Тема 3 «Объектно-ориентированное программирование. »

Общие принципы

ПОЛЯ И МЕТОДЫ КЛАССА

Для хранения данных в классе применяются **поля**.

Класс может определять некоторое поведение или выполняемые действия. Для определения поведения в классе применяются **методы**.

ПОЛЯ И МЕТОДЫ КЛАССА

```
class Person
{
    public string name = "Undefined";   // имя
    public int age;                     // возраст

    public void Print()
    {
        Console.WriteLine($"Имя: {name}  Возраст: {age}");
    }
}
```

ПОЛЯ И МЕТОДЫ КЛАССА

```
class Person
{
    public string name = "Undefined";   // имя
    public int age;                     // возраст

    public void Print()
    {
        Console.WriteLine($"Имя: {name}  Возраст: {age}");
    }
}
```

ПОЛЯ И МЕТОДЫ КЛАССА

```
string строка1 = "Он сказал \"This is the last \u0063hance\x0021\"";
string строка2 = @"Он сказал ""This is the last \u0063hance\x0021""";

Console.WriteLine(строка1);
Console.WriteLine(строка2);
// Пример выведет следующий текст:
//     Он сказал "This is the last chance!"
//     Он сказал "This is the last \u0063hance\x0021"
```

ПОЛЯ И МЕТОДЫ КЛАССА

```
string строка1 = "Он сказал \"This is the last \u0063hance\x0021\"";
string строка2 = @"Он сказал ""This is the last \u0063hance\x0021""";

Console.WriteLine(строка1);
Console.WriteLine(строка2);
// Пример выведет следующий текст:
//     Он сказал "This is the last chance!"
//     Он сказал "This is the last \u0063hance\x0021"
```

```
string name = "Андрей";
var date = DateTime.Now;

// Вывод с использованием обычного составного форматирования:
Console.WriteLine("Привет, {0}! Сегодня {1}, время {2:HH:mm}.",
                  name, date.DayOfWeek, date);
// Вывод с помощью интерполированной строки:
Console.WriteLine($"Hello, {name}! Today is {date.DayOfWeek},
                    it's {date:HH:mm} now.");
// Оба вызова сгенерируют одинаковый текст:
//    Привет, Андрей! Сегодня среда, время 09:58.
```

СОЗДАНИЕ ОБЪЕКТА КЛАССА

Для создания объекта применяются конструкторы.

```
new конструктор_класса(параметры_конструктора);
```

КОНСТРУКТОР ПО УМОЛЧАНИЮ

```
class Person
{
    public string name = "Undefined";   // имя
    public int age;                     // возраст

    public void Print()
    {
        Console.WriteLine($"Имя: {name}  Возраст: {age}");
    }
}
```

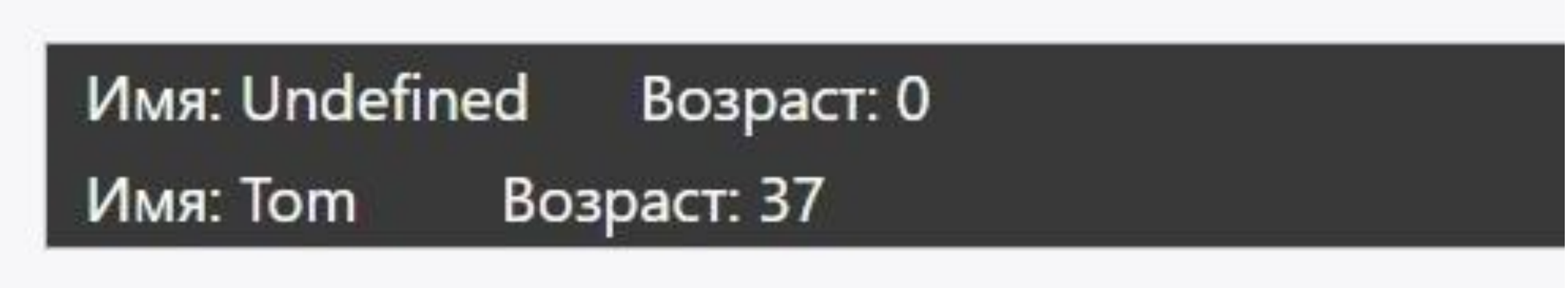
КОНСТРУКТОР ПО УМОЛЧАНИЮ

```
Person tom = new Person();  // создание объекта класса Person

// определение класса Person
class Person
{
    public string name = "Undefined";
    public int age;

    public void Print()
    {
        Console.WriteLine($"Имя: {name}  Возраст: {age}");
    }
}
```

ОБРАЩЕНИЕ К ФУНКЦИОНАЛЬНОСТИ КЛАССА

```
объект.поле_класса
объект.метод_класса(параметры_метода)
Person tom = new Person();  // создание объекта класса Person

// Получаем значение полей в переменные
string personName = tom.name;
int personAge = tom.age;
Console.WriteLine($"Имя: {personName}  Возраст {personAge}");   // Имя: Undefined  Возраст: 0

// устанавливаем новые значения полей
tom.name = "Tom";
tom.age = 37;

// обращаемся к методу Print
tom.Print();    // Имя: Tom  Возраст: 37

class Person
{
    public string name = "Undefined";
    public int age;

    public void Print()
    {
        Console.WriteLine($"Имя: {name}  Возраст: {age}");
    }
}
```

ОБРАЩЕНИЕ К ФУНКЦИОНАЛЬНОСТИ КЛАССА



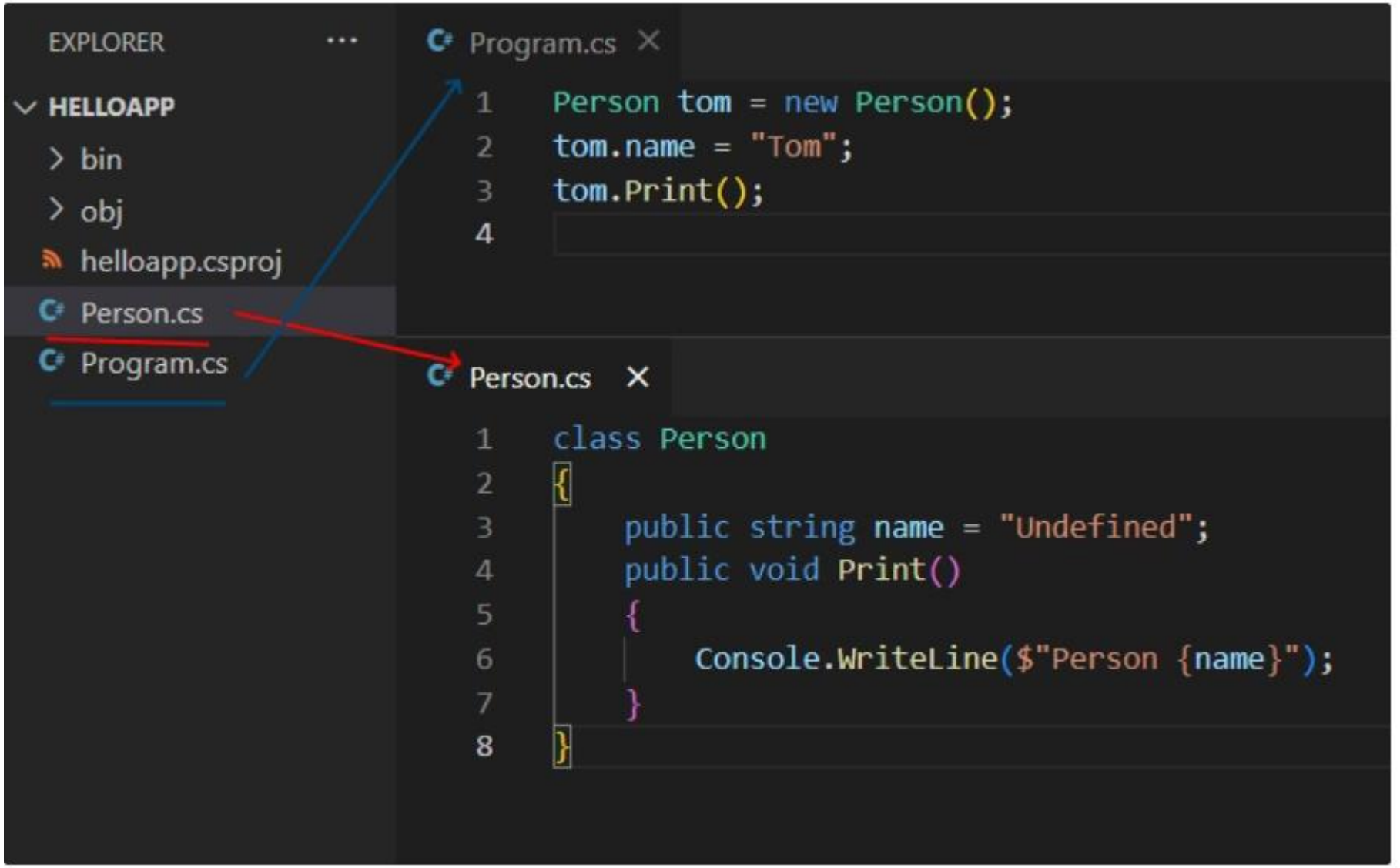
ДОБАВЛЕНИЕ КЛАССА

```
class Person
{
    public string name = "Undefined";
    public void Print()
    {
        Console.WriteLine($"Person {name}");
    }
}
```

ДОБАВЛЕНИЕ КЛАССА

```
Person tom = new Person();
tom.name = "Tom";
tom.Print();    // Person Tom
```

ДОБАВЛЕНИЕ КЛАССА



КОНСТРУКТОРЫ, ИНИЦИАЛИЗАТОРЫ И ДЕКОНСТРУКТОРЫ

СОЗДАНИЕ КОНСТРУКТОРОВ

КОНСТРУКТОР ВЫПОЛНЯЕТ ИНИЦИАЛИЗАЦИЮ ОБЪЕКТА.

```
Person tom = new Person();  // Создание объекта класса Person

tom.Print();    // Имя: Tom  Возраст: 37

class Person
{
    public string name;
    public int age;
    public Person()
    {
        Console.WriteLine("Создание объекта Person");
        name = "Tom";
        age = 37;
    }
    public void Print()
    {
        Console.WriteLine($"Имя: {name}  Возраст: {age}");
    }
}
```

Здесь определен конструктор, который выводит на консоль некоторое сообщение и инициализирует поля класса.

Определив конструктор, мы можем вызвать его для создания объекта Person:

```
Person tom = new Person();  // Создание объекта Person
```

Здесь определен конструктор, который выводит на консоль некоторое сообщение и инициализирует поля класса.

Определив конструктор, мы можем вызвать его для создания объекта Person:

```
Person tom = new Person();  // Создание объекта Person
```

```
Person tom = new Person();          // вызов 1-ого конструктора без параметров
Person bob = new Person("Bob");     //вызов 2-ого конструктора с одним параметром
Person sam = new Person("Sam", 25); // вызов 3-его конструктора с двумя параметрами

tom.Print();          // Имя: Неизвестно  Возраст: 18
bob.Print();          // Имя: Bob  Возраст: 18
sam.Print();          // Имя: Sam  Возраст: 25

class Person
{
    public string name;
    public int age;
    public Person() { name = "Неизвестно"; age = 18; }      // 1 конструктор
    public Person(string n) { name = n; age = 18; }         // 2 конструктор
    public Person(string n, int a) { name = n; age = a; }   // 3 конструктор

    public void Print()
    {
        Console.WriteLine($"Имя: {name}  Возраст: {age}");
    }
}
```

Ключевое слово this

Ключевое слово **this** представляет ссылку на текущий экземпляр/объект класса.

Ключевое слово this

```
Person sam = new("Sam", 25);
sam.Print();          // Имя: Sam  Возраст: 25

class Person
{
    public string name;
    public int age;
    public Person() { name = "Неизвестно"; age = 18; }
    public Person(string name) { this.name = name; age = 18; }
    public Person(string name, int age)
    {
        this.name = name;
        this.age = age;
    }
    public void Print() => Console.WriteLine($"Имя: {name}  Возраст: {age}");
}
```

ЦЕПОЧКА ВЫЗОВА КОНСТРУКТОРОВ

```
class Person
{
    public string name;
    public int age;
    public Person() : this("Неизвестно")    // первый конструктор
    { }
    public Person(string name) : this(name, 18) // второй конструктор
    { }
    public Person(string name, int age)     // третий конструктор
    {
        this.name = name;
        this.age = age;
    }
    public void Print() => Console.WriteLine($"Имя: {name}  Возраст: {age}");
}
```

ЦЕПОЧКА ВЫЗОВА КОНСТРУКТОРОВ

```
public Person(string name) : this(name, 18)
{ }
```

ЦЕПОЧКА ВЫЗОВА КОНСТРУКТОРОВ

```
Person tom = new();
Person bob = new("Bob");
Person sam = new("Sam", 25);

tom.Print();          // Имя: Неизвестно  Возраст: 18
bob.Print();          // Имя: Bob  Возраст: 18
sam.Print();          // Имя: Sam  Возраст: 25

class Person
{
    public string name;
    public int age;
    public Person(string name = "Неизвестно", int age = 18)
    {
        this.name = name;
        this.age = age;
    }
    public void Print() => Console.WriteLine($"Имя: {name}  Возраст: {age}");
}
```

ИНИЦИАЛИЗАТОРЫ ОБЪЕКТОВ

```
Person tom = new Person { name = "Tom", age = 31 };
// или так
// Person tom = new() { name = "Tom", age = 31 };
tom.Print();            // Имя: Tom  Возраст: 31
```

ИНИЦИАЛИЗАТОРЫ ОБЪЕКТОВ

```
Person tom = new Person{ name = "Tom", company = { title = "Microsoft"} };
tom.Print();          // Имя: Tom  Компания: Microsoft

class Person
{
    public string name;
    public Company company;
    public Person()
    {
        name = "Undefined";
        company = new Company();
    }
    public void Print() => Console.WriteLine($"Имя: {name}  Компания: {company.title}");
}

class Company
{
    public string title = "Unknown";
}
```

Деконструкторы

Деконструкторы позволяют выполнить декомпозицию объекта на отдельные части.

Например, пусть у нас есть следующий класс Person:

```
class Person
{
    string name;
    int age;
    public Person(string name, int age)
    {
        this.name = name;
        this.age = age;
    }

    public void Deconstruct(out string personName, out int personAge)
    {
        personName = name;
        personAge = age;
    }
}
```

В этом случае мы могли бы выполнить декомпозицию объекта Person так:

```
Person person = new Person("Tom", 33);

(string name, int age) = person;

Console.WriteLine(name);    // Tom
Console.WriteLine(age);     // 33
```

Деконструкторы

```
Person person = new Person("Tom", 33);

(_, int age) = person;

Console.WriteLine(age);    // 33
```

МОДИФИКАТОРЫ ДОСТУПА

Модификаторы доступа определяют контекст, в котором можно употреблять данную переменную или метод.

МОДИФИКАТОРЫ ДОСТУПА

В языке C# применяются следующие модификаторы доступа:

private: закрытый или приватный компонент класса или структуры. Приватный компонент доступен только **в рамках** своего класса или структуры.

private protected: компонент класса доступен из любого места в своем классе или в производных классах, которые определены в **той же сборке.**

file: применяется к типам, например, классам и структурам. Класс или структура с таким модификатором доступны только из **текущего файла кода.**

МОДИФИКАТОРЫ ДОСТУПА

protected: такой компонент класса доступен из любого места в своем классе или в производных классах. При этом производные классы могут располагаться в **других сборках**.

internal: компоненты класса или структуры доступен из любого места кода **в той же сборке**, однако он недоступен **для других программ и сборок**.

protected internal: совмещает **функционал двух модификаторов** protected и internal. Такой компонент класса доступен из **любого места** в **текущей сборке** и из производных классов, которые могут располагаться в других сборках.

МОДИФИКАТОРЫ ДОСТУПА

public: публичный, общедоступный компонент класса или структуры. Такой компонент доступен из любого места в коде, а также из других программ и сборок.

МОДИФИКАТОРЫ ДОСТУПА

Модификаторы	Текущий класс	Производный класс из текущей сборки	Производный класс из другой сборки	Непроизводный класс из текущей сборки	Непроизводный класс из другой сборки
private		■	■	■	■
private protected			■	■	■
protected				■	■
internal			■		■
protected internal					■
public					

МОДИФИКАТОРЫ ДОСТУПА

```
public class Person
{
    string name;
    public Person(string name)
    {
        this.name = name;
    }
    public void Print() => Console.WriteLine($"Name: {name}");
}
```

МОДИФИКАТОРЫ ДОСТУПА

```
class Phone
{
    struct Camera
    {

    }
}
```

МОДИФИКАТОРЫ В РАМКАХ ТЕКУЩЕГО ПРОЕКТА

```
class State
{
    // все равно, что private string defaultVar;
    string defaultVar ="default";
    // поле доступно только из текущего класса
    private string privateVar = "private";
    // доступно из текущего класса и производных классов, которые определены в этом же проекте
    protected private string protectedPrivateVar = "protected private";
    // доступно из текущего класса и производных классов
    protected string protectedVar = "protected";
    // доступно в любом месте текущего проекта
    internal string internalVar = "internal";
    // доступно в любом месте текущего проекта и из классов-наследников в других проектах
    protected internal string protectedInternalVar = "protected internal";
    // доступно в любом месте программы, а также для других программ и сборок
    public string publicVar = "public";

    // по умолчанию имеет модификатор private
    void Print() => Console.WriteLine(defaultVar);
```

МОДИФИКАТОРЫ В РАМКАХ ТЕКУЩЕГО ПРОЕКТА

```
// метод доступен только из текущего класса
private void PrintPrivate() => Console.WriteLine(privateVar);

// доступен из текущего класса и производных классов, которые определены в этом же проекте
protected private void PrintProtectedPrivate() => Console.WriteLine(protectedPrivateVar);

// доступен из текущего класса и производных классов
protected void PrintProtected() => Console.WriteLine(protectedVar);

// доступен в любом месте текущего проекта
internal void PrintInternal() => Console.WriteLine(internalVar);

// доступен в любом месте текущего проекта и из классов-наследников в других проектах
protected internal void PrintProtectedInternal() => Console.WriteLine(protectedInternalVar);

// доступен в любом месте программы, а также для других программ и сборок
public void PrintPublic() => Console.WriteLine(publicVar);
```

МОДИФИКАТОРЫ В РАМКАХ ТЕКУЩЕГО ПРОЕКТА

```
class StateConsumer
{
    public void PrintState()
    {
        State state = new State();

        state.Print(); //

        state.PrintPrivate(); //

        state.PrintProtectedPrivate(); //

        state.PrintProtected(); //

        state.PrintInternal();    //

        state.PrintProtectedInternal();  //

        state.PrintPublic();      //
    }
}
```

МОДИФИКАТОРЫ В РАМКАХ ТЕКУЩЕГО ПРОЕКТА

```
class StateConsumer
{
    public void PrintState()
    {
        State state = new State();

        state.Print(); //Ошибка, получить доступ нельзя

        state.PrintPrivate(); // Ошибка, получить доступ нельзя

        state.PrintProtectedPrivate(); // Ошибка, получить доступ нельзя

        state.PrintProtected(); // Ошибка, получить доступ нельзя

        state.PrintInternal();    // норм

        state.PrintProtectedInternal();  // норм

        state.PrintPublic();      // норм
    }
}
```

ФАЙЛ КАК ОБЛАСТЬ ВИДИМОСТИ

```
file class Person
{
}
```

СВОЙСТВА

Определение свойств

```
[модификаторы] тип_свойства название_свойства
{
    get { действия, выполняемые при получении значения свойства}
    set { действия, выполняемые при установке значения свойства}
}
```

Определение свойств

Полное определение свойства содержит два блока: **get** и **set**.

Блоки **get** и **set** еще называются **акссесорами** или методами доступа (к значению свойства), а также **геттером** и **сеттером**

Определение свойств

```
Person person = new Person();

// Устанавливаем свойство - срабатывает блок Set
// значение "Tom" и есть передаваемое в свойство value
person.Name = "Tom";

// Получаем значение свойства и присваиваем его переменной - срабатывает блок Get
string personName = person.Name;
Console.WriteLine(personName);  // Tom

class Person
{
    private string name = "Undefined";

    public string Name
    {
        get
        {
            return name;    // возвращаем значение свойства
        }
        set
        {
            name = value;   // устанавливаем новое значение свойства
        }
    }
}
```

Определение свойств

Через это свойство мы можем управлять доступом к переменной name. В свойстве в блоке **get** возвращаем значение поля:

```
get { return name; }
```

А в блоке **set** устанавливаем значение переменной name. Параметр value представляет передаваемое значение, которое передается переменной name.

```
set { name = value; }
```