

## Лабораторная работа №1: Создание и использование классов.

### Цель работы:

- Знакомство с базовыми принципами ООП.
- Знакомство с основами создания классов.

### Задачи:

- Реализовать программу для рисования геометрических фигур используя представленные примеры.
- Создать свой класс для четырехугольника. Создавать четырехугольник лучше всего через начальную точку и расстояния его длины и ширины.
- Реализовать функционал для создания треугольника и квадрата со случайными точками или заданными пользователем
- Реализовать функционал для перемещения фигур по сцене.

### Функционал программы:

- Рисование двух видов фигур на холсте – треугольника и прямоугольника. У пользователя должна быть возможность либо задать размеры фигур, либо они генерируются со случайным размером.
- Точка, треугольник и прямоугольник должны быть реализованы в качестве классов.
- В программе должен присутствовать функционал для перемещения фигур по холсту по обоим осям координат.

### Теоретическая информация.

**Объектно-ориентированное программирование (ООП)** — это парадигма программирования, в которой программа строится вокруг объектов.

**ООП** возникло как ответ на необходимость улучшения структуры и организации программ, а также на стремление сделать программирование более интуитивно понятным и управляемым. Процедурное программирование, основанное на разделении программы на функции и процедуры, становилось неэффективным для больших и сложных систем, так как это часто приводило к дублированию кода и запутанности.

Концепция ООП же основана на моделировании программы в терминах реальных сущностей или объектов, что сделало код более интуитивно понятным, легче расширяемым и поддерживаемым.

**Класс** — это основной инструмент в технологии ООП. Класс представляет собой шаблон или модель для создания объектов. Класс определяет общие характеристики (атрибуты) и поведение (методы) объектов, которые на его основе могут быть созданы. Класс является логической абстракцией. Физическое представление класса появится в программе лишь после того, как будет создан **объект** (экземпляр) этого класса.

Основная идея при выделении классов заключается в обобщении некоторых сущностей, которые обладают схожими параметрами и выполняют одинаковые функции. К примеру, мы делаем программу для учета книг в библиотеке. Нашим основным **классом** в данном случае будет как раз «Книга», которая будет обладать некоторыми атрибутами – названием, автором, количеством страниц. А уже какие-то записи о существующих в библиотеке книгах будут экземплярами **объектов** данного класса:

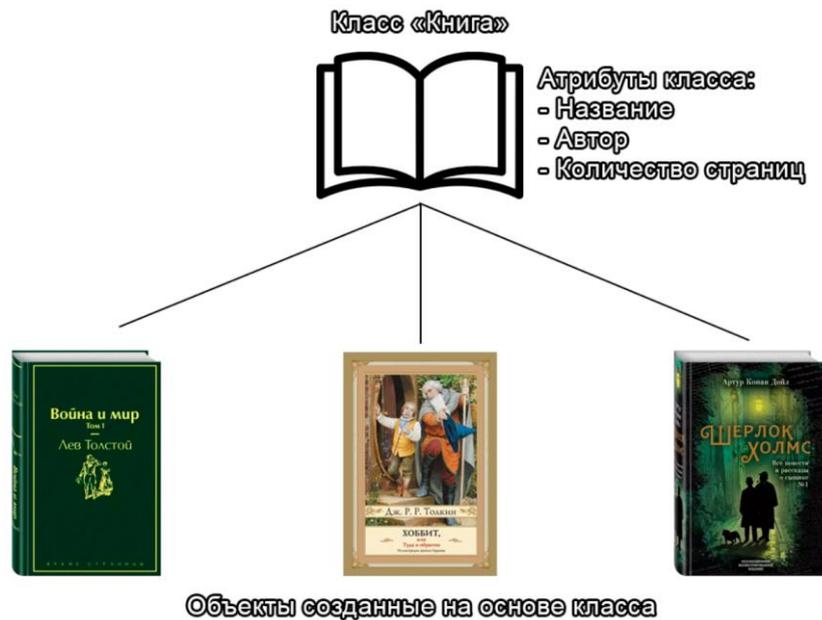


Рисунок 1 – Визуальный пример класса «Книга» и объектов, созданных на его основе

Данная реализация решает проблему модифицируемости нашей программы – при появлении в библиотеке новой книги требуется всего лишь создать новый экземпляр объекта с нужными данными. На языке программирования C# класс создается с помощью ключевого слова **class**. Ниже приведена общая структура класса, содержащая несколько атрибутов и методов:

```
//тело класса
public class Person
{
    //атрибуты класса
    int age;
    string first_name;
    string second_name;
    double height;

    //методы класса
    public string getFullName()
    {
        //метод возвращает строку из комбинации имени и фамилии
        return $"{first_name} {second_name}";
    }

    public double getHeightInIches()
    {
        //метод возвращает рост переведенный из сантиметров в дюймы
        return height * 2.54;
    }
}
```

В данном примере у класса под названием *Person* существует четыре различных атрибута разных типов данных и два метода которые используют эти атрибуты.

Перед каждым объявлением поля или метода указывается **модификатор доступа**. Модификатор доступа определяет тип разрешенного доступа. Существует несколько различных модификаторов, но в данной работе мы будем использовать следующие два модификатора:

- **public** - данный модификатор доступа делает атрибут или метод доступным всем. То есть к публичным параметрам можно обращаться из любой части кода, в том числе из других классов.
- **private** - данный модификатор доступа позволяет обращаться к атрибутам и методам класса только изнутри самого класса. Если не указывается модификатор доступа, то стандартным выбирается модификатор `private`.

В данном примере класса его атрибуты обозначены как *приватные*. Но методы, которые реализует данный класс являются *публичными* и через эти публичные методы мы имеем возможность взаимодействовать с *приватными* полями.

Добавление класса осуществляется в контекстном меню проекта: необходимо выбрать пункт «Добавить – Класс...» (или воспользоваться комбинацией клавиш `Shift+Alt+C`). В появившемся диалоговом окне необходимо ввести имя файла – данное имя так же служит и названием для реализуемого класса.

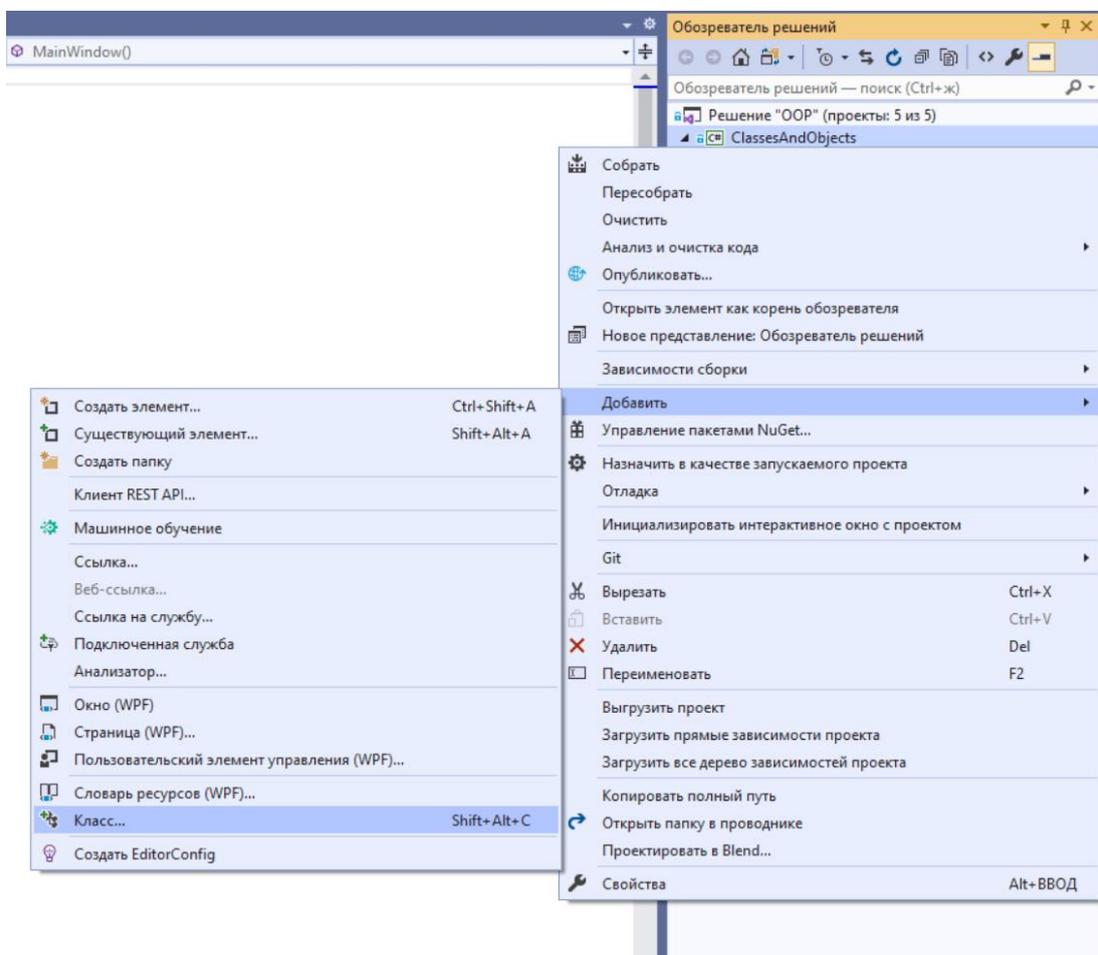


Рисунок 2 – Создание нового класса в проекте

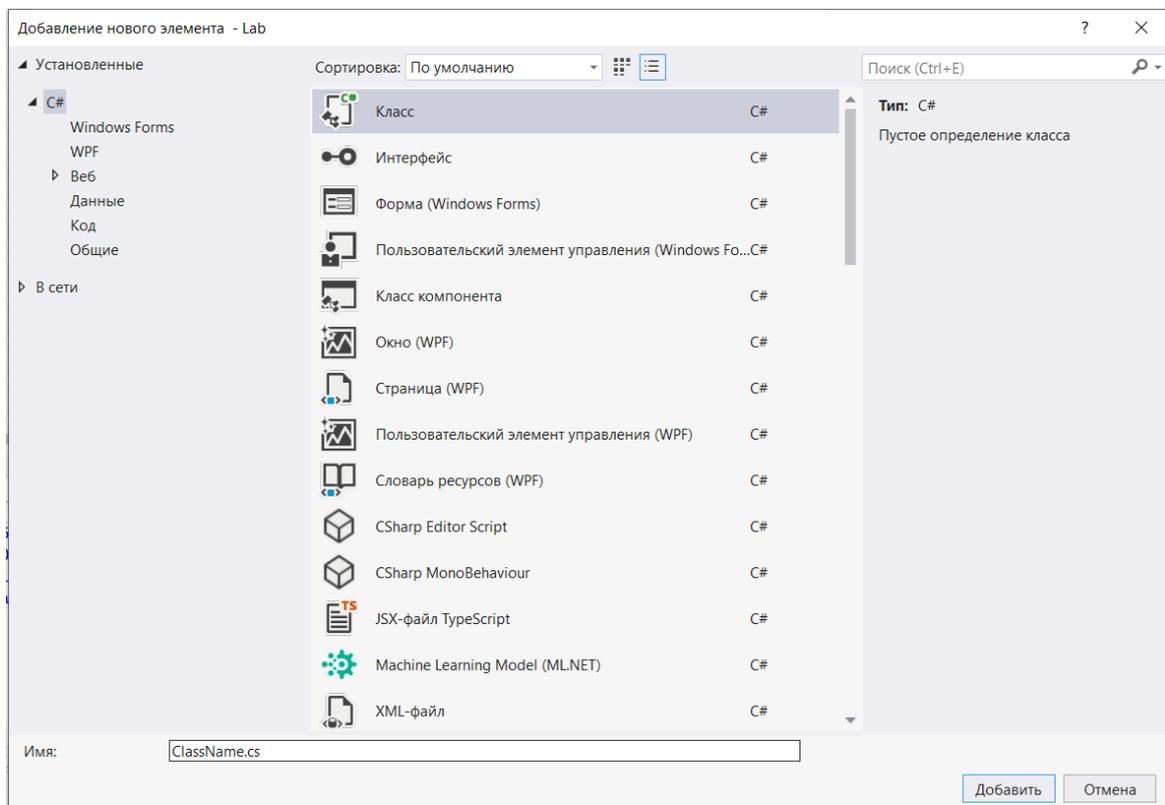


Рисунок 3 – Диалоговое окно добавления нового элемента

Одним из самых важных элементов класса является его конструктор. **Конструктор** – метод, инициализирующий объект при его создании. У конструктора такое же имя, как и у его класса, а с точки зрения синтаксиса он подобен методу, за исключением явно указываемого возвращаемого типа. Конструктор класса вызывается автоматически при использовании оператора **new**.

```
//тело класса
public class Person
{
    //атрибуты класса
    int age;
    string first_name;
    string second_name;
    double height;

    //конструктор класса
    public Person(int Age, string FName, string SName, double Height)
    {
        age = Age; //присваивание атрибутам передаваемые значения
        first_name = FName;
        second_name = SName;
        height = Height;
    }
}
```

Теперь в основной программе мы можем создать новый экземпляр класса используя данный конструктор:

```
public MainWindow()
{
    InitializeComponent();

    //создание экземпляра класса
    Person person1 = new Person(18, "Иван", "Иванов", 179.56);
}
```

### Задание. Программа рисования геометрических фигур.

В качестве задания на лабораторную работу требуется написать программу с соответствующим функционалом.

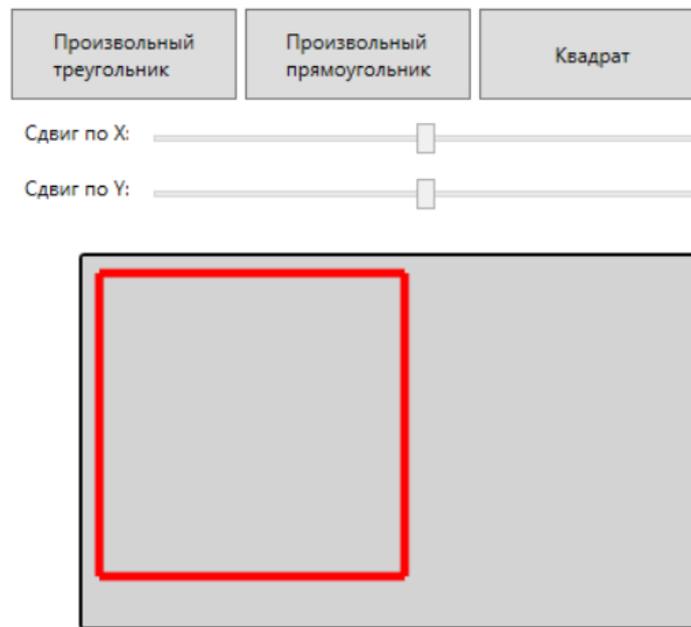


Рисунок 4 – Пример интерфейса программы

Для рисования геометрических фигур в окне необходимо создать «холст» в области окна. В качестве «холста» можно использовать элемент управления **Canvas**. В данном примере холст имеет название *“Scene”*.

```
<Grid>
    <Canvas x:Name="Scene" HorizontalAlignment="Left" Height="400" Margin="15,0,0,0"
    VerticalAlignment="Center" Width="680"/>
</Grid>
```

Для того чтобы рисовать геометрические фигуры предлагается использовать отрисовку по линиям. Для этого можно использовать класс **Line**. Рисование линии с помощью класса Line состоит из следующих шагов:

- создание объекта (экземпляра) класса Line;
- задание визуальных параметров линии: цвета и толщины;
- задание относительных координат начала и конца линии;
- добавление линии в холст.

Для того чтобы хранить информацию о координатах создадим класс точки *Point2D*. Данный класс логически будет представлять собой точку с координатами X и Y:

```
public class Point2D
{
    //Атрибуты класса
    private int X;
    private int Y;

    //Конструктор класса
    public Point2D(int x, int y)
    {
        //this используется для однозначного указания на атрибуты класса так как переменные
        имеют одинаковые имена
        this.X = x;
        this.Y = y;
    }

    //Методы для получения координат
    public int getX()
    {
        return X;
    }

    public int getY()
    {
        return Y;
    }

    //Методы для изменения координат
    public void addX(int x)
    {
        X += x;
    }

    public void addY(int y)
    {
        Y += y;
    }
}
```

Теперь, когда у нас есть готовый класс, который содержит в себе информацию о координатах его можно использовать в другом классе, который будет реализовывать структуру фигуры. Свойство, когда один класс использует другой класс в качестве атрибута называется **агрегацией**.

Создадим класс треугольника *Triangle*:

```
public class Triangle
{
    //Атрибуты класса
    private Point2D p1;
    private Point2D p2;
    private Point2D p3;

    //Конструктор класса
    public Triangle(Point2D p1, Point2D p2, Point2D p3)
    {
        this.p1 = p1;
        this.p2 = p2;
        this.p3 = p3;
    }

    public Point2D getP1()
    {
        return p1;
    }

    public Point2D getP2()
    {
        return p2;
    }

    public Point2D getP3()
    {
        return p3;
    }
}
```

Для того чтобы изменять координаты воспользуемся методом изменения координат который был реализован в классе точки и применим его ко всем точкам внутри класса фигуры:

```
public void addX(int X)
{
    p1.addX(X);
    p2.addX(X);
    p3.addX(X);
}

public void addY(int Y)
{
    p1.addY(Y);
    p2.addY(Y);
    p3.addY(Y);
}
```

Даже учитывая одинаковые названия методов программа однозначно вызывает метод для конкретного класса – в данном случае класс Point2D вызывает свой метод addX/addY так как он вызывается от экземпляра класса Point2D.

Полученные классы можно представить в виде следующих UML-диаграмм:

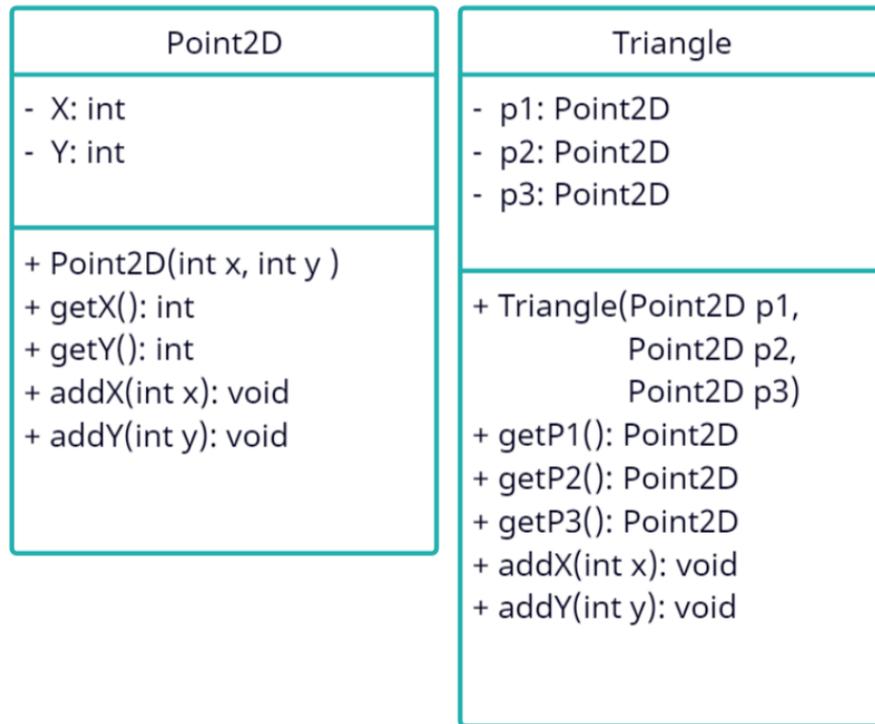


Рисунок 5 – UML-диаграммы классов Point2D и Triangle

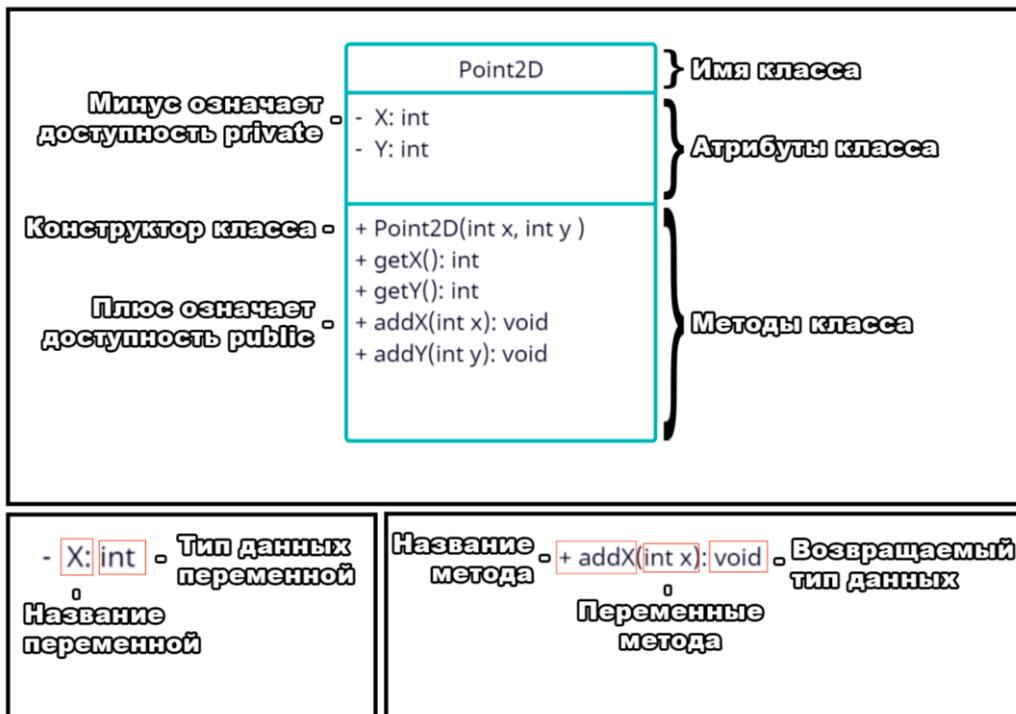


Рисунок 6 – Как читать UML-диаграммы

Далее реализуем функцию в основном коде программы для рисования линии по двум точкам, в данном примере используется встроенный класс для рисования линии:

```
//функция в основном теле программы
public void DrawLine(Point2D p1, Point2D p2)
{
    //Создание новой линии
    Line line = new Line();
    //Цвет и толщина линии
    line.Stroke = Brushes.Red;
    line.StrokeThickness = 3;

    //Установка координат линии из координат точек Point2D
    line.X1 = p1.getX();
    line.Y1 = p1.getY();
    line.X2 = p2.getX();
    line.Y2 = p2.getY();

    //Добавление линии в Canvas
    Scene.Children.Add(line);
}
```

Создадим новый треугольник со случайными координатами:

```
Triangle tr;
Random rnd = new Random();

public MainWindow()
{
    //Создание треугольника со случайными координатами
    Point2D p1 = new Point2D(rnd.Next(0, (int)Scene.Width), rnd.Next(0, (int)Scene.Height));
    Point2D p2 = new Point2D(rnd.Next(0, (int)Scene.Width), rnd.Next(0, (int)Scene.Height));
    Point2D p3 = new Point2D(rnd.Next(0, (int)Scene.Width), rnd.Next(0, (int)Scene.Height));
    tr = new Triangle(p1, p2, p3);
}
```

Функция рисования треугольника будет выглядеть следующим образом:

```
public void DrawTriangle(Triangle tr)
{
    //Отрисовка треугольника с помощью функции отрисовки линии
    DrawLine(tr.getP1(), tr.getP2());
    DrawLine(tr.getP2(), tr.getP3());
    DrawLine(tr.getP3(), tr.getP1());
}
```

Для того чтобы очистить сцену от линий можно использовать следующую функцию:

```
public void ClearScene()
{
    //Очистка Canvas от всех объектов
    Scene.Children.Clear();
}
```

## Отчет:

Результат выполнения работы предоставить в виде:

- Архив с проектом (если размер архива больше 2 Мбайт, то рекомендуется загрузить проект на <https://github.com/> или на другое общедоступное хранилище и предоставить ссылку);
- Отчет по лабораторной работе в формате Microsoft Word, который содержит следующие разделы:
  1. титульный лист;
  2. задание на лабораторную работу;
  3. краткое описание разработанных программ и используемых алгоритмов со скриншотами выполнения;
  4. вывод о проделанной работе.

## Список литературы:

- 1) Герберт Шилдт "С# 4.0: полное руководство"
- 2) Эндрю Троелсен "Язык программирования С# 5.0 и платформа .NET 4.5"
- 3) Полное руководство по языку программирования С# 7.0 и платформе .NET 4.7  
<https://metanit.com/sharp/tutorial/>
- 4) Руководство по WPF <https://metanit.com/sharp/wpf/>
- 5) С# 5.0 и платформа .NET 4.5  
[http://professorweb.ru/my/csharp/charp\\_theory/level1/infocsharp.php](http://professorweb.ru/my/csharp/charp_theory/level1/infocsharp.php)
- 6) <https://github.com/Microsoft/WPF-Samples>