

Программирование и обработка графических интерфейсов

Лекция 4. Абстрактные классы и интерфейсы.
Полиморфизм

Абстрактные классы

- **Абстрактный класс** – это класс, экземпляры которого нельзя создать
- Базовый класс объявляют абстрактным, если не имеет смысла создавать его экземпляры, а имеет смысл создавать только экземпляры его наследников
- Пусть класс Shape – абстрактный:

```
Shape shape = new Shape(); // будет ошибка компиляции  
//нельзя создавать экземпляр абстрактного класс
```

Абстрактные классы

```
public abstract class Shape
{
    private Color color;
    public abstract double GetWidth();
    public abstract double GetHeight();
    public abstract double GetArea();

    protected Shape(Color color)
    {
        this.color = color;
    }

    protected Color GetColor()
    {
        return color;
    }
}
```

Абстрактный класс надо пометить словом `abstract`

Абстрактный класс может иметь абстрактные методы – методы без реализации. Их надо пометить словом `abstract`

Конструктор часто делают `protected` - все равно создать экземпляры нельзя

Абстрактный класс может иметь обычные поля и методы

Абстрактные свойства

```
abstract class Person
```

```
{  
    public abstract string Name { get; set; }  
}
```

Абстрактное свойство

```
class Client : Person
```

```
{  
    private string name;
```

```
    public override string Name  
    {  
        get { return "Mr/Ms. " + name; }  
        set { name = value; }  
    }  
}
```

Переопределение свойства

```
class Employee : Person
```

```
{  
    public override string Name { get; set; }  
}
```

Автоматическое свойство

Абстрактные классы

- Экземпляры абстрактного класса нельзя создать

```
Shape shape = new Shape(); //ошибка компиляции
```

- Если в классе есть хотя бы один абстрактный метод или от родителей достался нереализованный абстрактный метод, то класс обязан быть абстрактным

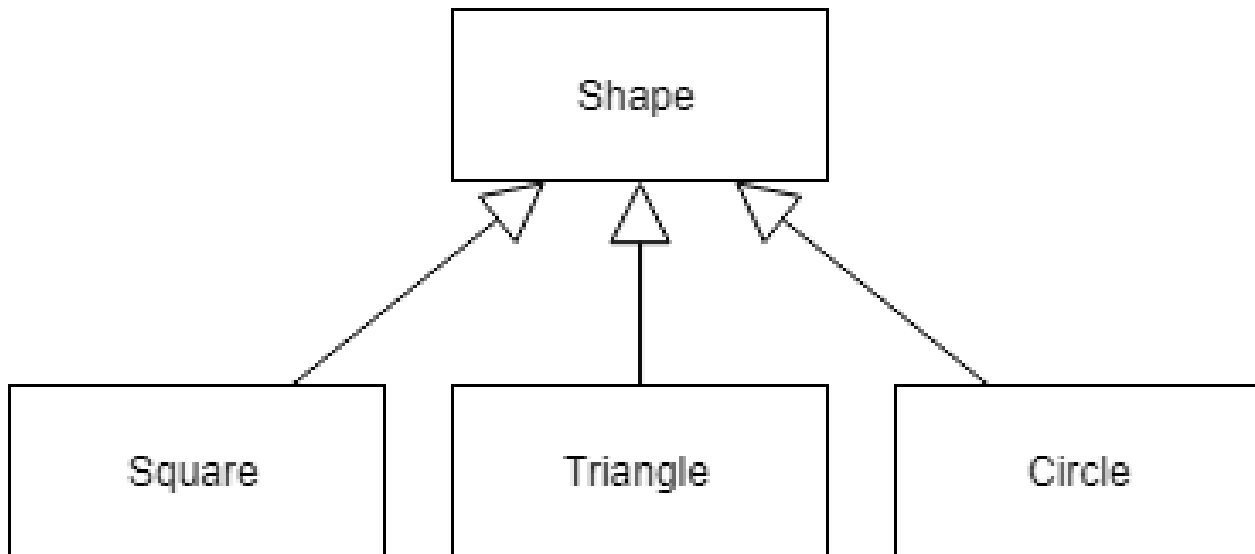
```
public class TestClass
{
    // ошибка компиляции – класс должен быть помечен как abstract
    public abstract void TestMethod();
}
```

Когда следует делать класс абстрактным?

- Когда нет смысла создавать экземпляры этого класса
- Например, на текущем уровне абстракции непонятно, как реализовать какие-то методы, и нет какой-то разумной реализации по умолчанию
- Тут непонятно, каковы размеры и площадь фигуры, потому что мы не знаем её тип, положение

```
public abstract class Shape
{
    public abstract double GetWidth();
    public abstract double GetHeight();
    public abstract double GetArea();
}
```

Когда следует делать класс абстрактным?



Когда не следует делать класс абстрактным?

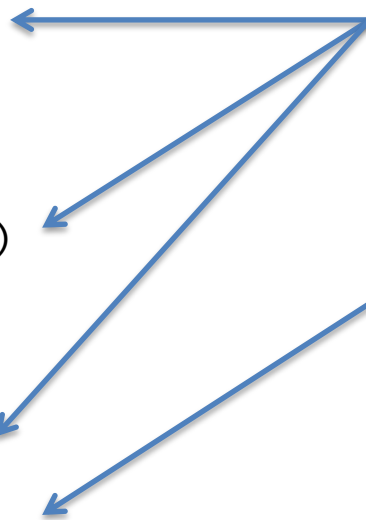
```
public class Square : Shape
{
    private double sideLength;
    public Square(Color color, double sideLength) : base(color)
    {
        this.sideLength = sideLength;
    }

    public override double GetWidth()
    {
        return sideLength;
    }

    public override double GetHeight()
    {
        return sideLength;
    }

    public override double GetArea()
    {
        return sideLength * sideLength;
    }
}
```

Мы реализовали все абстрактные методы всех родителей, поэтому класс можно делать не абстрактным



Для квадрата мы уже понимаем как посчитать размеры и площадь

Когда не следует делать класс абстрактным?

- Объекты квадратов уже можно создавать и использовать:

```
Shape s = new Square(Color.RED, 1);  
Console.WriteLine(s.GetArea());
```

Абстрактный класс без абстрактных методов

- Класс можно делать абстрактным, даже если в нем нет собственных или унаследованных от родителей абстрактных методов

```
public abstract class TestClass
{
    public void TestMethod()
    {
        Console.WriteLine("TestMethod");
    }
}
```

Отказ от реализации абстрактных членов

```
abstract class Person
{
    public abstract string Name { get; set; }
}
```

```
abstract class Manager : Person
{
    // абстрактный класс не обязан иметь реализацию абстрактных членов,
    // доставшихся от базового класса
}
```

Зачем нужны абстрактные классы?

- Чтобы создавать базовые классы, которые реализуют некоторую общую логику, но при этом некоторые аспекты на данном уровне абстракции еще не известны
- Пример – абстрактный класс Shape, который мы рассматривали

Интерфейсы в терминах ООП

- **Интерфейс** обычно подобен абстрактному базовому классу, имеющему только абстрактные члены.

Интерфейсы в С#

- В С# интерфейсы обозначаются при помощи ключевого слова **interface**
- Для интерфейсов принято соглашение именования – начинать их с буквы I (от Interface)

```
public interface IShape
{
    double GetWidth();
    double GetHeight();
    double GetArea();
}
```

Интерфейсы в C# могут содержать:

```
interface IMovable
```

```
{
```

```
    void Move();
```

← Методы

```
    string Name { get; set; }
```

← Свойства

```
    event MoveHandler MoveEvent;
```

← События

```
    string this[int index] { get; set; }
```

← Индексаторы

```
}
```

Начиная C# 8.0, интерфейсы могут содержать:

- Константы
- Операторы
- Статический конструктор
- Вложенные типы
- Статические поля, методы, свойства, индексы и события

Интерфейсы в С#

- Интерфейс может иметь только public члены
- Модификаторы видимости указывать нельзя, они всегда подразумеваются public

```
public interface IShape
{
    double GetWidth();
    double GetHeight();
    double GetArea();
}
```

Интерфейсы в C#

- Но также, начиная с версии C# 8.0, мы можем явно указывать модификаторы доступа у компонентов интерфейса:

```
public interface IShape
{
    private double GetWidth();
    private double GetHeight();
    protected double GetArea();
}
```

Реализация интерфейса

- Так как интерфейсы в терминах ООП – абстрактные классы, то нельзя создавать их экземпляры
- От интерфейса можно наследоваться, как от обычного класса, указав двоеточие

```
public class Square : IShape
{
    public void GetWidth()
    {
        // ..
    }
    // реализация методов GetHeight, GetArea
}
```

Реализация интерфейса

- Вместо слова «наследуют», про интерфейсы говорят что их «реализуют»
- То есть **класс Square реализует интерфейс IShape**
- Implement с англ. – реализовывать

Реализация интерфейса

- Если класс реализует интерфейс, то он должен реализовывать все его методы, либо быть абстрактным

Реализация нескольких интерфейсов


- Класс может реализовывать несколько интерфейсов, в этом отличие от классов

```
public interface ITest1
{
    void TestMethod1();
}

public interface ITest2
{
    void TestMethod2();
}

public class TestClass : ITest1, ITest2
{
    public void TestMethod1()
    {
    }

    public void TestMethod2()
    {
    }
}
```



The diagram illustrates the implementation of multiple interfaces by a class. It shows two interfaces, `ITest1` and `ITest2`, and a class `TestClass` that implements both. Arrows indicate the mapping from the abstract methods in the interfaces to the concrete implementations in the class: one arrow points from `void TestMethod1();` in `ITest1` to `public void TestMethod1()` in `TestClass`, and another arrow points from `void TestMethod2();` in `ITest2` to `public void TestMethod2()` in `TestClass`.

Реализация интерфейсов и наследование

- Можно одновременно наследоваться от некоторого класса и реализовывать сколько угодно интерфейсов. В этом случае класс должен идти первым после «:»

```
public abstract class Animal
{
    public abstract void Move();
}
```

```
public interface IEating
{
    void Eat();
}
```

```
public interface ISpeaking {
    void Speak();
}
```

```
public class Cat : Animal, IEating, ISpeaking
{
    public override void Move()
    {
    }

    public void Eat()
    {
    }

    public void Speak()
    {
    }
}
```

Реализация по умолчанию

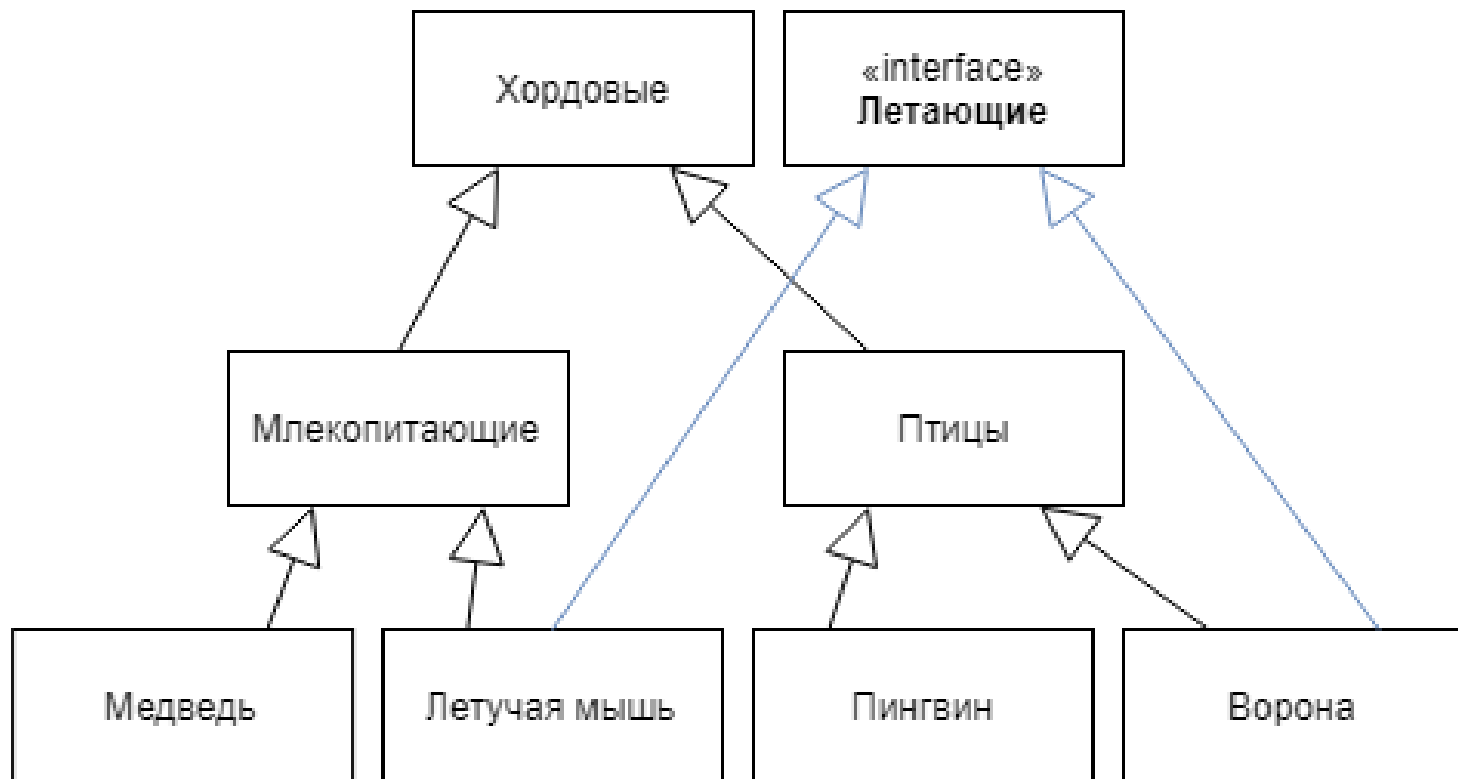
- Начиная с версии C# 8.0 интерфейсы поддерживают реализацию методов и свойств по умолчанию.

```
interface IMovable
{
    void Move()
    {
        Console.WriteLine("Walking");
    }

    // реализация свойства по умолчанию только для чтения
    int MaxSpeed { get { return 0; } }
}
```


Когда полезны интерфейсы?

- Чтобы указать для класса признак, который не вписывается в иерархию классов



Когда полезны интерфейсы?

- Когда хочется выделить некоторую абстракцию, но непонятно, как она будет реализована. Или реализации могут быть абсолютно несхожими между собой

```
interface ILogger
{
    // сообщает о предупреждении
    void Warning(string text);
    // сообщает об ошибке
    void Error(string text);
    // информационное сообщение
    void Info(string text);
}
```

Применение интерфейсов в классах и структурах

```
interface IMovable
```

```
{  
    void Move();  
}
```

```
class Person : IMovable
```

```
{  
    public void Move()  
    {  
        Console.WriteLine("Человек идет");  
    }  
}
```

Применение
интерфейса в
классе

```
struct Car : IMovable
```

```
{  
    public void Move()  
    {  
        Console.WriteLine("Машина едет");  
    }  
}
```

Применение
интерфейса в
структуре

Использование интерфейсов в коде

```
class ConsoleLogger : ILogger
{
    public void Warning(string text)
    {
        Console.WriteLine($"Warning: {text}");
    }

    public void Error(string text)
    {
        Console.WriteLine($"Error: {text}");
    }

    public void Info(string text)
    {
        Console.WriteLine($"Info: {text}");
    }
}

...
ILogger logger = new ConsoleLogger();
```

Полиморфизм

- **Полиморфизм** — возможность объектов с одинаковой спецификацией иметь различную реализацию.
- Кратко смысл полиморфизма можно выразить фразой: **«Один интерфейс, множество реализаций»**.

Пример 1. Телевизоры

- Все телевизоры устроены по-разному, но имеют общий интерфейс - экран



Пример 2. Автомобили

- Водителю не обязательно знать внутреннее устройство и характеристики каждого автомобиля – все автомобили управляются одинаково



Полиморфизм

Преимущества

- Наглядность кода, повышение его абстракции
- Повышение повторного использования кода

Недостатки

- Сложность в понимании

Виды полиморфизма

Ad-hoc- полиморфизм

- Позволяет определять поведение в зависимости от входных аргументов

Параметрический полиморфизм

- Позволяет создавать универсальные базовые типы и методы для них
- Повышает повторное использования кода

Полиморфизм подтипов

- Упрощает разработку методов для разных типов данных

Виды полиморфизма

Ad-hoc- полиморфизм

- Перегрузка методов
- Методы с произвольным количеством аргументов
- Необязательные аргументы

Параметрический полиморфизм

- Обобщения (универсальные шаблоны)

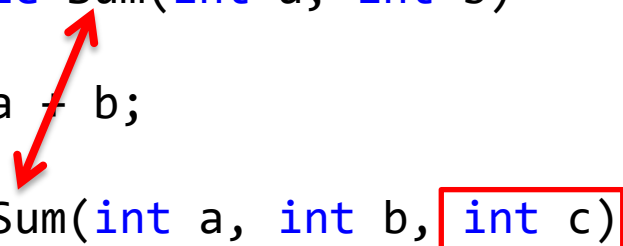
Полиморфизм подтипов

- Переопределение методов

Перегрузка методов

- В одном классе можно создавать методы с одинаковыми именами, но разной сигнатурой

```
public class Summator
{
    public double Sum(int a, int b)
    {
        return a + b;
    }
    public int Sum(int a, int b, int c)
    {
        return a + b + c;
    }
}
```



Сигнатура метода для перегрузки

- В сигнатуру входят: название метода, количество, тип, порядок и модификаторы аргументов
- В классе нельзя определить два метода с одинаковой сигнатурой
- В сигнатуру не входит возвращаемый тип!

```
abstract public double Sum(int a, int b);
```

```
abstract public double Sum(double a, double b);
```

Перегрузка
методов

```
abstract public int Sum(int a, int b, int c);
```

```
abstract public double Sum(int a, int b);
```

```
abstract public int Sum(int a, int b);
```

Конфликт
именования
методов

Методы с произвольным количеством аргументов

```
public static double Average(params double[] numbers)
{
    double total = 0.0;
    foreach (double d in numbers)
    {
        total += d;
    }
    return total / numbers.Length;
}
```

```
Average(5); // выведется 5
Average(2, 4); // выведется 3
```

Методы с произвольным количеством аргументов

- Использовать **params** в списке параметров метода можно только один раз
- При этом такой параметр обязательно должен быть **последним** в списке аргументов

```
public static double Example1(int a, params double[] numbers)
{
    // OK
}
```

```
public static double Example2(params double[] numbers, int a)
{
    // Ошибка компиляции, params double[] numbers
    // должен быть последним параметром
}
```

Необязательные аргументы

- Сигнатура метода может указывать, являются ли его параметры обязательными или нет

```
public static Square CreateSquare(int size, int color)
{
    return new Square(size, color);
}
```

```
public static Square CreateRedSquare(int size)
{
    return new Square(size, Color.Red);
}
```

Методы создания
квадрата с заданным
цветом и красного
квадрата

```
public static Square CreateSquare(int size, int color = Color.Red)
{
    return new Square(size, color);
}
```

Универсальный
метод

```
var redSquare = CreateSquare(5);
var square = CreateSquare(5, Color.BLUE);
```

Пример вызова метода
с необязательными
аргументами

Необязательные аргументы

- Определение каждого необязательного параметра содержит его значение по умолчанию
- Необязательные параметры (их может быть несколько) определяются в конце списка параметров после всех обязательных параметров

Обобщения (универсальные шаблоны)

- Иногда нужно производить одинаковые операции над данными разного типа
- Пример: идентификатор банковского счета может быть любого типа (число, строка и др.)

```
class Account
{
    public int Id { get; set; }
    public int Sum { get; set; }
}
```

Решение 1

- Объявить идентификатор как object

```
class Account
{
    public object Id { get; set; }
    public int Sum { get; set; }
}
```

```
Account acc1 = new Account { Id = 1 };
Account acc2 = new Account { Id = "705b58a" };
int id1 = (int)acc1.Id;
string id2 = (string)acc2.Id;
Console.WriteLine(id1);
Console.WriteLine(id2);
```

Необходимо конвертировать тип идентификатора при каждом обращении, возникает путаница и есть вероятность ошибки

Решение 2

- Использовать обобщения

```
class Account<T>
{
    public T Id { get; set; }
    public int Sum { get; set; }
}
```

Тип идентификатора задается переменной (в данном случае T)

```
Account<int> acc1 = new Account<int> { Id = 1 };
Account<string> acc2 = new Account<string> { Id = "705b58a" };
int id1 = acc1.Id;
string id2 = acc2.Id;
Console.WriteLine(id1);
Console.WriteLine(id2);
```

При инициализации объекта переменная T инициализируется типом (в данном случае int и string), конвертировать типы не нужно

Использование нескольких универсальных параметров

```
class Transaction<U, V>
{
    public U FromAccount { get; set; } // с какого счета перевод
    public U ToAccount { get; set; }   // на какой счет перевод
    public V Code { get; set; }         // код операции
    public int Sum { get; set; }        // сумма перевода
}
```

Наследование обобщенных типов

```
abstract class Account<T>
{
    public T Id { get; private set; }
    protected Account(T id)
    {
        Id = id;
    }
}
```

Обобщенный
базовый класс

```
class StringAccount : Account<string>
{
    public StringAccount(string id) : base(id) { }
}
```

Необобщенный
класс-наследник

```
class UniversalAccount<T> : Account<T>
{
    public UniversalAccount(T id) : base(id) { }
}
```

Класса-наследник,
который
типизирован тем
же типом, что и
базовый

Ограничения обобщений

```
class NumbersCollection<T> where T : ICollection<int>, new()  
{  
    public T Numbers { get; set; } = new T();  
  
    // добавление чисел в коллекцию  
    public void Add(int i)  
    {  
        Numbers.Add(i);  
    }  
  
    // вывод чисел  
    public void Print()  
    {  
        foreach (int v in Numbers)  
        {  
            Console.WriteLine(v);  
        }  
    }  
}
```

С помощью выражения `where T : ICollection<int>` мы указываем, что используемый тип `T` обязательно должен реализовывать интерфейс `ICollection<int>` и соответственно обращаться к `Numbers` как к коллекции.

Стандартное ограничение `new()` позволяет создавать новые экземпляры универсального типа `T` с помощью общедоступного (`public`) конструктора без параметров.

Ограничения обобщений

// создание коллекции на основе списка

```
var numbersListCollection = new NumbersCollection<List<int>>();  
numbersListCollection.Add(5);  
numbersListCollection.Add(1);  
numbersListCollection.Add(2);  
numbersListCollection.Print(); // выведется 5 1 2
```

// создание коллекции на основе отсортированного множества

```
var numbersSortedSetCollection = new NumbersCollection<SortedSet<int>>();  
numbersSortedSetCollection.Add(5);  
numbersSortedSetCollection.Add(1);  
numbersSortedSetCollection.Add(2);  
numbersSortedSetCollection.Print(); // выведется 1 2 3
```

Обобщенные методы

```
static void Swap<T>(ref T lhs, ref T rhs)
{
    T temp;
    temp = lhs;
    lhs = rhs;
    rhs = temp;
}
```

```
int a = 1, b = 2;
Swap<int>(ref a, ref b); // или Swap(ref a, ref b);
```

```
string s1 = "hello", s2 = "world";
Swap<string>(ref s1, ref s2); // или Swap(ref s1, ref s2);
```


Обобщения в стандартных библиотеках C#

- Обобщенные коллекции (System.Collections.Generic):
 - List<T>
 - Dictionary<TKey, TValue>
 - LinkedList<T>
 - Queue<T>
 - и др.

Пример

```
class Product
{
    public Product(string title, double price, string description)
    {
        Title = title;
        Price = price;
        Description = description;
    }

    public string Title { get; private set; }
    public double Price { get; private set; }
    public string Description { get; private set; }
}

Dictionary<long, Product> products = new Dictionary<long, Product>();
// добавление элементов
products.Add(0, new Product("Хлеб", 30, "Бородинский"));
products.Add(1, new Product("Молоко", 50, "Простоквашино"));
products.Add(2, new Product("Стиральный порошок", 30, "Для цветной стирки"));

// проверка на наличие элемента по заданному ключу
if (products.ContainsKey(0))
{
    // обращение к элементу по ключу
    Console.WriteLine(products[0].Title);
    // удаление элемента по ключу
    products.Remove(0);
}
```

Переопределение методов

- При наследовании нередко возникает необходимость изменить в классе-наследнике функционал метода, который был унаследован от базового класса. В этом случае класс-наследник может переопределять методы и свойства базового класса.

Переопределение методов

- Те методы и свойства, которые мы хотим сделать доступными для переопределения, в базовом классе помечаются модификатором **virtual**. Такие методы и свойства называют виртуальными.
- 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.

Переопределение методов

```
public class A
{
    public virtual void F()
    {
        Console.WriteLine(1);
    }
}
```

```
public class B : A
{
    public override void F()
    {
        Console.WriteLine(2);
    }
}
```

```
A a = new A();
a.F(); // 1
B b = new B();
b.F(); // 2
A c = new B();
c.F(); // 2
```

Для виртуальной функции
вызывается та реализация,
которая определена для
фактического типа объекта,
а не для типа ссылки

Пример – геометрические фигуры

```
public class Shape
{
    public virtual double GetArea()
    {
        return 0; // нет разумной реализации
    }
}
```

```
public class Rectangle : Shape
{
    // опущен код полей и конструктора
    public override double GetArea()
    {
        return width * height;
    }
}
```

Пример – геометрические фигуры

```
//И тогда эти примеры будут работать правильно  
Shape s1 = new Rectangle(10, 2);  
Console.WriteLine(s1.GetArea()); // 20 (вызывается реализация для  
прямоугольника)
```