object cur in pets)
{
    Console.WriteLine(cur);
}
```

Объекты типов **Cat** и **Dog** собраны в массиве ссылок с типом **Object[]**

ПРИНЦИПЫ ООП И СТРУКТУРА КЛАССА

ПРОГРАММА СЧИТАЕТСЯ ОБЪЕКТНОЙ, ЕСЛИ

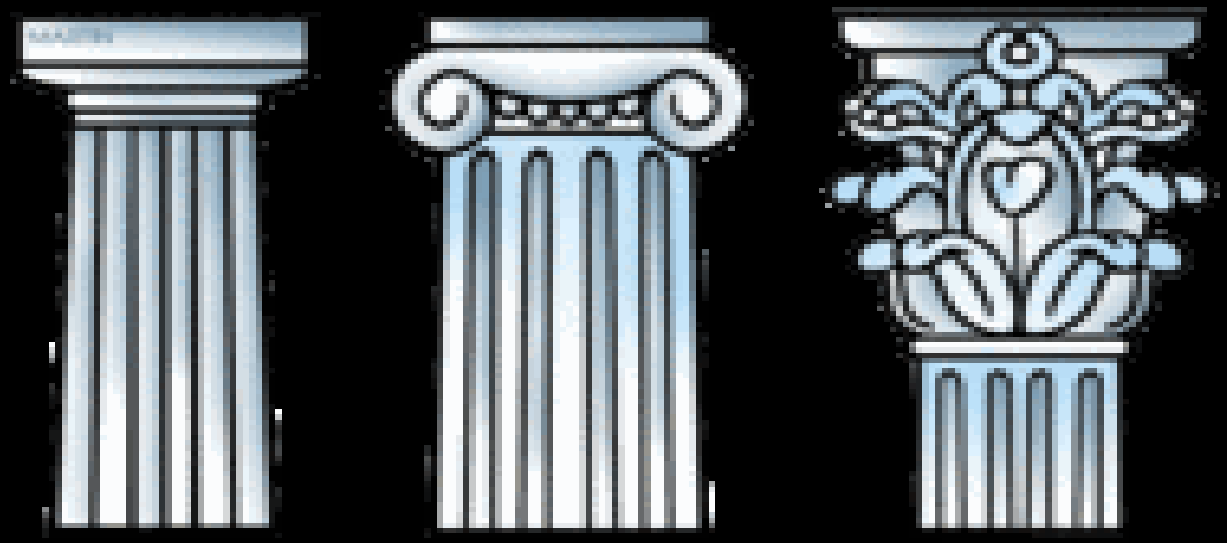
- использует в качестве основных конструктивных элементов объекты, а не алгоритмы
- каждый объект является экземпляром определённого класса
- классы образуют иерархии

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;                     // Инициализируется 0 - тип значения.
    string F2;                  // Инициализируется null - ссылочный тип.
    int F3 = 25;                // Инициализируется 25.
    string F4 = "abcd";         // Инициализируется "abcd".
    Random rnd = new Random();   // 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

Синтаксически вызов свойства похож на обращение к членам-данных классов

Объявление свойства состоит из описания методов-аксессоров

- геттер
- сеттер

```
[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 ĝêtƒ**: ř. ĝêtƒÝ лрл ř. ĝêtƒ

Такой геттер реализует вариант доступа «только на чтение»

Обращение к геттерам «как к членам-данным»

```
Point p0 = new Point();

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

СЕТТЕРЫ

```
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; } }
}
```

set-аксессор используется для назначения значения скрытому полю

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

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

set обязательно указывается, чтобы отметить, что это описание сеттера

```
Point p0 = new Point();
p0.X = 1;
p0.Y = 2;

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

value в свойстве

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

ГЕТТЕР ДЛЯ ОДНОКРАТНОГО НАЗНАЧЕНИЯ

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

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

    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;

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

Недопустимо переназначение

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

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

Свойства, состоящие из одной строчки (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

C# 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.Write("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.Write($"{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 индексаторы

СТАТИЧЕСКИЕ ЧЛЕНЫ НЕСТАТИЧЕСКИХ КЛАССОВ

const
readonly
static

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

МАТЕРИАЛЫ К СЕМИНАРУ 1-2

- Общие сведения о классах (https://learn.microsoft.com/ru-ru/dotnet/csharp/fundamentals/types/classes)
- Экземпляры структуры и экземпляры классов (https://learn.microsoft.com/ru-ru/dotnet/csharp/fundamentals/object-oriented/objects#struct-instances-vs-class-instances)
- Для сильных духом. Статья на английском C# YAGNI Principle (You Aren't Gonna Need It!)
- Модификаторы доступа (Справочник по C#) (https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/keywords/access-modifiers)

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.
- https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/const
- Модификаторы доступа (Справочник по C#) (https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/keywords/access-modifiers)
- Оператор is (справочник по C#)
- The Unix Philosophy in One Lesson [http://www.catb.org/~esr/writings/taoup/html/ch01s07.html]
- Occam's Razor and the Art of Software Design [http://michaellant.com/2010/08/10/occams-razor-and-the-art-of-software-design/]