

Тема 2. Инкапсуляция, наследование и полиморфизм.

Цель занятия:

**Разобраться в основных понятиях об
объектно-ориентированного
программирования (ООП) в C#,.**

Учебные вопросы:

- 1. Инкапсуляция.**
- 2. Наследование.**
- 3. Полиморфизм.**
- 4. Делегаты.**
- 5. Лямбда-выражения.**

1. Инкапсуляция.

Инкапсуляция — это один принципов ООП, который позволяет объединять данные (поля) и методы для работы с ними в одной сущности (классе) и скрывать детали реализации от внешнего кода.

Это обеспечивает защиту данных и управляемый доступ к ним.

Основные аспекты инкапсуляции:

Соккрытие данных. Поля класса скрываются от внешнего доступа с помощью модификаторов доступа, например, **private**. Доступ к данным осуществляется только через свойства или публичные методы, которые могут содержать дополнительную логику для проверки, преобразования или обработки данных.

Контроль доступа. Модификаторы доступа (**private**, **protected**, **public**, **internal**) позволяют управлять тем, где и как можно использовать элементы класса. Это помогает защитить состояние объекта от некорректного использования.

Интерфейс взаимодействия. Внешний код работает с объектом через публичные методы или свойства, а не напрямую с полями. Это позволяет изменять внутреннюю реализацию объекта, не затрагивая внешний код.

Пример инкапсуляции

```
public class Person
{
    // Приватное поле (скрыто от внешнего кода)
    private int age;
    // Публичное свойство с логикой доступа к полю
    public int Age
    {
        get { return age; } // Получение значения
        set
        {
            if (value >= 0 && value <= 120) // Проверка корректности данных
                age = value;
            else
                Console.WriteLine("Возраст должен быть от 0 до 120.");
        }
    }
    // Публичный метод для отображения данных
    public void DisplayAge()
    {
        Console.WriteLine($"Возраст: {age}");
    }
}
```

Проще говоря, **инкапсуляция** — это способ «упаковать» данные (поля) и методы для работы с этими данными в одном месте — в классе, а также скрыть эти данные от внешнего мира, чтобы никто не мог их случайно (или намеренно) изменить неправильно.

Простой пример из жизни.

Представьте, что вы используете банкомат:

Вы вставляете карту, вводите пин-код и снимаете деньги.

При этом вы **не видите**, как банкомат работает внутри: как он проверяет ваш баланс или взаимодействует с банком.

Всё, что вам доступно, — это кнопки (интерфейс) для работы с банкоматом.

Инкапсуляция в программировании работает точно так же:

Программный объект (например, класс) скрывает свои внутренние данные (например, поля) и предоставляет только ограниченные способы работы с ними через методы или свойства.

2. Наследование.

Наследование — это механизм, который позволяет создавать новый класс (производный), унаследовав свойства и методы существующего класса (базового).

Наследование позволяет создавать производные (дочерние) классы на основе базового (родительского) класса.

Производный класс наследует свойства, методы и другие члены базового класса, а также может добавлять свои собственные или переопределять унаследованные.

Базовые и производные классы.

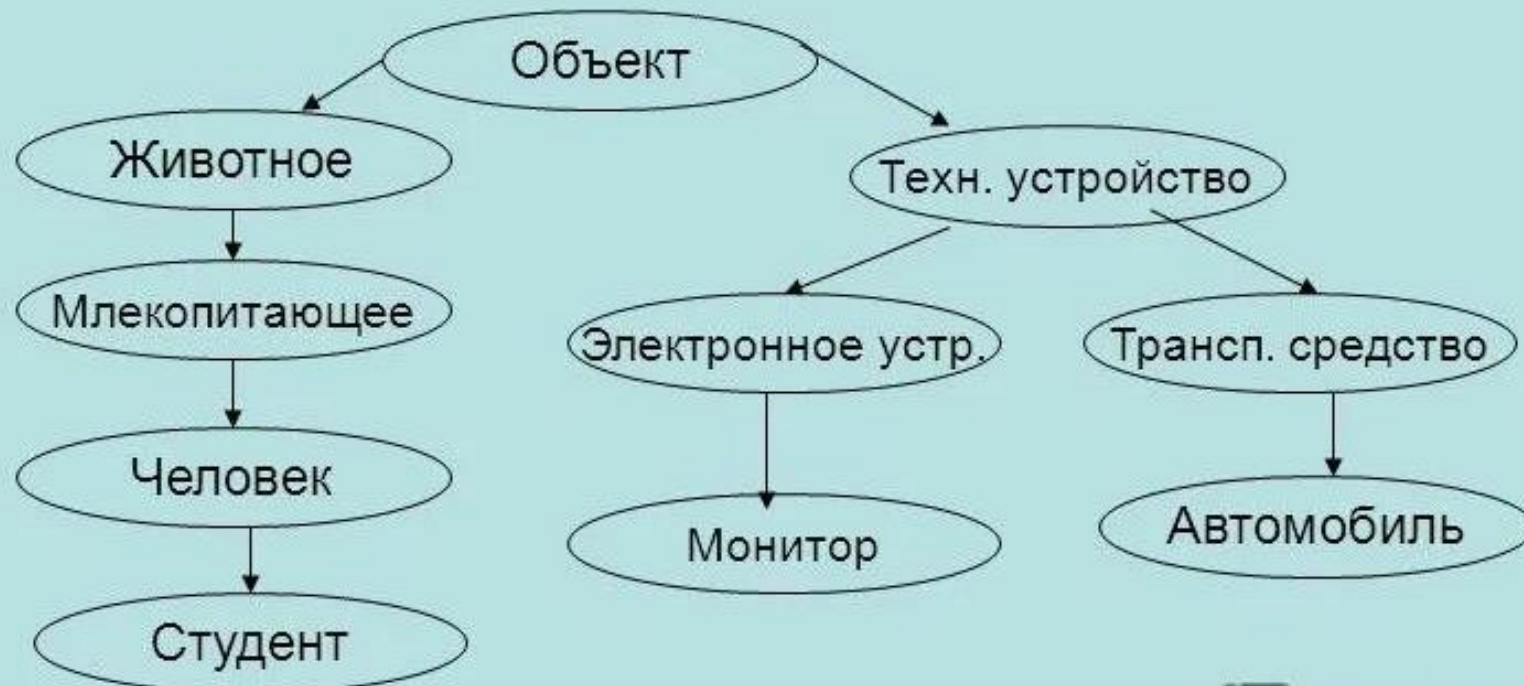
Базовый класс — это класс, от которого наследуются. Он содержит общие свойства и методы, которые могут использоваться в производных классах.

Производный класс — это класс, который наследует функциональность базового класса, добавляя или изменяя поведение.

Иерархия наследования

Классы могут быть организованы в иерархическую структуру с наследованием свойств.

Дочерний класс (подкласс) наследует атрибуты родительского класса (надкласса), расположенного выше в иерархическом дереве.



// Базовый класс

Ссылка: 1

```
public class Animal
```

```
{
```

Ссылка: 3

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

Ссылка: 1

```
public void Eat()
```

```
{
```

```
    Console.WriteLine($"{Name} ест.");
```

```
}
```

```
}
```

// Производный класс

Ссылка: 2

```
public class Dog : Animal
```

```
{
```

Ссылка: 1

```
    public void Bark()
```

```
    {
```

```
        Console.WriteLine($"{Name} лает.");
```

```
    }
```

```
}
```

```
Dog dog = new Dog();
```

```
dog.Name = "Sharik";
```

```
dog.Eat(); // Использование метода базового класса
```

```
dog.Bark(); // Метод производного класса
```

Ключевое слово **base**.

Ключевое слово **base** используется для:

- Обращения к членам базового класса (например, методам, свойствам).
- Вызова конструктора базового класса из конструктора производного класса.

Обращение к методам или свойствам базового класса.

Если производный класс переопределяет метод или свойство базового класса, **base** позволяет вызвать оригинальную версию из базового класса.

Вызов конструктора базового класса

Производный класс обязан вызывать конструктор базового класса, если базовый класс требует передачи параметров в свой конструктор.

// Базовый класс

Ссылка: 3

```
public class Animal
```

```
{
```

Ссылка: 2

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

Ссылка: 1

```
    public Animal(string name) // Конструктор базового класса
```

```
    {
```

```
        Name = name;
```

```
    }
```

```
}
```



```
// Производный класс
```

```
Ссылка: 3
```

```
public class Dog : Animal
```

```
{
```

```
Ссылка: 2
```

```
public string Breed { get; }
```

```
Ссылка: 1
```

```
public Dog(string name, string breed) : base(name) // Вызов конст-ра базового класса
```

```
{
```

```
    Breed = breed;
```

```
}
```

```
Ссылка: 0
```

```
public void DisplayInfo()
```

```
{
```

```
    Console.WriteLine($"Собака: {Name}, порода: {Breed}");
```

```
}
```

```
}
```

```
Dog dog = new Dog("Шарик", "Овчарка");  
dog.DisplayInfo();
```

Важные моменты наследования:

- Один базовый класс. В C# класс может наследоваться только от одного базового класса. Это называется **одиночным наследованием**.
- **Переопределение методов**. Методы базового класса могут быть переопределены в производном классе, если они отмечены ключевым словом **virtual**. Производный класс использует **override** для переопределения.
- Модификатор доступа **protected**. Члены с модификатором **protected** доступны только в самом классе и его производных классах, но недоступны извне.

```
// Базовый класс
class Animal
{
    // Виртуальный метод, который может быть переопределен в производных классах
    public virtual void MakeSound()
    {
        Console.WriteLine("The animal makes a sound");
    }
}
```

```
// Производный класс
class Dog : Animal
{
    // Переопределение метода MakeSound
    public override void MakeSound()
    {
        Console.WriteLine("The dog barks");
    }
}
```

```
// Производный класс
class Cat : Animal
{
    // Переопределение метода MakeSound
    public override void MakeSound()
    {
        Console.WriteLine("The cat meows");
    }
}
```

```
class Program
{
    static void Main(string[] args)
    {
        // Создаем объекты базового и производных классов
        Animal myAnimal = new Animal();
        Animal myDog = new Dog();
        Animal myCat = new Cat();

        // Вызываем метод MakeSound для каждого объекта
        myAnimal.MakeSound(); // Вывод: The animal makes a sound
        myDog.MakeSound();    // Вывод: The dog barks
        myCat.MakeSound();    // Вывод: The cat meows
    }
}
```

Пояснение:

Базовый класс Animal:

- Метод MakeSound объявлен как **virtual**, что позволяет производным классам переопределять его.

Производный класс Dog:

- Метод MakeSound переопределен с использованием ключевого слова **override**. Теперь, когда метод вызывается для объекта типа Dog, будет выполняться переопределенная версия метода.

Производный класс Cat:

- Аналогично, метод MakeSound переопределен для класса Cat.

Метод Main:

- Создаются объекты базового и производных классов.
- При вызове метода MakeSound для каждого объекта выполняется соответствующая версия метода, в зависимости от типа объекта.

3. Полиморфизм.

Полиморфизм — это одна из ключевых концепций объектно-ориентированного программирования (ООП), которая позволяет объектам разных типов обрабатывать запросы через единый интерфейс.

Суть полиморфизма можно выразить фразой: "Один интерфейс — много реализаций."

Пример: "Один интерфейс — много реализаций"

Представим, что у нас есть базовый класс **Shape** (фигура) и несколько производных классов: **Circle** (круг) и **Rectangle** (прямоугольник).

Каждая форма должна уметь вычислять свою площадь, но реализация этого метода будет разной.

//базовый класс

Ссылка: 2

```
public class Shape
```

```
{
```

Ссылка: 2

```
public virtual double GetArea() // Виртуальный метод
```

```
{
```

```
    return 0;
```

```
}
```

```
}
```

```
public class Circle : Shape
{
    // Ссылка: 3
    public double Radius { get; set; }
    // Ссылка: 0
    public Circle(double radius)
    {
        Radius = radius;
    }
    // Ссылка: 1
    public override double GetArea() // Переопределение метода
    {
        return Math.PI * Radius * Radius;
    }
}
```

```
public class Rectangle : Shape
{
    Ссылка: 2
    public double Width { get; set; }
    Ссылка: 2
    public double Height { get; set; }
    Ссылка: 0
    public Rectangle(double width, double height)
    {
        Width = width;
        Height = height;
    }
    Ссылка: 1
    public override double GetArea() // Переопределение метода
    {
        return Width * Height;
    }
}
```

```
Circle circle = new Circle(5);  
Rectangle rectangle = new Rectangle(4, 6);  
  
// Площадь круга:  
Console.WriteLine($"Площадь круга: {circle.GetArea()}");  
// Площадь прямоугольника:  
Console.WriteLine($"Площадь прямоугольника: {rectangle.GetArea()}");
```

В этом примере оба объекта (Circle и Rectangle) имеют общий интерфейс — метод `GetArea()`.

Благодаря полиморфизму вызывается реализация, соответствующая конкретному объекту.

Важный момент реализации полиморфизма.

Базовый класс должен объявить метод с ключевым словом **virtual**, чтобы его можно было переопределить.

Производный класс переопределяет метод с помощью **override**.

// Базовый класс

Ссылка: 1


```
public class Animal
{
    Ссылка: 2
    public virtual void MakeSound()
    {
        Console.WriteLine("Животное издаёт звук.");
    }
}
```

// Производный класс

Ссылка: 2

```
public class Dog : Animal
{
    Ссылка: 2
    public override void MakeSound()
    {
        base.MakeSound(); // Вызов метода базового класса
        Console.WriteLine("Собака лает.");
    }
}
```

```
Dog dog = new Dog();  
dog.MakeSound();
```

 Консоль отладки Microsoft Visual Studio

```
Животное издаёт звук.  
Собака лает.
```

Ключевое слово **virtual**:

- Указывает, что метод базового класса может быть переопределён в производном классе.
- Используется только для методов, свойств и событий.
- Без **virtual** метод базового класса не может быть переопределён.

Ключевое слово **override**

- Указывает, что метод в производном классе переопределяет метод базового класса.
- Метод должен иметь ту же сигнатуру, что и метод базового класса.
- Производный метод заменяет поведение метода базового класса.

4. Делегаты.

Делегаты в C# — это типы, которые представляют ссылки на методы.

Они позволяют передавать методы как параметры в другие методы, что является основой для функционального программирования и гибкости кода.

1. Action

Action — это встроенный обобщённый делегат, который используется для представления методов, не возвращающих значения (`void`), но принимающих до 16 входных параметров.

Он позволяет вам передавать методы как параметры или хранить ссылки на них.

Определён в пространстве имен `System`.

Пример:

```
static void PrintMessage(string message) // Объявление метода PrintMessage
{
    Console.WriteLine($"Сообщение: {message}");
}
```

Ссылка: 0

```
static void Main()
{
    // Создаётся делегат action, который получает ссылку на метод PrintMessage
    Action<string> action = PrintMessage;
    // Вызов делегата
    action("Привет, мир!"); // Вывод: Сообщение: Привет, мир!
}
```

2. Func.

Func — это встроенный обобщённый делегат, который используется для представления методов, возвращающих значение и принимающих до 16 входных параметров.

Пример:

```
static int Add(int a, int b)
{
    return a + b;
}
```

Ссылки: 0

```
static void Main()
{
    Func<int, int, int> func = Add; // Метод с 2 параметрами, возвращает int
    Console.WriteLine(func(5, 7));
}
```

3. Predicate.

Predicate — это встроенный обобщённый делегат, который представляет метод, возвращающий логическое значение (bool) и принимающий один входной параметр.

Он часто используется для проверки условий.

Методы, которые принимают предикат в качестве аргумента, предоставляют способ фильтрации, поиска или сортировки элементов в коллекциях на основе заданного условия.

Использование с обычным методом

```
static bool IsEven(int number)
{
    return number % 2 == 0;
}
```

Ссылка: 0

```
static void Main()
{
    Predicate<int> isEvenPredicate = IsEven; // Метод, проверяющий чётность числа
    Console.WriteLine(isEvenPredicate(4)); // true
    Console.WriteLine(isEvenPredicate(5)); // false
}
```

Использование с лямбда-выражением

```
static void Main()
{
    Predicate<string> isLongString = str => str.Length > 5;
    // Лямбда для проверки длины строки
    Console.WriteLine(isLongString("Привет")); // true
    Console.WriteLine(isLongString("Hi"));    // false
}
```


Использование в стандартных методах C#

```
static bool IsGreaterThanFive(int number) // метод является реализацией предиката
{
    return number > 5;
}
Ссылка: 0
static void Main()
{
    List<int> numbers = new List<int> { 1, 3, 7, 10, 2 };

    // Ищем первое число, большее 5
    int result = numbers.Find(IsGreaterThanFive);

    Console.WriteLine(result); // 7
}
```

```
static bool IsNegative(int number)
{
    return number < 0;
}
Ссылка: 0
static void Main()
{
    List<int> numbers = new List<int> { -1, 2, -3, 4, 5 };

    // Удаляем все отрицательные числа
    numbers.RemoveAll(IsNegative);

    Console.WriteLine(string.Join(", ", numbers)); // 2, 4, 5
}
```

Метод RemoveAll: Для каждого элемента в списке вызывает метод IsNegative: Если метод возвращает true, элемент удаляется из списка. Если метод возвращает false, элемент остаётся.

Предикат - лямбда-выражение, которое принимает каждый элемент списка (num) и проверяет, является ли он чётным числом

```
static void Main()
{
    List<int> numbers = new List<int> { 1, 2, 3, 4, 5, 6, 7, 8, 9 };

    // Фильтруем только чётные числа
    List<int> evenNumbers = numbers.FindAll(num => num % 2 == 0);

    Console.WriteLine(string.Join(", ", evenNumbers)); // 2, 4, 6, 8
}
```

Сравнение делегатов:

Делегат	Параметры	Возвращаемое значение
Action	От 0 до 16	void
Func	От 0 до 16 (последний тип — результат)	Любой тип
Predicate	1 параметр	bool

5. Лямбда-выражения.

Лямбда-выражения в C# — это компактный способ представления анонимных методов, которые позволяют писать функции прямо в месте их использования.

Лямбда-выражения особенно полезны при работе с делегатами, **LINQ**-запросами, коллекциями и событиями.

Общий синтаксис лямбда-выражения:

```
(parameters) => expression_or_statement
```

parameters — параметры, которые передаются в лямбду.

expression_or_statement — выражение или блок кода, выполняющийся внутри лямбда-выражения.

=> лямбда-оператор

Лямбда с одним параметром

```
Func<int, int> square = x => x * x;  
Console.WriteLine(square(5)); // Вывод: 25
```

Лямбда с несколькими параметрами

```
Func<int, int, int> add = (x, y) => x + y;  
Console.WriteLine(add(3, 4)); // Вывод: 7
```

Лямбда с блоком кода

```
Func<int, int, int> multiplyAndPrint = (x, y) =>
{
    int result = x * y;
    Console.WriteLine($"Результат умножения: {result}");
    return result;
};

Console.WriteLine(multiplyAndPrint(3, 4)); // Вывод: Результат умножения: 12
```


Лямбда с использованием коллекций

```
List<int> numbers = new List<int> { 1, 2, 3, 4, 5, 6, 7, 8, 9 };  
var evenNumbers = numbers.Where(n => n % 2 == 0).ToList();  
Console.WriteLine(string.Join(", ", evenNumbers)); // Вывод: 2, 4, 6, 8
```

В данном примере используется метод **Where**, который фильтрует чётные числа в списке с помощью лямбда-выражения (предиката) `n => n % 2 == 0`.

Лямбда-выражения часто применяются с методами расширений коллекций, такими как `Where`, `Select`, `OrderBy`, `Sum` и другими, предоставляемыми LINQ.

Лямбда-выражения в LINQ, примеры:

Фильтрация данных (Where)

Лямбда-выражения часто используются для фильтрации элементов коллекции. Например, можно отфильтровать только четные числа из списка:

```
List<int> numbers = new List<int> { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };  
  
var evenNumbers = numbers.Where(n => n % 2 == 0).ToList();  
  
Console.WriteLine(string.Join(", ", evenNumbers));  
// Вывод: 2, 4, 6, 8, 10
```

Проекция данных (Select)

Лямбда-выражения могут использоваться для преобразования элементов коллекции. Например, можно преобразовать список чисел в список их квадратов:

```
List<int> numbers = new List<int> { 1, 2, 3, 4, 5 };  
  
var squares = numbers.Select(n => n * n).ToList();  
  
Console.WriteLine(string.Join(", ", squares));  
// Вывод: 1, 4, 9, 16, 25
```

Сортировка данных (OrderBy, OrderByDescending)

Лямбда-выражения используются для указания ключа сортировки. Например, можно отсортировать список строк по длине:

```
List<string> words = new List<string> { "apple", "banana", "cherry", "date" };  
  
var sortedWords = words.OrderBy(word => word.Length).ToList();  
  
Console.WriteLine(string.Join(", ", sortedWords));  
// Вывод: date, apple, banana, cherry
```

Поиск элементов (First, FirstOrDefault, Single, SingleOrDefault)

Лямбда-выражения используются для поиска элементов, удовлетворяющих определенному условию. Например, можно найти первое четное число в списке:

```
List<int> numbers = new List<int> { 1, 3, 5, 7, 8, 9, 10 };  
  
int firstEvenNumber = numbers.First(n => n % 2 == 0);  
  
Console.WriteLine(firstEvenNumber);  
// Вывод: 8
```

Проверка условий (All, Any)

Лямбда-выражения могут использоваться для проверки, удовлетворяют ли все или хотя бы один элемент коллекции определенному условию. Например, можно проверить, все ли числа в списке положительные:

```
List<int> numbers = new List<int> { 1, 2, 3, 4, 5 };  
  
bool allPositive = numbers.All(n => n > 0);  
  
Console.WriteLine(allPositive);  
// Вывод: True
```

Заключение

Лямбда-выражения — мощный инструмент в C#, который позволяет создавать компактные, удобные и функциональные решения для работы с делегатами, событиями, LINQ и многими другими задачами.

Например. Код с использованием обычного метода:

```
static bool IsGreaterThanFive(int number) // метод является реализацией предиката
{
    return number > 5;
}
Ссылка: 0
static void Main()
{
    List<int> numbers = new List<int> { 1, 3, 7, 10, 2 };

    // Ищем первое число, большее 5
    int result = numbers.Find(IsGreaterThanFive);

    Console.WriteLine(result); // 7
}
```


Код с использованием лямбда-выражения :

```
List<int> numbers = new List<int> { 1, 3, 7, 10, 2 };  
// Ищем первое число больше 5  
int result = numbers.Find(el => el > 5);  
Console.WriteLine(result); // 7
```

Список литературы:

1. [Наследование классов](#)
2. [Объектно-ориентированное программирование. Наследование.](#)
3. [Лямбды](#)
4. [Наследование классов.](#)
5. [Урок по С# #14 Полиморфизм](#)
6. [Уроки С# – Лямбда выражение](#)
7. [Уроки С# – LINQ – Where, Select, GroupBy, AsParallel, x.Key](#)
8. [Введение в ООП](#)

Материалы лекций:

<https://github.com/ShViktor72/Education>

Обратная связь:

colledge20education23@gmail.com

Домашнее задание:

1. Инкапсуляция. Создайте класс Thermostat, который управляет температурой в комнате. В нем должны быть:

Приватные поля:

- `currentTemperature` (текущая температура, `double`)
- `minTemperature` (минимально допустимая температура, `double`)
- `maxTemperature` (максимально допустимая температура, `double`)

Свойство `Temperature` для установки и получения текущей температуры.

Если пользователь пытается установить температуру ниже `minTemperature`, устанавливается `minTemperature`. Если выше `maxTemperature`, устанавливается `maxTemperature`.

Методы:

- `IncreaseTemperature(double value)` – увеличивает температуру на указанное значение.
- `DecreaseTemperature(double value)` – уменьшает температуру на указанное значение.
- `GetStatus()` – возвращает строку "Комфортная температура", если `currentTemperature` в пределах нормы, "Слишком жарко!", если температура максимальная, "Слишком холодно!", если температура минимальная.

2. Наследование

Создайте класс Device (устройство) с полями brand (бренд) и model (модель). Добавьте метод GetInfo(), который возвращает информацию об устройстве.

Создайте классы-наследники:

- Smartphone, добавьте поле os (операционная система) и переопределите метод GetInfo(), чтобы он включал информацию о системе.
- Laptop, добавьте поле ram (оперативная память в ГБ) и переопределите GetInfo(), чтобы включать информацию о памяти.

3. Полиморфизм

Создайте базовый класс MusicalInstrument, который содержит:

- виртуальный метод PlaySound(), выводящий "Издает звук...".
- виртуальное свойство Name, возвращающее "Музыкальный инструмент".

Создайте производные классы:

- Guitar, где PlaySound() выводит "Бринь-бринь!", а Name возвращает "Гитара".
- Piano, где PlaySound() выводит "Пиу-пиу!", а Name возвращает "Пианино".
- Drum, где PlaySound() выводит "Бум-бум!", а Name возвращает "Барабан".