Промышленное программирование

Лекция 2

❶ Классы

❷ Наследование и полиморфизм

❶ **Классы**

❷ Наследование и полиморфизм

Поля

```
[AccessModifier] [static] [readonly] Type _name [= value];
[AccessModifier] const Type _name = value;
```

AccessModifier	Visibility
`private`	Текущий класс
`private protected`	Наследники внутри текущей сборки
`protected`	Любые наследники
`internal`	Текущая сборка
`protected internal`	Текущая сборка и любые наследники
`public`	Любые классы

Свойства

```
[AccessModifier] [static] Type Name {
    [AccessModifier] get { /* Type get() */ }
    [AccessModifier] set { /* void set(Type value) */ }
}
```

```
public class List {
    public int Count {
        get => _count;
        private set {
            if (value < 0)
                throw new ArgumentOutOfRangeException(nameof(value));
            _count = value;
        }
    }

    private int _count;
}
```

Автоматически реализуемые свойства

```
public class List {
    public int Count { get; set; } = 0;
    public bool IsReadOnly => false;
}
```

Пример	Описание
`int Count { get; }`	Сгенерированное поле будет readonly. Присвоение возможно только: 1) непосредственной инициализацией; 2) в конструкторе.
`int Count { get; init; }`	Присвоение возможно только: 1) непосредственной инициализацией; 2) в конструкторе; 3) в инициализаторе объекта.

Методы

```
[AccessModifier] [static] Type Name([ref|out] Type name [, …]);
```

Модификатор	Описание
`ref`	Передача по ссылке: 1) при вызове значение **должно** быть указано; 2) метод **может** перезаписать саму ссылку или значение.
`out`	Возвращаемый параметр: 1) при вызове значение **может** быть неопределено; 2) метод **должен** присвоить значение.

Пример

```
internal static class Program
{
    static int Sum(out int count, params int[] values)
    {
        count = values.Length;
        int sum = 0;
        foreach (var value in values)
            sum += value;
        return sum;
    }

    static void Main()
    {
        var sum = Sum(out var count, 1, 2, 3);
        Console.WriteLine($"count={count}, sum={sum}");
    }
}
```

Инициализаторы объектов

```
static class Program {
    class List {
        public int Count { get; set; }

        public bool IsReadOnly { get; init; }

        public override string ToString()
            => $"Count={Count}, IsReadOnly={IsReadOnly}";
    }

    static void Main() {
        var list = new List {
            Count = 5,
            IsReadOnly = true
        };
        Console.WriteLine(list);
    }
}
```

Перегрузка операторов

```
public struct Point
{
    public static Point operator +(Point lhs, Point rhs)
        => new(lhs.X + rhs.X, lhs.Y + rhs.Y);

    public int X { get; init; }
    public int Y { get; init; }

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

Индексаторы

```
static class Program {
    class List {
        private readonly int[] _values = new int[2];

        public int this[int index] {
            get => _values[index];
            set => _values[index] = value;
        }
    }

    static void Main() {
        var list = new List();
        Console.WriteLine($"{list[0]} {list[1]}");
        list[0] = 5;
        Console.WriteLine($"{list[0]} {list[1]}");
    }
}
```

❶ Классы

❷ Наследование и полиморфизм

Наследование

```
static class Program {
    class A {
        public int Value { get; }
        public A(int value) { Value = value; }
        public void Print() => Console.WriteLine(Value);
    }

    class B : A {
        public B() : base(5) { }
    }

    static void Print(A a) => a.Print();

    static void Main() {
        var b = new B();
        Print(b);
    }
}
```

Перекрытие методов

```
static class Program {
    class A {
        public void Print() => Console.WriteLine(nameof(A));
    }

    class B : A {
        public new void Print() => Console.WriteLine(nameof(B));
    }

    static void Main() {
        var aa = new A();
        A ab = new B();
        var bb = new B();
        aa.Print();
        ab.Print();
        bb.Print();
    }
}
```

Переопределение методов

```
static class Program {
    class A {
        public virtual void Print() => Console.WriteLine(nameof(A));
    }

    class B : A {
        public override void Print() => Console.WriteLine(nameof(B));
    }

    static void Main() {
        var aa = new A();
        A ab = new B();
        var bb = new B();
        aa.Print();
        ab.Print();
        bb.Print();
    }
}
```

Абстрактные классы и методы

```
static class Program {
    abstract class A {
        public abstract void Print();
    }

    class B : A {
        public override void Print() => Console.WriteLine(nameof(B));
    }

    static void Main() {
        // var aa = new A();
        A ab = new B();
        var bb = new B();
        // aa.Print();
        ab.Print();
        bb.Print();
    }
}
```

Интерфейсы

```
static class Program {
    interface IContext {
        int GetContext();
    }
    class BaseWindow : IContext {
        public int GetContext() => 3;
    }
    class DerivedWindow : BaseWindow {
        public new int GetContext() => 7;
    }

    static void Main() {
        IContext baseContext = new BaseWindow();
        IContext derivedContext = new DerivedWindow();
        Console.WriteLine(baseContext.GetContext());
        Console.WriteLine(derivedContext.GetContext());
    }
}
```

Явно реализуемые интерфейсы

```
static class Program {
    interface IIntContext {
        int Context { get; }
    }
    interface IStringContext {
        string Context { get; }
    }
    class Window : IIntContext, IStringContext {
        int IIntContext.Context => 5;
        string IStringContext.Context => "Hello";
    }
    static void Main() {
        var window = new Window();
        Console.WriteLine(((IIntContext)window).Context);
        Console.WriteLine(((IStringContext)window).Context);
        // Console.WriteLine(window.Context);
    }
}
```

Заключение

1. Классы
    1. Новое понятие: свойство
    2. Новая конструкция: инициализатор объекта
2. Перегрузка операций
    1. В общем случае для классов не рекомендуется
    2. Исключение – индексатор
3. Наследование
    1. Множественного наследования нет
    2. Но класс может реализовать любое количество интерфейсов
    3. В случае конфликтов есть механизм явной реализации
    4. Реализации интерфейсов по умолчанию невиртуальные