

ЛК 9 . Инкапсуляция

Инкапсуляция

СВОЙСТВА

Свойства

Свойства — это специальные методы, обеспечивающие доступ к полям класса.

С точки зрения клиентского кода свойство ведет себя как поле.

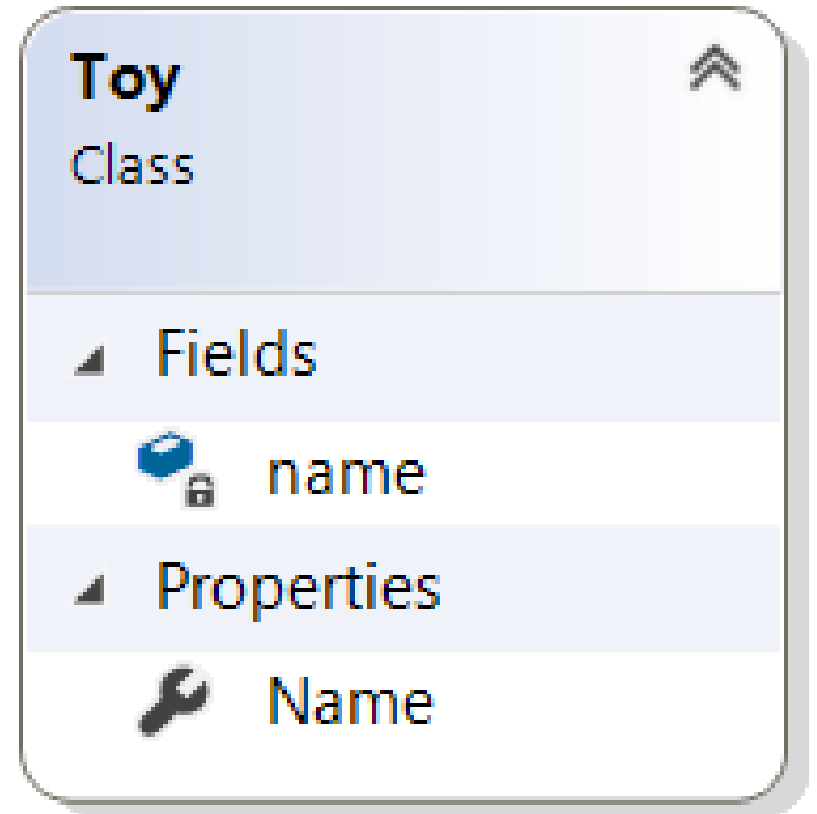
Свойства

Описание свойства имеет следующий вид:

```
[модификатор доступа] тип имя_свойства  
{  
    get {код чтения значения}  
    set {код записи значения}  
}
```

СВОЙСТВА

```
class Toy
{
    private string name;
    public string Name
    {
        get { return name; }
        set { name = value; }
    }
}
```



Свойства

```
var toy = new Toy();  
toy.Name = "Моя игрушка"; // вызывается аксессор set  
Console.WriteLine(toy.Name); // вызывается аксессор get
```

Свойства

Свойства позволяют вложить дополнительную логику в кодовых блоках аксессоров (get и set), которая может быть необходима при присвоении переменной класса какого-либо значения либо при чтении.

СВОЙСТВА

```
class Car
{
    int speed;
    int maxSpeed;
    public int Speed {
        get { return speed; }
        set
        {
            speed = value >= maxSpeed
                ? maxSpeed
                : value;
        }
    };
}
```


Свойства

Используя
модификаторы
доступа,
можно
управлять
свойствами:

- создать свойство только для чтения – клиент сможет только получать значение, но не изменять его
- свойство только для записи – клиент сможет записывать значение, но не сможет прочитать

СВОЙСТВА

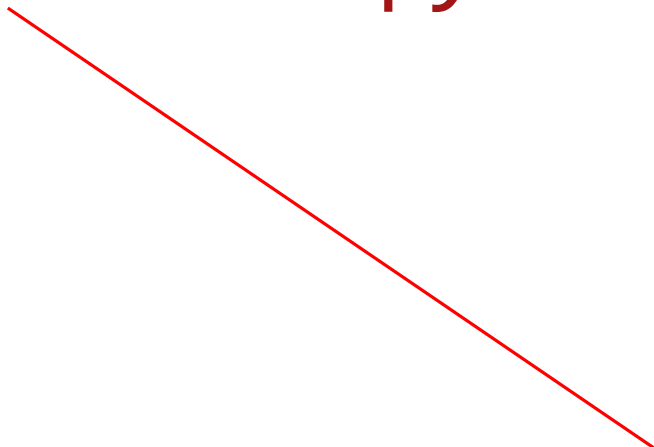
```
class Toy
{
    private string name;
    public string Name
    {
        get { return name; }
        private set { name = value; }
    }
}
```

СВОЙСТВА

```
class Toy
{
    private string name;
    public string Name
    {
        get { return name; }
        // private set { name = value; }
    }
}
```

Свойства

```
var toy = new Toy();  
toy.Name = "Моя игрушка";
```




Ошибка!
Свойство Name доступно
только для чтения

Автоматические свойства

Автоматические свойства — это сокращенная запись свойств вида:

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

В этом случае компилятор сам создает поля для свойств и методы доступа к ним



Автоматические свойства

Автоматические свойства можно сразу инициализировать значением:

```
public string Name { get; set; } = "George";
```

Автоматические свойства

Аксесоры автоматических свойств могут иметь модификаторы доступа:

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

Автоматические свойства

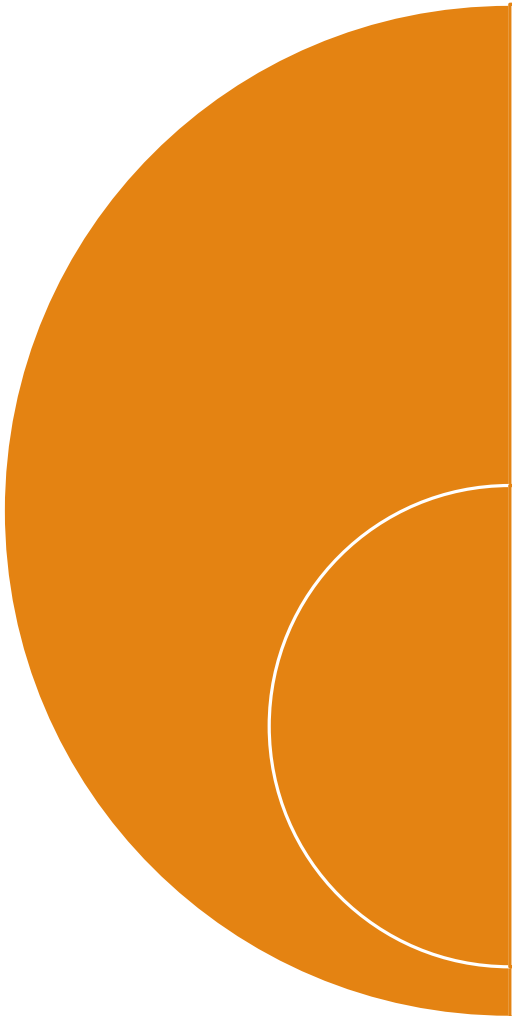
Пример автоматического свойства, доступного только для чтения:

```
public string Name { get; } = "George";
```


Классы

ПЕРЕГРУЗКА МЕТОДОВ

Перегрузка методов



Перегрузкой методов называется использование нескольких методов с одним и тем же именем, но различными типами и количеством параметров.

Компилятор определяет, какой именно метод требуется вызвать, по типу и количеству фактических параметров

Перегрузка методов

```
void MyMeth()  
{ }
```

```
int MyMeth()  
{  
    return ...;  
}
```

```
int MyMeth(int a, string s)  
{  
    return ...;  
}
```

При вызове метода компилятор выбирает вариант, **соответствующий типу и количеству** передаваемых в метод аргументов.

Если точного соответствия не найдено, выполняются **неявные преобразования типов** в соответствии с общими правилами.

Если преобразование невозможно, выдается сообщение об ошибке.

Если выбор перегруженного метода возможен более чем одним способом, то **выбирается «лучший» из вариантов** (вариант, содержащий меньшее количество и длину преобразований в соответствии с правилами преобразования типов).

Если существует несколько вариантов, из которых невозможно выбрать лучший, выдается **сообщение об ошибке**.

Перегрузка методов

Задача.

В классе Car есть метод SpeedUp(), увеличивающий скорость на 20 км/ч.

```
public int SpeedUp()
{
    speed += 20;
    // Если полученная скорость больше максимальной
    if (speed > MaxSpeed)
    {
        speed = MaxSpeed;
    }
    ShowInfo();
    return speed;
}
```

Перегрузка методов

Требуется описать перегруженный метод `SpeedUp`, который увеличивает скорость на произвольную заданную величину.

Перегрузка методов

```
public int SpeedUp(int step)
{
    speed += step;
    // Если полученная скорость больше максимальной
    if (speed > MaxSpeed)
    {
        speed = MaxSpeed;
    }
    ShowInfo();
    return speed;
}
```

Перегрузка методов

Теперь у нас два метода SpeedUp:

```
public int SpeedUp()
```

и

```
public int SpeedUp(int step)
```


Перегрузка методов

Чтобы не было дублирования кода, перепишем первый метод SpeedUp:

```
public int SpeedUp()  
{  
    return SpeedUp(20);  
}
```

Классы

РЕКУРСИВНЫЕ МЕТОДЫ

Рекурсивные методы

Рекурсивным называют метод,
если он вызывает сам себя

Рекурсивные методы

Классический пример рекурсии – вычисление факториала.
Формально алгоритм выглядит так:

$$F(n) = n * F(n-1)$$

Рекурсивные методы

```
static int Factorial(int n)
{
    return n == 0
        ? 1
        : n*Factorial(n - 1);
}
```

Рекурсивные методы

Со входом в рекурсию осуществляется вызов метода, а для выхода необходимо помнить точку возврата, т.е. то место программы откуда мы пришли и куда нам нужно будет возвратиться после завершения метода.

Место хранения точек возврата называется **стеком вызовов** и для него выделяется определенная область оперативной памяти.

В этом стеке запоминаются не только адреса точек возврата, но и **копии** значений всех параметров.

По этим копиям восстанавливается при возврате вызывающий метод.

Рекурсивные методы

Недостатки рекурсии

- При развертывании рекурсии за счет создания копий параметров **возможно переполнение стека.**
- Сложность отлавливания ошибок

Классы

КОНСТРУКТОРЫ КЛАССОВ

Конструкторы классов

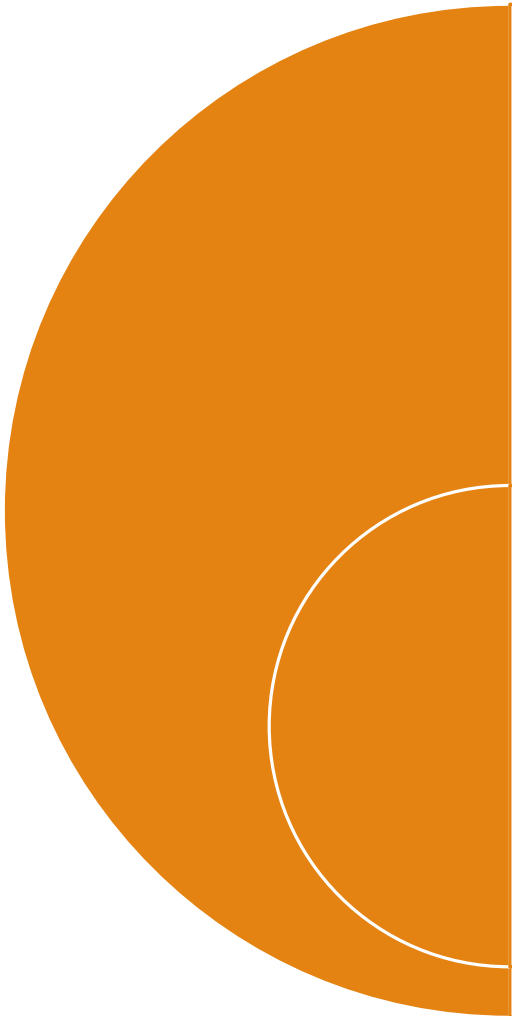


Конструктор – особый метод класса, используемый при создании новых объектов данного класса.

Конструктор всегда имеет то же имя, что и класс.

Конструктор никогда не имеет возвращаемого значения.

Конструкторы классов



Если класс не имеет ни одного конструктора, компилятор создает конструктор по умолчанию.

Добавление хотя бы одного конструктора отменяет создание конструктора по умолчанию

Конструкторы классов

```
class Toy
{
    public string Name { get; private set; }
    public decimal Price { get; set; }

    public Toy(string name)
    {
        Name = name;
    }
}
```

Конструкторы классов

```
var toy = new Toy();
```

Ошибка!
У класса Toy нет
конструктора по умолчанию


```
var toy=new Toy("Teddy the Bear")
```

Ключевое слово `this`

Ключевое слово `this` представляет ссылку на текущий экземпляр класса.

Ключевое слово this

```
private DateTime production;  
  
public Toy(DateTime production)  
{  
    this.production = production;  
}
```



Перегрузка конструкторов

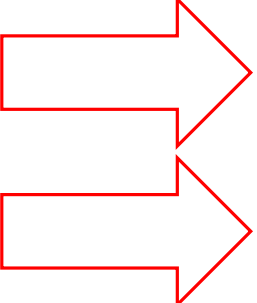
Как и любой метод класса,
конструкторы можно
перегружать.

Перегрузка конструкторов

```
class Toy
{
    public string Name { get; private set; }
    public decimal Price { get; set; }

    public Toy(string name) => Name = name;

    public Toy(string name, decimal price)
    {
        Name = name; Price = price;
    }
}
```


Two red arrows pointing to the constructor definitions. The first arrow points to the single-parameter constructor, and the second arrow points to the two-parameter constructor.

Перегрузка конструкторов


```
class Toy
{
    public string Name { get; private set; }
    public decimal Price { get; set; }

    public Toy(string name) => Name = name;

    public Toy(string name, decimal price): this(name)
    {
        Price = price;
    }
}
```



Статический конструктор



Статические конструкторы не должны иметь модификатор доступа и не принимают параметров

В статических конструкторах нельзя использовать ключевое слово `this` для ссылки на текущий объект класса и можно обращаться только к статическим членам класса

Статические конструкторы нельзя вызвать в программе вручную. Они выполняются автоматически при самом первом создании объекта данного класса или при первом обращении к его статическим членам (если таковые имеются)

Статический конструктор

```
class Toy
{
    . . .

    static Toy()
    {
        Console.WriteLine("Создан объект класса Toy");
    }

    . . .
}
```

Статический конструктор

```
var toy1 = new Toy("Teddy the Bear");  
var toy2 = new Toy("Barbie Doll");
```



Microsoft Visual Studio Debug Conso

Создан объект класса Toy

D:\Temp\ConsoleApp3\ConsoleApp3\

Инициализация свойств через конструктор

Явное указание параметров конструктора

```
class Toy
{
    private DateTime production;
    public string Name { get; private set; }
    public decimal Price { get; set; }

    public Toy(string name, decimal price):this(name)
    {
        Price = price;
    }
}
```

Явное указание параметров конструктора

```
var toy = new Toy(price:200, name:"Pan Cake");
```

Инициатор свойств

```
var toy = new Toy  
        { Name = "NewName", Price = 100 };
```


Индексаторы

Индексаторы

Индексатор – это особый вид свойства, который позволяет работать с классом или структурой таким образом, как если бы это были массивы.

Индексация класса выполняется по индексу, указываемому как параметр.

Иногда классы, используемые как индексаторы, называют классами-индексаторами.

Индексаторы

```
[атрибуты][модификаторы]    min    this    [[атрибуты]  
min_параметра идентификатор_параметра .,...]  
    {set; get;}
```

Индексатор должен иметь как минимум один параметр.
Тип и идентификатор параметра указываются в квадратных скобках после ключевого слова **this**.

Индексаторы

```
public decimal this[int key]
{
    get { . . . };
    set { . . . };
}
```

Индексаторы

Значение индексатора не классифицируется как переменная, поэтому не допускается передача значения индексатора как параметра `ref` или `out`.

Индексаторы

```
public class Price
{
    private decimal _basePrice;
    public Price(decimal basePrice)
    {
        _basePrice = basePrice;
    }
    . . .
}
```

Индексаторы

```
public decimal this[DayOfWeek key]
{
    get
    {
        switch (key)
        {
            case DayOfWeek.Monday:
            case DayOfWeek.Tuesday:
            case DayOfWeek.Thursday:
            case DayOfWeekk.Sunday: return _basePrice;
            case DayOfWeek.Wednesday:
            case DayOfWeek.Saturday: return _basePrice * 0.9M;
            case DayOfWeek.Friday: return _basePrice * 0.7M;
            default: throw (new IndexOutOfRangeException());
        }
    }
}
```

Индексаторы

```
var price = new Price(100);  
var today = DateTime.Now.DayOfWeek;  
Console.WriteLine(  
    $"Сегодня это стоит {price[today]} рублей");
```


Шаблон проектирования Singleton (одиночка)

Шаблон проектирования Singleton

Singleton - порождающий паттерн, который гарантирует, что для определенного класса будет создан только один объект.

Шаблон проектирования Singleton

```
class SingletonDemo
{
    private static SingletonDemo instance;

    private SingletonDemo()
    { }

    public static SingletonDemo GetInstance()
    {
        if (instance==null)
        {
            instance = new SingletonDemo();
        }
        return instance;
    }
}
```

Деструкторы

Деструктор

Деструктор - метод, вызываемый после удаления объекта - может использоваться для очистки ресурсов, используемых объектом.

Деструктор

```
class Station
{
    ~Station()
    {
        Console.WriteLine("Станция уничтожена");
    }
}
```

Деструктор

- ☐ Деструкторы применяются только в классах.
- ☐ Деструктор класса не имеет возвращаемого типа.
- ☐ В структурах определение деструктора невозможно. Класс может иметь только один деструктор.
- ☐ Деструкторы не могут наследоваться или перегружаться.
- ☐ Деструкторы невозможно вызвать. Они запускаются автоматически.
- ☐ Деструктор не принимает модификаторы и не имеет параметров.