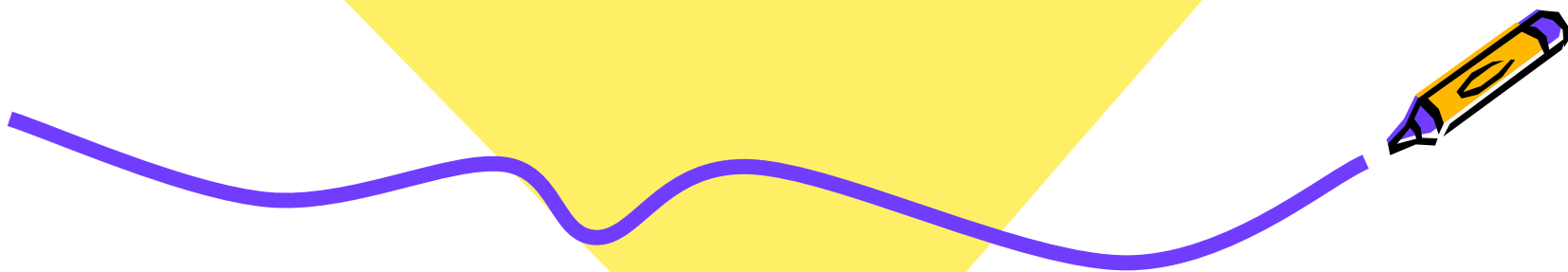




Лекция 3. Класове



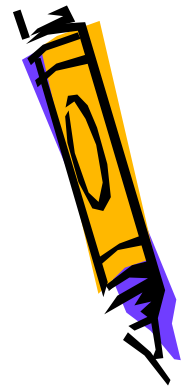
Класове (class) и структури (struct)

Основната разлика между класовете (class) и структурите (struct) в .NET Framework:

- Структурите са стойностни типове, докато класовете са референтни типове.
- Структурите по-интуитивно моделират данни, от които се очаква поведение като на примитивни типове, докато класовете по-добре моделират обекти от реалния свят, които могат да извършват определени действия.
- Структурите не могат да се наследяват, докато за класовете това е основна характеристика.
- Тъй като типовете по стойност в общия случай се създават в стека за изпълнение на програмата, структурите трябва съдържат малки по-обем данни, а по-големите количества е удачно да се обработват с помощта на класове, инстанциите на които съхраняват членовете си в динамичната памет.



```
class TestSomeCoords_struct //Пример за структура
{ struct SomeCoords
    { public int x, y;
      public SomeCoords(int x, int y)
        { this.x = x;
          this.y = y;
        }
    }
  static void Main()
  { // Инициализация:
    SomeCoords first = new SomeCoords(10, 20);
    // Разпечатване на резултати:
    Console.Write("first coordinates: ");
    Console.WriteLine("x = {0}, y = {1}", first.x, first.y);
  }
} //Изход: first coordinates: x = 10, y = 20
//Може успешно да се замени struct с class!!!
```



- Следващият фрагмент демонстрира, една уникална за структурите възможност. Създава се обект `other` от тип `SomeCoords` без да се използва `new`.
- Ако се замени ключовата дума `struct` със `class`, програмата няма да бъде компилирана.

...// Инициализация:

```
SomeCoords other;
```

```
other.x = 10;
```

```
other.y = 30;
```

```
// Разпечатване на резултати:
```

```
Console.Write("other coordinates: ");
```

```
Console.WriteLine("x = {0}, y = {1}", other.x, other.y);
```

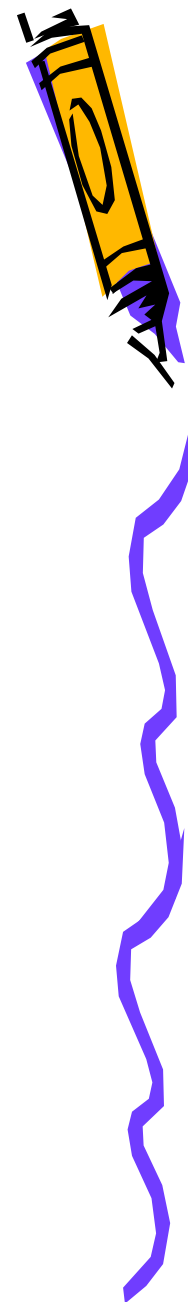
```
...
```

```
}
```

```
}
```

```
// Изход:...
```

```
other coordinates: x = 10, y = 30
```

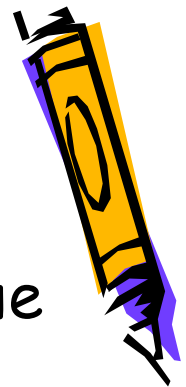


Множествено наследяване в .NET Framework

Интерес представлява въпроса, защо при проектирането на .NET Framework е взето решение да не се допуска множествено наследяване. Аргументите за това решение са:

- множественото наследяване често води до конфликти, например ако един клас наследи елемент с едно и също име от повече от един родител.
- множественото наследяване води до по-сложни и трудно разбираеми йерархии – такива, образуващи граф, докато при наследяването от единствен родител се получава дърво.

В .NET Framework множествено наследяване се реализира чрез имплементиране на няколко интерфейса едновременно, при което обаче не може да се наследят данни или програмен код, а само дефиниции на действия.



Класовете в C#

- Класовете в C# са основните единици, от които се състоят програмите. Те моделират обектите от реалния свят и могат да дефинират различни членове (член-променливи, методи, свойства и др.).

- Общ синтаксис на дефиниране на клас:

[спецификатори] class <име> [:<базови типове>]
{<тяло>}

Пример: `public class Person { /* членове... */ }`

- Създаване на обект от даден клас:

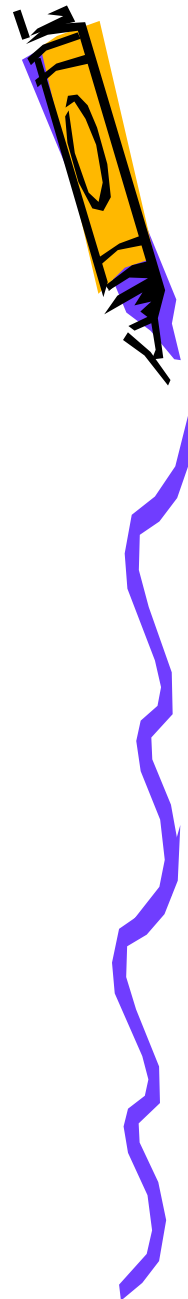
<клас> <реф.>=new <клас> (<args на конструктора (ctor)>)


Пример: `Person s = new Person("111");`



Класовете в C#

- Възможни спецификатори пред ключовата дума `class` могат да бъдат спецификаторите `abstract` и `sealed`, за които ще говорим по-нататък, както и модификаторите за достъп.
- Възможни модификатори за достъп до класа: `internal` (по подразбиране) и `public`.
- Модификатори `private` и `protected` също могат да бъдат задавани, но само за класове, вложени в даден клас.





```
using System;
namespace ConsoleApplication3
{
    class Program
    {
        //подразбира се internal
        class Person //подразбира се internal
        {
            // Private member declarations - fields
            private string mFirstName;
            private string mLastName;
            private string EGN;

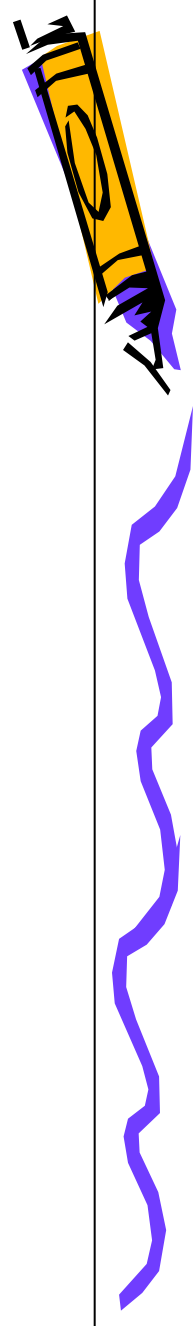
            // Constructor
            public Person(string Id)
            {
                EGN = Id;
            }

            // Properties:
            public string FirstName
            {
                get
                {
                    return mFirstName;
                }
                set
                {
                    mFirstName = value;
                }
            }
        }
    }
}
```

```
public string LastName
{
    get
    {
        return mLastName;
    }
    set
    {
        mLastName = value;
    }
}

// Read-only property:
public string PersonId
{
    get
    {
        return EGN;
    }
}

static void Main()
{
    //use the class
    Person s = new Person("111");
    s.FirstName = "Ivan";
    s.LastName = "Ivanov";
    Console.WriteLine("{0},{1},{2}",
        s.PersonId, s.FirstName,
        s.LastName);
    Console.ReadLine();
}
}
```

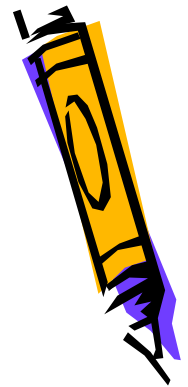


- В примерът е дефиниран класът **Person**, илюстриращ някои от видовете членове:
- капсулираните (**private**) полета **mFirstName**, **mLastName** и **EGN**,
- конструкторът **Person(...)**, чрез който се инициализира поле **EGN**,
- свойствата **FirstName**, **LastName** и **PersonId** – първите 2 за четене и задаване на стойност, а последното само за четене.



Членове на клас в .NET Framework

- полета, или член-променливи (fields)
- константи (constants)
- методи, или член-функции (methods)
- конструктори (constructors) и деструктори
- свойства (properties)
- индексатори (indexers)
- събития (events)
- вложени типове (класове, структури, изброени типове и др.)



Видимост на членовете на тип (class, struct, enum, interface)

- **Ниво public**

Глобална видимост – членовете (или типове) с такова ниво на достъп могат да се достъпват от всеки тип в приложението.

- **Ниво private**

Капсулирани членове (или типове) , видими единствено в рамките на типа, например в класа.

- **Ниво internal**

Членът (или типът) е видим от всички типове, дефинирани в асемблито, в което е дефиниран дадения тип, но недостъпен за други асемблита (за разлика от public).

- **Ниво protected**

Членът е видим в типа и в наследниците на дадения тип и е невидим за останалите типове (в частност – класове).

- **Ниво protected internal**

Това е член, видим от всички типове, в асемблито, в което е дефиниран дадения тип, а също и от наследниците на типа.

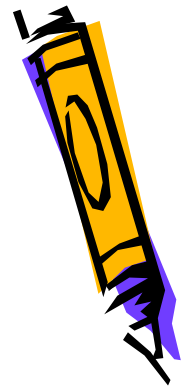


Член-променливи

- Данните, с които инстанцията на класа работи, се съхраняват в **член-променливи** (или още **полета**). Те се дефинират в тялото на класа и могат да се достъпват от други видове членове: методи, конструктори, индексатори, свойства.

Пример:

```
class Student
{
    private string mFirstName;
    //ниво на видимост private
    int mCourse = 1;
    // липсва ниво на видимост
    //подразбира се private
}
```



Дефиниция поле

1. Дефиницията на всяко поле започва с **НИВО НА ВИДИМОСТ**.
2. Следващият елемент от дефиницията на член-променлива е **ТИПЪТ**, който се указва задължително!
3. Задаване на стойност на полетата:

- при декларацията им
- да нямат изрично зададена стойност.
- или да бъдат инициализират в конструктор,

Ако инициализацията бъде пропусната, на член-променливата се задава стойност по подразбиране. За референтните типове това е **null**, а за стойностните типове е **0** или неин еквивалент (**false** за **boolean**). Можем да приемем, че при компилацията инициализациите на полетата се добавят в началото на всеки конструктор, скрито от програмиста.



Обикновени константи (compile-time константи) и read only полета (run-time константи)

class ReadOnlyDemo

```
{ public const double PI = 3.14159; /*
```

- Заместват със стойността си по време на компилация.
- Ключова дума const и
- задължително присвоена стойност;
- по подразбиране са static (принадлежат на типа) и достъпът до тях е чрез името на типа. */

```
private readonly int max=100;
```

```
private readonly int mSize; //стойност, чрез конструктор  
/*
```

- Заместват със стойността си по време на изпълнение;
- Ключова дума readonly;
- Стойността им се задава задължително:
 - или при дефиниране
 - или чрез конструктор (може да са static, тогава и конструктора се дефинира като static);

```
*/
```

```
}
```



using System;

class **Test_Const_and_Read_Only_field**

{ class Student

{ internal const string name = "Ana";

internal readonly int Id;

public Student(int g) **//Конструктор**

{Id = g;}

}

static void Main()

{ **// creating new instance of Student class**

Student st = new Student(123123);

Console.WriteLine

("The Faculty Number of {0} is {1}.", Student.name, st.Id);

}

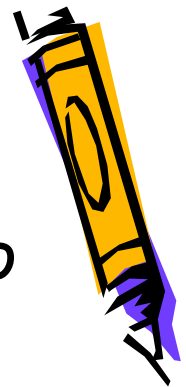
}

ИЗХОД: **The Faculty Number of Ana is 123123**



Конструктори и инициализатори

- Конструкторите се използват при създаване на обекти и служат за **инициализация**, или начално установяване на състоянието на полетата на обекта.
 - Допуска се използването на повече от един конструктор, като конструкторите трябва да се различават по броя и/или типа на параметрите.
 - Възможно е и да не се дефинира конструктор и в такъв случай компилаторът създава подразбиращ се – публичен, с празно тяло и без параметри.
 - Ако не е описан конструктор, то стойностните полета се инициализират с 0, а референтните с null.
 - Името на конструктора съвпада с името на класа.
- Конструкторът не връща стойност.



Конструктори - пример:

```
class Student //подразбира се internal
```

```
{  internal const int group = 3;
```

```
    internal string name;
```

```
    internal int age; ... // fields
```

```
public Student(string name, int age)
```

```
{  this.name = name;
```

```
    this.age = age; }
```

```
public Student(string aName)
```

```
    : this(aName, 20) { }
```

```
...}
```

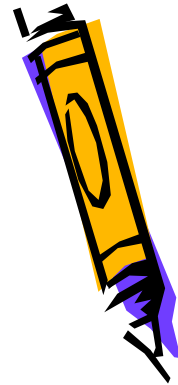
```
class Studentka : Student
```

```
{
```

```
    public Studentka()
```

```
        : base("Maria", 21) { }
```

```
} //end of class Studentka
```



- В показания пример има два конструктора в клас **Student**. Налице е наследяване: клас **Studentka** наследява класа **Student**.
- Вижда се употребата на думата **this** след дефиницията на един от конструкторите на клас **Student**:
инициализаторът **this** позволява на текущия конструктор да извика друг конструктор от собствения си клас, в случая друг конструктор на клас **Student**.
- Вижда се, също че е употребена думата **base** след дефиницията на конструктора на клас **Studentka**
- Инициализаторът **base** указва обръщение на породения клас **Studentka** към конструктор на базовия клас **Student**. Изредените в скобите параметрите, които се подават на извиквания конструктор на базовия клас **Student**, указват кой точно конструктор на базовия клас ще се извика. Вижда се, че това е конструктора: `public Student(string name, int age)`



Деструктори

- В C# съществува специален метод, наречен деструктор, който се извиква автоматично от Garbage Collector (Събирач на боклук), непосредствено преди изчистване на обекта от паметта.
- Ще отбележим, че Garbage Collector отстранява обектите, когато не се използват автоматично, в съответствие със свой собствен алгоритъм, в неизвестен за програмиста момент.

- Синтаксис на деструктора:

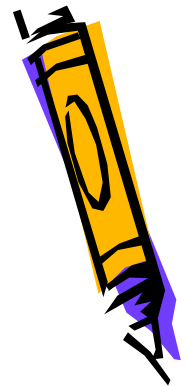
[extern] ~име_на_класа(){тяло на деструктора}

- Деструкторът няма параметри, не връща резултат и няма спецификатор за достъп.
- Името му съвпада с името на класа, но се предхожда със знак (~), символизирайки действие обратно на конструктора.
- Ако деструкторът е определен като външен, използваният спецификатор е extern.
- В общият случай, деструкторите не са често използвани в C# тъй като използването им забавя процеса на Garbage Collector. Създаването на деструктор следва да е само тогава, когато е необходимо да се гарантира коректността на последващото отстраняване на обекта, например да се освободят някакви ресурси преди отстраняването на обекта.



Статични и екземплярни членове

- Членовете могат да бъдат статични (static). Static член - ът принадлежи на самия клас, а не на негов екземпляр.
- Екземплярни (instance) - по default. Всеки екземпляр член принадлежи на конкретен обект (екземпляр).
- По подразбиране - членовете на един тип са екземплярни, с изключение на константните полета



Пример 1:

```
using System;
class MyAge
{   public static int age = 30; //имаме статичен член
    public MyAge()
    {   age += 5;   }
}
class Test
{   static void Main()
    {   Console.WriteLine(MyAge.age); //
        MyAge.age = 20;
        Console.WriteLine(MyAge.age); //
        MyAge s = new MyAge();
        Console.WriteLine(MyAge.age); //
        Console.ReadLine();
    }
}
```

Какъв е резултата от този код?

а) 30, 20, 20

б) 30, 20, 25

в) 35, 20, 25

г) не е посочен коректен отговор



Пример 1 - резултат:

```
using System;
class MyAge
{   public static int age = 30; //имаме статичен член
    public MyAge()             //имаме конструктор, не static
    { age += 5; }
}
class Test
{   static void Main()
    { /*не е допустимо да напишем: MyAge s = new MyAge(); s.age = 20; */
      //Достъпът става чрез името на самия клас:
      Console.WriteLine(MyAge.age); // Отпечатва се 30
      MyAge.age = 20;
      Console.WriteLine(MyAge.age); // Отпечатва се 20
      //Създаваме инстанция s, тоест извикваме конструктора:
      MyAge s = new MyAge();      //тогава:
      //Console.WriteLine(s.age); Това е недопустимо!
      Console.WriteLine(MyAge.age); // Отпечатва се 25
      Console.ReadLine();
    }
}

// 6) 30,20, 25
```



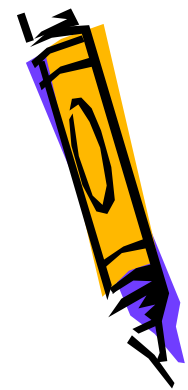
Пример 2:

Малка модификация на примера: конструктора става static, без модификатор за достъп:

```
using System;
class MyAge
{
    public static int age = 30; //имаме статичен член
    static MyAge() //имаме статичен конструктор, може да е само един!
    { age += 5; } //модификатор за достъп не се указва
                  //не може да приема параметри
}
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine(MyAge.age);
        MyAge.age = 20;
        Console.WriteLine(MyAge.age);
        MyAge s = new MyAge();

        Console.WriteLine(MyAge.age);
        Console.ReadLine();
    }
}
```

Какъв е резултата от този код?



Пример 2 - резултат:

Малка модификация на примера: конструктора е static, без модификатор за достъп:

```
using System;
class MyAge
{
    public static int age = 30; //имаме статичен член
    static MyAge() //имаме статичен конструктор, може да е само един!
    { age += 5; } //модификатор за достъп не се указва
                  //не може да приема параметри
}
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine(MyAge.age); /*Отпечатва се 35, тоест задкулисно се
        извиква static конструктор */
        MyAge.age = 20;
        Console.WriteLine(MyAge.age); // Отпечатва се 20
        //Но ако създадем инстанция, тоест ако извикаме конструктора:
        MyAge s = new MyAge();
        //вижда се, че на практика конструктора не се е извикал!!!
        Console.WriteLine(MyAge.age); //тъй като отново се отпечатва 20
        Console.ReadLine(); } }
```



Изводи:

- Статичният конструктор се извиква задкулисно (когато се използва по някакъв начин статичен член).
- Статичен конструктор се използва за инициализация само на статични членове.
- Един клас може да има най-много един статичен конструктор.
- Статичният конструктор не може да приема параметри и модификатори за достъп (`private` е по default).
- Както се вижда статичният конструктор не се извиква, тогава когато се създава нов обект. В практиката е удачно, с цел по-голямо бързодействие, някои сложни изчисления да се направят в тялото на статичен конструктор.



Още изводи:

- Важна особеност, която трябва да имаме предвид при използването на статични методи и свойства, е че те могат да използват само статични полета.
- Статичните полета на типа много приличат на глобалните променливи в по-старите езици за програмиране като C, C++ и Pascal. Както глобалните променливи, статичните полета са достъпни от цялото приложение и имат само една инстанция.
- Конструкторите, индексаторите и събитията също могат да бъдат обявени като статични.
- Константите също са общи за всички инстанции на типа, но не могат да се обявяват изрично като статични. Деструкторите също не могат да бъдат статични



Свойства

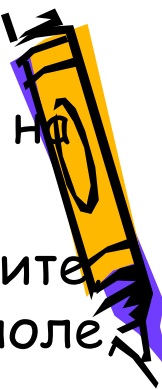
- Видяхме, че клас Person има 3 свойства.

Първите 2 са (read/write) а последното (read only) - за прочитане на стойност на поле.

- Свойствата са членове на класовете, структурите и интерфейсите които се използват за да присвояват и прочитат стойност на поле, като контролират достъпа до полетата на типа и валидират присвояваните стойност (пример на следващия слайд).
- Синтаксис:

[спецификатори] <тип> име_на_свойство{get [{код за четене}]set [{код за запис}]}

- Както се вижда и от примера за клас Person , свойствата могат да имат един или два компонента (accessors):
- get accessor - дефинира се код за присвояване на стойност на полета.
- set accessor - дефинира се код за прочитане на стойност на полета.
- Вижда се, че когато създаваме свойства можем да предоставим дефиниции и на двата компонента, както и на един от тях, но задължително трябва да е дефиниран поне единият.



Пример: Валидиране на присвояваните стойности:

...

```
private int marksOfSoftEng;
```

```
internal int MarksOfSoftEng
```

```
{    set
```

```
{        if (value >= 0 && value <= 100)
```

```
            marksOfSoftEng = value;
```

```
            else
```

```
            {    marksOfSoftEng = 0;
```

```
                // or throw some exception informing
```

```
                // user marks out of range
```

```
                throw new
```

```
                ArgumentException("Invalid value!");
```

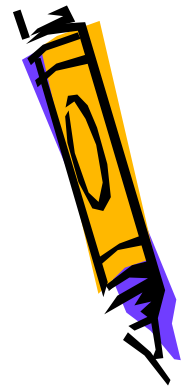
```
            }
```

```
    }
```



Автоматични свойства (автосвойства)

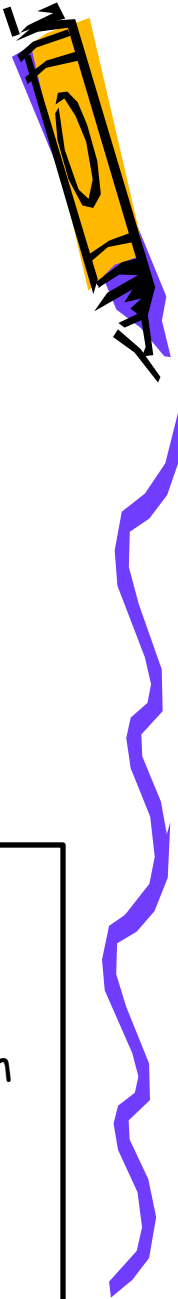
- Свойствата управляват достъпа до полетата на класа.
- Но, ако имаме десетки и стотици полета, то да се пишат еднотипни свойства е досадно.
- Ето защо, от версия .NET 4.0 във Framework-а са добавени тъй наречените „автоматични свойства“.
- Те имат съкратено обявяване:



АВТОМАТИЧНИ СВОЙСТВА (АВТО-СВОЙСТВА)

```
class Person
{   public string Name { get; set; }
    public int Age { get; set; }
    public Person(string name, int age)
    {   Name = name;
        Age = age;
    }
}
```

```
class Program //използване
{   static void Main()
    {   Person person = new Person("Tom",23);
        Console.WriteLine(person.Name); // Tom
        Console.WriteLine(person.Age);  // 23
        Console.ReadLine();
    }
}
```

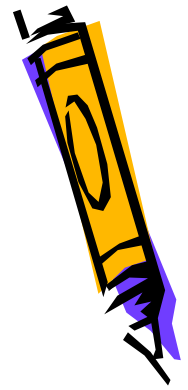


АВТОМАТИЧНИ СВОЙСТВА (АВТО-СВОЙСТВА)

```
class Person
{   public string Name { get; set; }
    public int Age { get; set; }
    public Person(string name, int age)
    {   Name = name;
        Age = age;
    }
}
```

На практика, тук също се създават полета, но те се генерират автоматично от компилатора при компилиране, тоест те не се пишат от програмиста.

От една страна, автоматичните свойства са доста удобни, но от друга, стандартните имат редица преимущества: например, те могат да капсулират допълнителна логика за проверка на стойността.



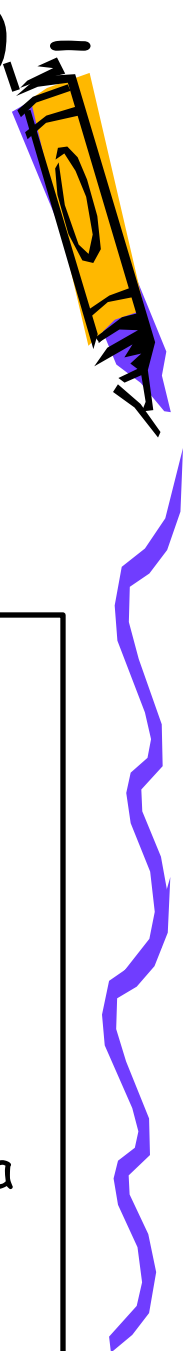
АВТОМАТИЧНИ СВОЙСТВА ПРИ C# 6.0

ИМА ИНИЦИАЛИЗАЦИЯ

```
class Person
{ public string Name { get; set; } = "Tom";
  public int Age { get; set; } = 23;
}
```

```
class Program //използване на класа
{ static void Main(string[] args)
  { Person person = new Person();
    Console.WriteLine(person.Name); // Tom
    Console.WriteLine(person.Age); // 23
  }
}
```

Така, ако не зададем стойност на свойствата Name и Age, за обект от тип Person ще са валидни стойностите по подразбиране.



Автоматични свойства при C# 6.0, спрямо C# 5.0

Докато в C# 5.0, за да направим едно свойство достъпно само за класа трябваше да укажем модификатор `private` за `set`, т.е. трябваше да напишем `private set`:

```
class Person // C# 5.0
{
    public string Name { get; private set; }
    public Person(string n)
    {
        Name = n;
    }
}
```

В C# 6.0 изобщо не е нужно да се пише `set`:

```
class Person // C# 6.0
{
    public string Name { get;}
    public Person(string n)
    {
        Name = n;
    }
}
```

Индексатори

- **Индексаторите в C#** (indexers) са членове на класовете, структурите и интерфейсите, които представляват разновидност на свойство и обикновено **се използват за организация на достъп до скрити полета на типа (в частност клас) по индекс**, така както се обръщаме към масив.
- Синтаксисът на индексаторите е много подобен на синтаксиса на свойствата (дори в някои .NET езици, например VB.NET, синтаксисът на декларирането им е същият като при свойствата), но имат параметър (параметри), най-често един индекс, по които се осъществява достъп до елементи:

```
[спецификатори] <тип> this [списък_от_параметри]  
// последните [ ] се явяват елементи на синтаксиса  
{  
get [{код за четене}]  
set [{код за запис}]  
}
```



Индексатори - пример

using System; //намира, изчислява и извежда числото на Фибоначи:

class DemoFib

```
{ public int this[int i] //индексатор, само за четене
    { get { int a = 1, b = 1, c;
            for (int j = 3; j <= i; ++j)
                {c = a + b; a = b; b = c; }
            return b;
        }
    }
```

```
}
```

class Test_Indexer

```
{ static void Main()
```

```
{Console.Write("Въведете цяло положително число n="); //n=45
```

```
int n = int.Parse(Console.ReadLine());
```

```
DemoFib a = new DemoFib();
```

```
Console.WriteLine("a[{0}]={1}", n, a[n]);
```

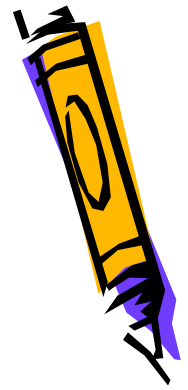
//a[45]=1134903170

```
Console.ReadKey();
```

```
}
```



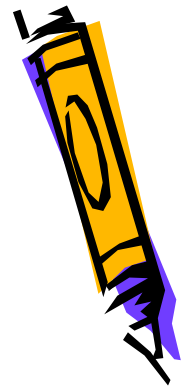
- Виждаме, че дефиницията на индексатор прилича на тази на свойство.
- Индексаторите често се обявяват с модификатор за достъп `public`, тъй като се използват като интерфейс на обекта.
- Както и при свойствата, задължително е налице поне един от аксесорите `get` или `set`. Но има и някои разлики:
 - На индексатора не се задава име, а вместо него се задава запазената дума **this**.
 - Достъпът до индексатор на обект се извършва посредством името на променливата (в случая `a`) от тип, дефиниращ индексатора (в случая `DemoFib`), последвана от индекс в квадратни скоби, подобно на достъп до елемент на масив (например, `a[n]`).



- Индексаторите са много удобни за създаване на специализирани масиви и колекции, чиято работа е свързана с някакви ограничения.

Следват два примера:

- Вдин пример, в който чрез един клас DemoArray се дефинира специализиран масив, чиито допустими елементи да са в диапазона $[0, 100]$.
- Ето още елементарен пример за индексатор, който приема 3 параметъра от цял тип и връща тип символен низ:



using System;

class DemoArray

```
{ private int[] MyArray; //масив, до който няма директен достъп
  public DemoArray(int size) //конструктор
  { MyArray = new int[size]; }
  public int this[int i] //индексатор
  { get {return MyArray[i]; }
    set {if (value >= 0 && value <= 100) MyArray[i] = value;} }
}
```

class Test_Indexers

```
{ static void Main()
{ int numberOfElements = 10;
  DemoArray a = new DemoArray(numberOfElements);
  for (int i = 0; i < numberOfElements; i++) {
    a[i] = i * i; // използване на индексатора
    Console.Write(a[i] + " ");
  }
}
```



using System;

class Grade

{

public string this[int Grade1, int Grade2, int Grade3]

{

get

{if ((Grade1 + Grade2 + Grade3)/3 >89)return "ти си отличник!";

else return "ти не си отличник!";}

}

}

class GradeTest

{ static void Main()

{

Grade d = new Grade();

Console.WriteLine("Ако успехът ти е {0}, {1} и {2} точки, то {3}." ,50,
90, 56, d[50, 90, 56]);

} **Резултат:**

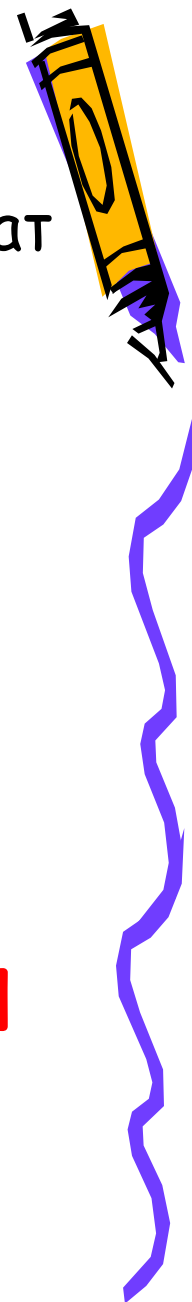
**Ако успехът ти е 50, 90 и 56 точки,
то ти не си отличник!**



Особености на наследяването

- Класовете с в C#, както вече бе показано чрез някои примери (за клас Student, например) могат да имат произволно количество потомци (наследници), но само един предшественик (родител).
- При описанието на класа, името на неговия предшественик се описва в заглавния ред след знак (:). Ако предшественик не е описан, то за базов се счита System.Object.
- Ще припомним синтаксиса на наследяване:

**[спецификатори] class <име> [:<базови типове>]
{<тяло>}**



Особености на наследяването

- Независимо, че е в общия вид на описанието на клас е записано „базови типове“ (а не „базов тип“), то това е защото класът може да наследява наред с един единствен клас-предшественик и много на брой интерфейси.
- Прието е, класът, който се наследява да се нарича базов клас, а класът, който наследява базовия клас да се нарича производен. Производният клас наследява всички членове на базовия клас, но може да добавя и свои уникални членове както и да предефинира наследени.



- В следващия демонстрационен пример е дефиниран базов клас DemoPoint (точка в равнината), а производния е DemoLine (линия в равнината).
- Резултатът от изпълнение показва, че когато в базов клас е използван модификатор virtual а в производен клас - ключовата дума override, се реализира всъщност предефиниране на виртуалния член (в случая метод Display()). Така на практика се реализира полиморфизъм в C#. Ще отбележим също, че когато в производен клас се предефинира виртуален член на базовия, то този член е виртуален и в производния клас.
- Резултатът от изпълнение показва също, че когато в породения клас се предефинира един член на базовия клас (в случая Show()), използвайки ключовата дума new, то той се скрива от базовия клас.



```

using System;
class DemoPoint //базов клас
{
    public int x;
    public int y;
    public virtual void Display()
    {Console.WriteLine("the base class!!!");
    }
    public void Show()
    {Console.WriteLine("(x={0}, y={1})", x, y);}
}

class DemoLine : DemoPoint //производен клас
{
    public int xEnd;
    public int yEnd;
    public override void Display()
    {Console.WriteLine("the overridden method!!!");
    }
    public new void Show()
    {Console.WriteLine("(x={0}, y={1})-(xEnd={2}, yEnd={3})", x, y, xEnd, yEnd);
    }
}

```

```

class Test
{
    static void Main()
    {
        DemoPoint point = new DemoPoint();
        point.x = 0;
        point.y = 0;
        point.Display();
        point.Show();
        DemoLine line = new DemoLine();
        line.x = 1; line.y = 1;
        line.xEnd = 10; line.yEnd = 10;
        line.Display();
        line.Show(); //Нека point=line;
        point = line;
        point.Display();
        point.Show();
    }
}

```

```

/* Иъход:
the base class!!!
(x=0, y=0)
the overridden method!!!
(x=1, y=1)-(xEnd=10, yEnd=10)
the overridden method!!!
(x=1, y=1)
*/

```



Sealed клас

- В C# е налице ключова дума **sealed**, позволяваща описание на „защитен“ клас, такъв клас, който не може да бъде наследяван.
- Така, например ако дефинираме клас Student по следния начин:

```
sealed class DemoPoint //базов клас  
{...}
```

- То наследяването му ще е невъзможно, тоест:

```
class DemoLine : DemoPoint //производен клас  
//Error 'DemoLine': cannot derive from  
sealed type 'DemoPoint'
```

