В.В. Подбельский

Использованы иллюстрации пособия Daniel Solis, Illustrated C#

Иллюстрации к курсу лекций по дисциплине «Программирование на C#»

09. Часть 1

Подробнее об ООП в C#

Благодарность Д.Д. Сагалову за помощь в переработке презентаций

Вспомним Виды Членов Типов Данных

В данной лекции будут подробнее разбираться некоторые члены типов данных:

- Данные (readonly-поля и константы);
- Функциональные (свойства, индексаторы, деструкторы).

Данные	Функциональные Члены	
Поля*	***Методы****	Операции
Константы	Свойства	Индексаторы
	Конструкторы	События
	Деструкторы (Финализаторы)	

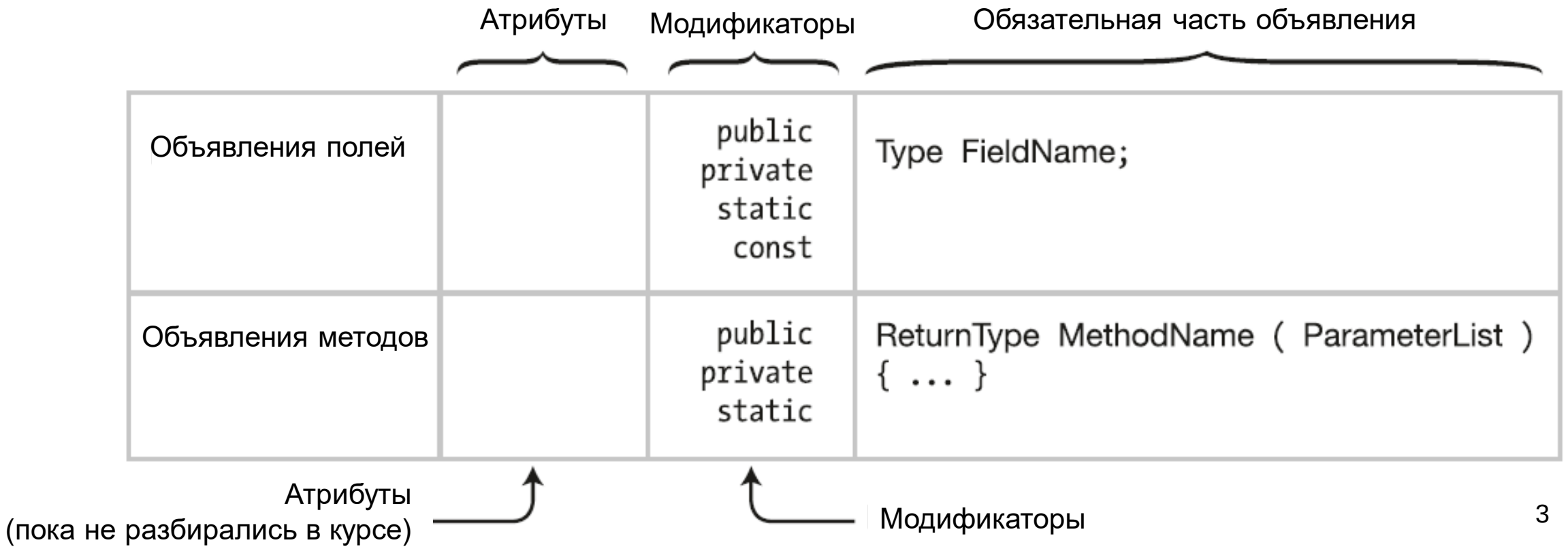
* существуют на уровне IL

Порядок Модификаторов в Объявлениях

Синтаксис объявления члена класса:

[Атрибуты] [Модификаторы] <Обязательная часть объявления>

Помните: порядок модификаторов не важен (public static то же, что и static public)



Константы как Члены Класса

```
class MyClass1      {
    const int IntVal = 100; // определяем константу типа int со значением 100.
}

class MyClass2      {
    const int IntVal1 = 100;
    const int IntVal2 = 2 * IntVal1;    // ок, т.к. IntVal1 определена
}

class MyClass3      {
      const int IntVal;   // ошибка: нет инициализации.
      IntVal = 100;       // ошибка : ошибочное присваивание.
}
```

Константы, как члены класса (2)

```
using System;

class X
{
    public const double PI = 3.1416;
}

class Program {
    static void Main()
    {
        Console.WriteLine($"pi = {X.PI}"); // используем константу PI
    }
}
```

Вывод:
Pi = 3,1416

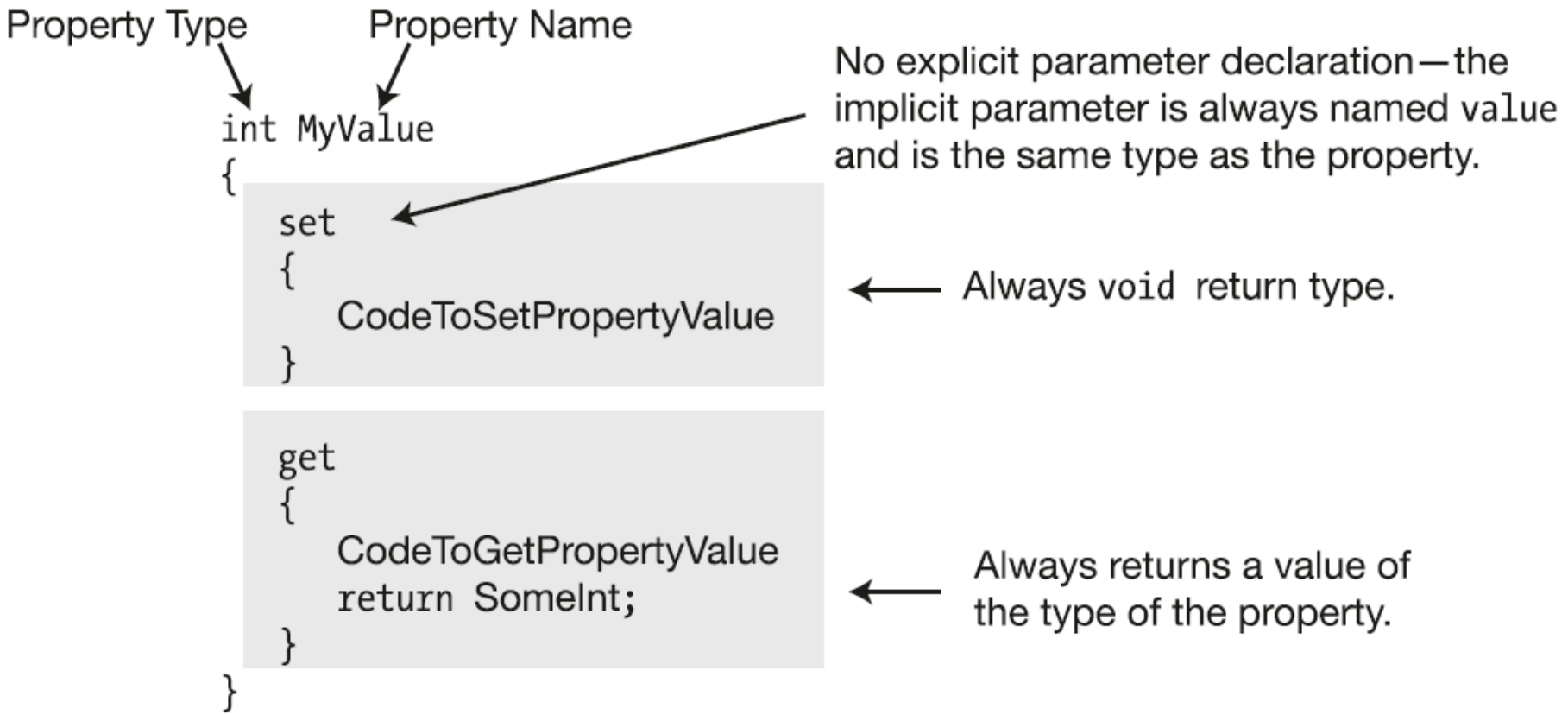
Свойства и внешняя похожесть на поля

```
MyClass mc = new MyClass();
mc.myField = 5; // присваивание значения полю
mc.MyValue = 10; // присваивание значения свойству
Console.WriteLine($"{mc.myField} {mc.MyValue}"); // свойство и поле
```

Свойство, как функциональный член класса

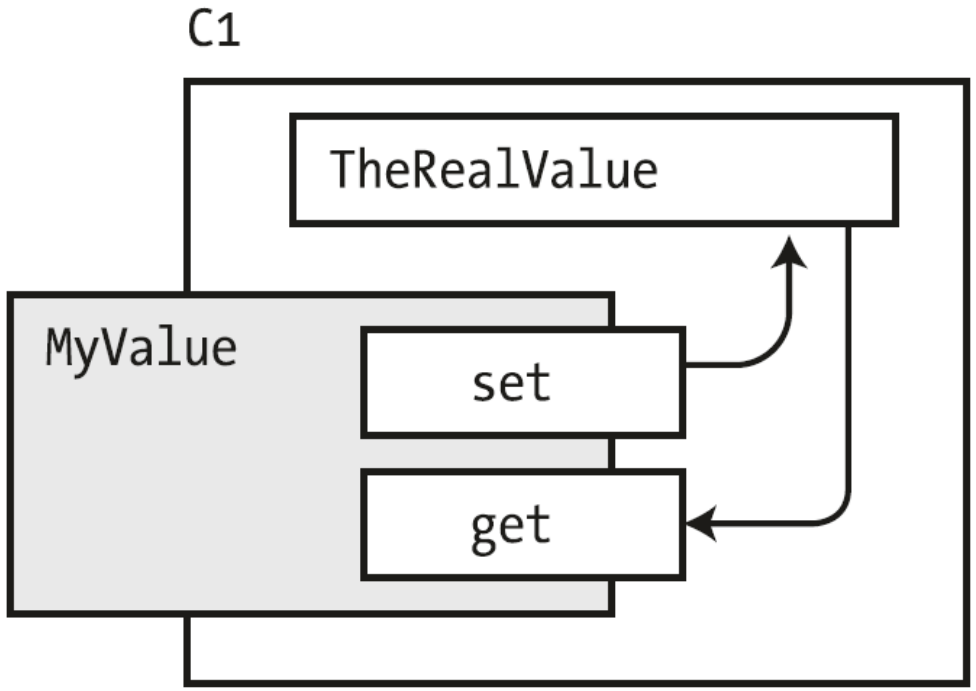
```
int MyValue
{
    set
    {
        SetAccessorCode
    }
    get
    {
        GetAccessorCode
    }
}
```

Объявление свойства и аксессоры



Пример объявления свойства. Инкапсуляция

```
class C1
{
    private int _theRealValue = 10;
    public int MyValue
    {
        set { _theRealValue = value; }
        get { return _theRealValue; }
    }
}
```



В C# 9 появился аксессор **init** (для инициализации):
https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/proposals/csharp-9.0/init

Использование свойств

```
int MyValue      // объявление свойства
{
   set{ ... }
   get{ ... }
}
...
имя свойства
   ↓
MyValue = 5;     // присваивание: вызывается неявно аксессор set
z = MyValue;     // выражение: вызывается неявно аксессор get
        ↑
   имя свойства
```

Ошибки в применении свойств:

```
y = MyValue.get();     // Ошибка! Нельзя явно вызвать get.
MyValue.set(5);        // Ошибка! Нельзя явно вызвать set.
```

Свойства и связанные поля

```
class C1
{
    private int theRealValue = 10; // Поле: выделение памяти
    public int MyValue // Свойство: нет выделения памяти
    {
        set { theRealValue = value; } // Установка значения поля
        get { return theRealValue; } // Чтение значения поля
    }
}
class Program
{
    static void Main()
    {
        Read from the property as if it were a field.
        C1 c = new C1(); // Получаем значение свойства ↓
        Console.WriteLine("MyValue: {0}",          c.MyValue);
        c.MyValue = 20; // Исп. присваивание для установки свойства
        Console.WriteLine("MyValue: {0}", c.MyValue);
    }
}
```

Именование полей и свойств

```
class C1
{
    private int _firstField;     // Underscore and camel casing
    public int FirstField        // UpperCamelCase (PascalCase)
    {
        get { return _firstField; }
        set { _firstField = value; }
    }

    private int _secondField;    // Underscore and camel casing
    public int SecondField
    {
        get { return _secondField; }
        set { _secondField = value; }
    }
}
```

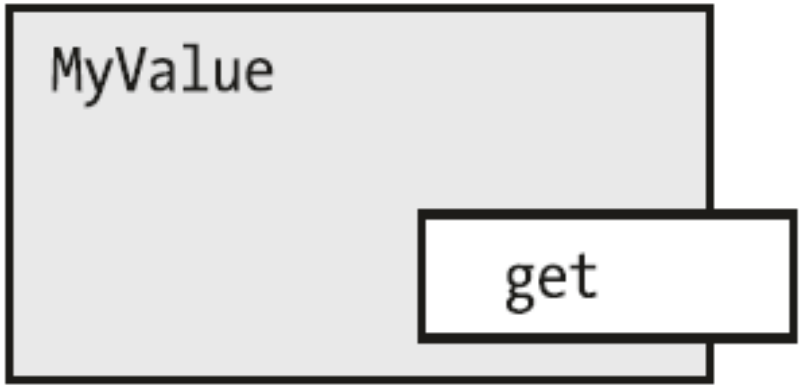
Выполнение вычислений в свойствах

```
public int Useless   {
   set { /* ничего не устанавливается */ }
   get { // возврат 5
       return 5;
   }
}

int _theRealValue = 10;    // поле
int MyValue {        // свойство
    set { // устанавливаем значение поля
        _theRealValue = value > 100     // с проверкой (не более 100).
                  ? 100 : value;
        }
    get { // получаем значение поля
        return _theRealValue;
        }
}
```

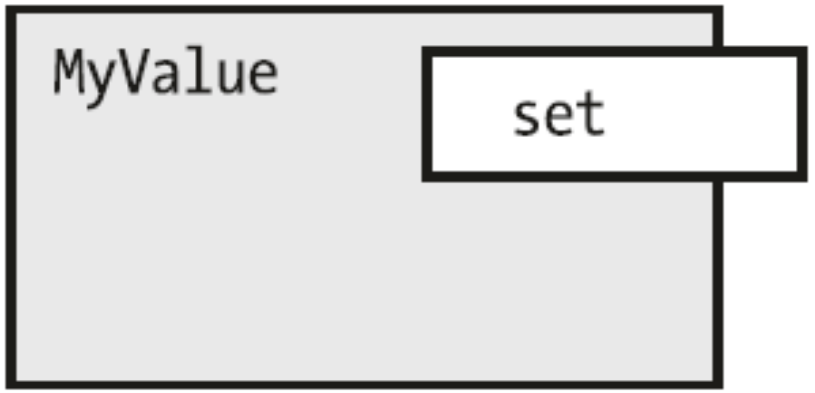
Свойства только для чтения и только для записи

```
int MyValue
{
   get{...}
}
```



Read-Only Property

```
int MyValue
{
   set{...}
}
```

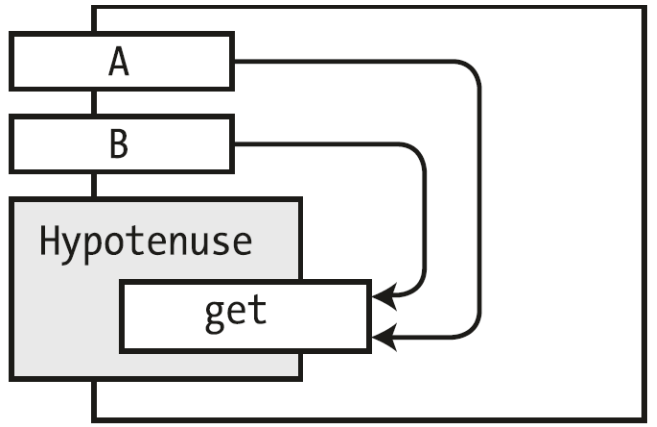


Write-Only Property

Класс «прямоугольный треугольник»

```
class RightTriangle {
   public double A = 3;
   public double B = 4;
   public double Hypotenuse // свойство только для чтения
   {
       get { return Math.Sqrt((A * A) + (B * B)); } // вычисляем
   }
}


class Program {
   static void Main()      {
       RightTriangle c = new RightTriangle();
       Console.WriteLine("Hypotenuse: {0}", c.Hypotenuse);
   }
}
```



Автореализуемые свойства (Automatically Implemented Properties, C# 3.0)

Нет явного определения вспомогательного поля

```
class C1
{
    public int MyValue { get; set; }
}

class Program
{
    static void Main()
    {
        C1 c = new C1();
        System.Console.WriteLine($"MyValue: {c.MyValue}");
        c.MyValue = 20;
        System.Console.WriteLine($"MyValue: {c.MyValue}");
    }
}
```

Тела аксессоров заменяются на “;”

Используем автореализуемые свойства как обычные свойства

Вывод:
MyValue: 0
MyValue: 20

Автореализуемые свойства (2)

```
// Read-only auto-implemented property
using System;
class C1 {
    public int MyValue { private set; get; }
    public C1 (int n) { MyValue = n; }   // Конструктор
    public void SetProperty(int n) { MyValue = n; }
}
class Program {
    static void Main() {
        C1 c = new C1(55);
        Console.WriteLine($"MyValue: {c.MyValue}");
        c.SetProperty(21);
        Console.WriteLine($"MyValue: {c.MyValue}");
    }
}
```

Вывод:
MyValue: 55
MyValue: 21

Статические свойства

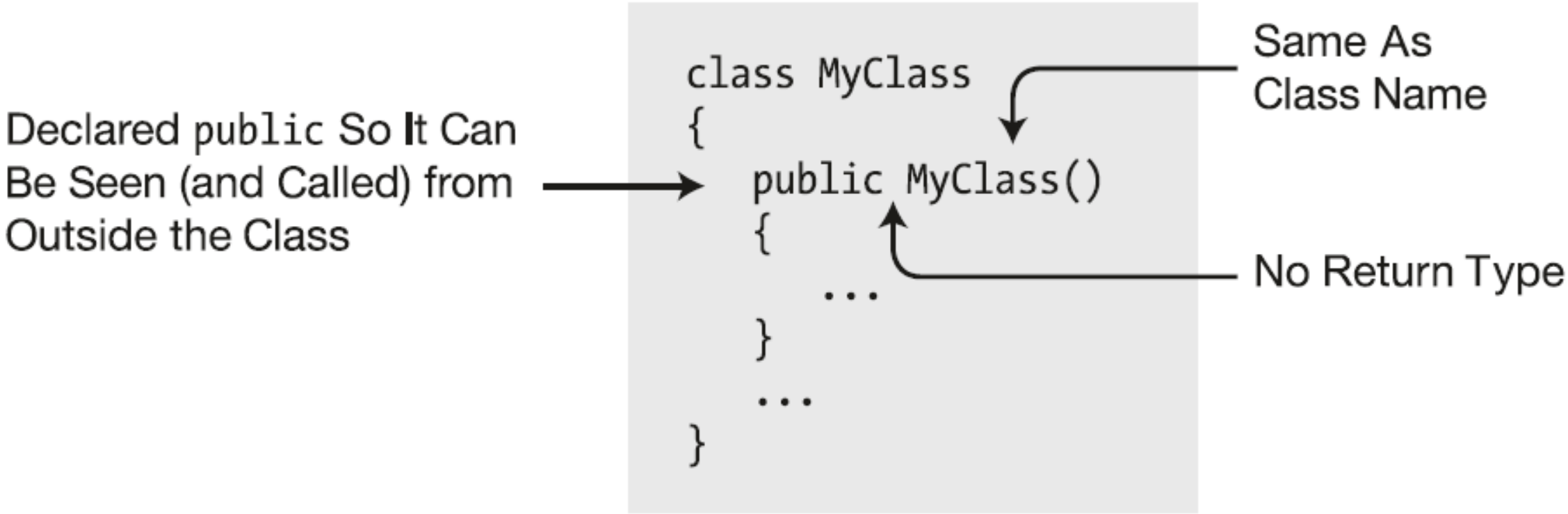
```
// Статическое свойство и использующий его нестатический метод
class Trivial
{
    static int myValue;
    public static int MyValue
    {
        set { myValue = value; }
        get { return myValue; }
    }
    public void PrintValue()
      => System.Console.WriteLine($"Value from inside: {MyValue}");
}
```

Статические свойства (2)

```
// Обращение к статическому свойству извне класса
class Program
{
    static void Main()
    {
        System.Console.WriteLine($"Init Value: {Trivial.MyValue}");
        Trivial.MyValue = 10;
        System.Console.WriteLine($"New Value : {Trivial.MyValue}");
        Trivial tr = new Trivial();
        tr.PrintValue();
    }
}
```

Вывод:
Init Value: 0
New Value : 10
Value from inside: 10

Конструкторы экземпляров класса (объектов)



Конструкторы объектов (2)

```
class MyClass
{
    DateTime timeOfInstantiation; // Поле
    ...
    public MyClass()    // Конструктор
    {
        timeOfInstantiation = DateTime.Now;      // Инициализ.
    }
    ...
}
```

Конструкторы с параметрами. Перегрузка конструкторов

```
using System;
class Class1 {
    int Id;
    string Name;

// Constructor 0
    public Class1() { Id = 28; Name = "Nemo"; }
// Constructor 1
    public Class1(int val) { Id = val; Name = "Nemo"; }
// Constructor 2
    public Class1(string name) { Name = name; }

    public void SoundOff() => Console.WriteLine($"Name {Name}, Id {Id}");
}
```

Применение конструкторов и метода

```
class Program
{
    static void Main()
    {
        Class1 a = new Class1();          // вызов конструктора 0
        Class1 b = new Class1(7);         // вызов конструктора 1
        Class1 c = new Class1("Bill");    // вызов конструктора 2
        a.SoundOff();
        b.SoundOff();
        c.SoundOff();
    }
}
```

Вывод:
Name Nemo, Id 28
Name Nemo, Id 7
Name Bill, Id 0

Конструктор по умолчанию (без параметров)

Программа с типичной ошибкой:

```
class Class2
{
    public Class2(int value) { ... }    // Constructor 0
    public Class2(string value) { ... }     // Constructor 1
}
class Program
{
    static void Main()
    {
        Class2 a = new Class2();     // Ошибка компиляции
        ...
    }
}
```

Формат объявления конструктора

модификаторы_конструктора$_{opt}$

имя_конструктора

(спецификация_параметров$_{opt}$*)*

инициализатор_конструктора$_{opt}$

тело_конструктора

Синтаксические элементы объявления конструктора

модификаторы_конструктора:

public, protected, internal, private, protected internal, private protected, extern

инициализатор_конструктора:

: base (список_аргументов$_{opt}$)

: this (список_аргументов$_{opt}$)

Конструктор с инициализатором

```
class Triangle {
    public double A, B, C; // Стороны треугольника

    public Triangle() { A = B = C = 1; }

    public Triangle(double a, double b, double c)
    {
        if (a + b <= c || a + c <= b || b + c <= a) {
            Console.WriteLine("Ошибка в параметрах!");
            return;
        }
        A = a; B = b; C = c;
    }

    public Triangle(double legA, double legB)
        : this(legA, legB, Math.Sqrt(legA * legA + legB * legB)) { }
}
```

Создание объектов разными конструкторами

```
class Program {
    static void Main() {
        Triangle t1 = new Triangle();
        Console.WriteLine($"t1.A={t1.A}, t1.B={t1.B}, t1.C={t1.C}");

        Triangle t2 = new Triangle(5, 7, 8);
        Console.WriteLine($"t2.A={t2.A}, t2.B={t2.B}, t2.C={t2.C}");

        Triangle t3 = new Triangle(3, 4);
        Console.WriteLine($"t3.A={t3.A}, t3.B={t3.B}, t3.C={t3.C}");

        Triangle t4 = new Triangle(3, 5, 9);
        Console.WriteLine($"t4.A={t4.A}, t4.B={t4.B}, t4.C={t4.C}");

        Triangle t5 = new Triangle(-1, -1, -1);
    }
}
```

Результаты выполнения

Вывод:
t1.A=1, t1.B=1, t1.C=1
t2.A=5, t2.B=7, t2.C=8
t3.A=3, t3.B=4, t3.C=5
Ошибка в параметрах!
t4.A=0, t4.B=0, t4.C=0
Ошибка в параметрах!

Инициализаторы объектов

Инициализатор объекта

```
new TypeName(ArgList) { FieldOrProp = InitExpr, FieldOrProp = InitExpr, ...}
new TypeName { FieldOrProp = InitExpr, FieldOrProp = InitExpr, ...}
```

Инициализаторы свойств или полей

Использование инициализатора объекта

```
public class Point
{
    public int X { get; set; } = 1;
    public int Y { get; set; } = 2;
}
class Program {
    static void Main()
    {
        Point pt1 = new Point();
        Point pt2 = new Point { X = 5, Y = 6 };
        System.Console.WriteLine($"pt1: {pt1.X}, {pt1.Y}");
        System.Console.WriteLine($"pt2: {pt2.X}, {pt2.Y}");
    }
}
```

Инициализатор объекта ↓

Вывод:
pt1: 1, 2
pt2: 5, 6

Неизменяемая точка. Аксессор init (C# 9)

```
class Point {
    public int X { get; }
    public int Y { get; }
    public Point(int x, int y) {
        this.X = x;
        this.Y = y;
    }
}

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

public static void Main() {
    Point point = new Point(x: 42, y: 13);
    PointInit pointInit = new PointInit() { X = 42, Y = 13 };
}
```

В C# 9 появился аксессор **init** (для инициализации):
https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/proposals/csharp-9.0/init

Деструкторы

```
class Class1
{
    ~Class1() // деструктор
    {
        // <Код очистки…>
    }
    // <Остальной код класса…>
}
```

Сравнение конструкторов и деструкторов

		When and How Often Called
Instance	Constructor	Called once on the creation of each new instance of the class.
	Destructor	Called for each instance of the class, at some point after the program flow can no longer access the instance.
Static	Constructor	Called only once—either before the first access of any static member of the class, or before any instances of the class are created—whichever is first.
	Destructor	Does not exist—destructors only work on instances.

Конструкторы и модификатор readonly

```
class Shape {
    readonly double PI = 3.1416;
    readonly int numberOfSides;

    public Shape(double side1, double side2) {
        // четырёхугольник
        numberOfSides = 4;  //  Set in constructor
    }

    public Shape(double side1, double side2, double side3) {
        // треугольник
        numberOfSides = 3;  //  Set in constructor
    }
}
```

Ключевое слово this

```
class ThisDemoClass {
    int number = 10;  // Поле
    public ThisDemoClass(int number) {
        this.number = number;
    }
    public int ReturnMax(int number)
        => number > this.number
                ? number          // параметр метода
                : this.number;    // поле объекта
}
class Program {
    static void Main() {
        ThisDemoClass demo = new(20);
        System.Console.WriteLine($"Max: {demo.ReturnMax(30)}");
        System.Console.WriteLine($"Max: {demo.ReturnMax(5)}");
    }
}
```

Индексаторы (Indexers)

кл. слово список параметров

```
ReturnType this[Type param1,...]
{
   get
   {
        ...
   }
   set
   {
        ...
   }
}
```

квадратные скобки

Список параметров не может быть пустым!

Сравнение индексатора и свойства

Unlike a property, an indexer

- Has a parameter list (in square brackets, no less)
- Uses reference `this`, instead of a name

Like a property, an indexer

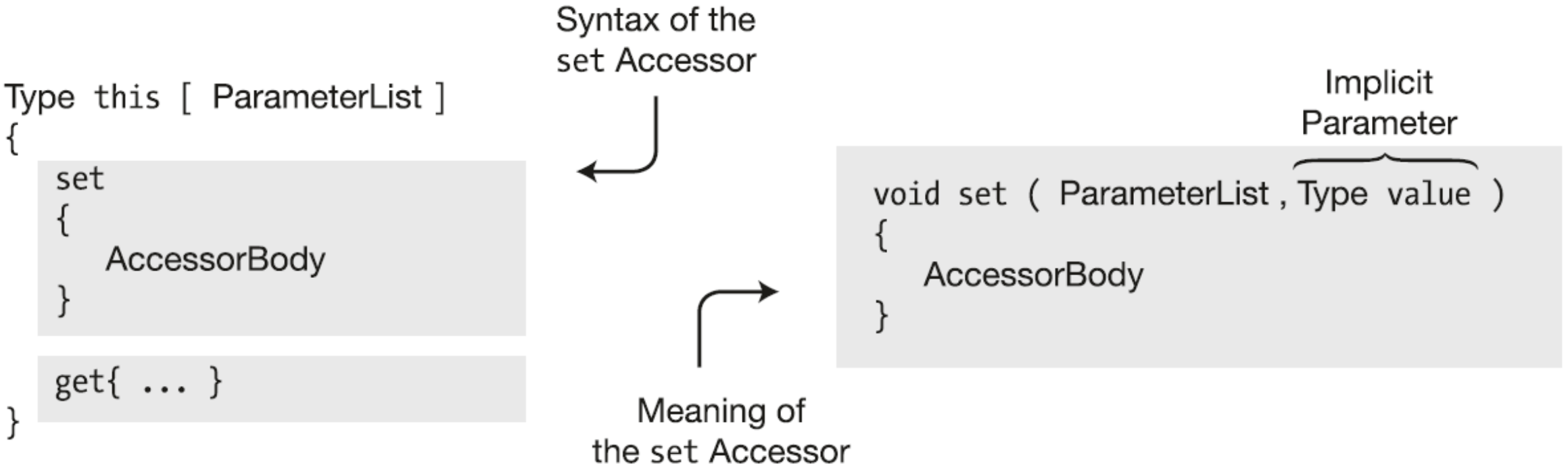
- Has a type
- Has get and set accessors

```
string this [ int index ]
{
    set
    {
        SetAccessorCode
    }
    get
    {
        GetAccessorCode
    }
}
```

Аксессор set

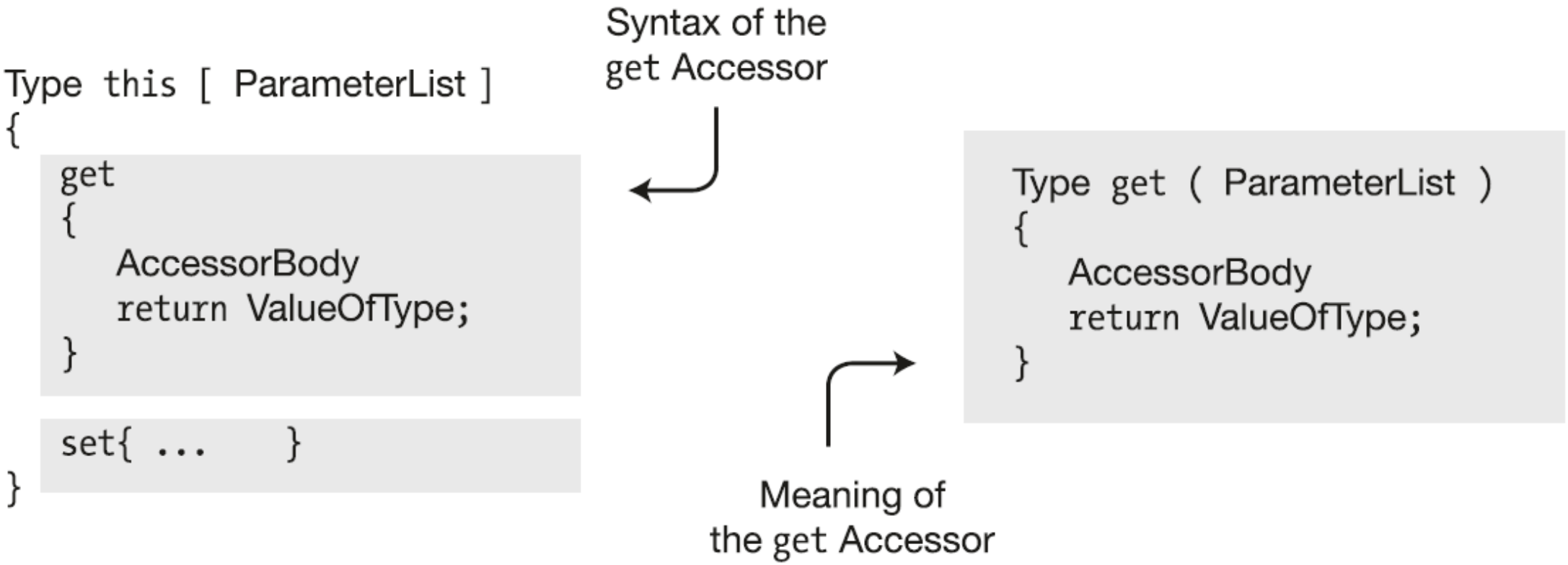
```
emp[0] = "Doe";
```

индекс — значение



Аксессор get

```
string s = emp[0];
           ↑
аргумент индексатора
```



«Вызовы» индексаторов

```
     индекс     значение
       ↓           ↓
emp1[0] = "Doe";                  // вызов аксессора set
string NewName = emp[0];          // вызов аксессора get
                    ↑
                индекс
```

Класс с индексатором

```
class Employee
{
    string LastName;       // обращение к полю по индексу 0.
    string FirstName;      // обращение к полю по индексу 1.
    string CityOfBirth;    // обращение к полю по индексу 2.
    public string this[int index] // объявление индексатора
    {
        set
        {   // аксессор set
            // <Код тела аксессора>
        }
        get
        {   // аксессор get
            // <Код тела аксессора>
        }
    }
}
```

Индексатор в классе Employee

```
public string this[int index]        // определение индексатора
{
    set    { // аксессор set
        switch (index) {
            case 0: LastName = value;          break;
            case 1: FirstName = value;         break;
            case 2: CityOfBirth = value;       break;
            default:// (Exceptions – об исключениях позже…)
                    throw new ArgumentOutOfRangeException("index");
        }
    }
    get    { // аксессор get
        switch (index) {
            case 0: return LastName;           break;
            case 1: return FirstName;          break;
            case 2: return CityOfBirth;        break;
            default:// (Exceptions – об исключениях позже…))
                    throw new ArgumentOutOfRangeException("index");
        }
    }
}
```

Пример индексатора

```
class IndexerClass {
    int _temp0; // закрытое поле
    int _temp1; // закрытое поле
    public int this[int index] // индексатор
    {
        get => index == 0 ? _temp0 : _temp1;
        set
        {
            if (0 == index) {
                _temp0 = value; // неявный параметр "value".
            }
            else {
                _temp1 = value; // неявный параметр "value".
            }
        }
    }
}
```

Перегрузка Индексаторов

```
class MyClass
{
    public string this[int index]
    {
        get { ... }
        set { ... }
    }
    public string this[int index1, int index2]
    {
        get { ... }
        set { ... }
    }
    public int this[float index1]
    {
        get { ... }
        set { ... }
    }
}
```

Модификаторы Доступа в Аксессорах

```
class MyPropertyClass
{
    private string _name = "John Doe";

    public string Name {
        get => _name;
        protected set => _name = value;
    }
}
```