Лекции ООП (C#)

Съдържание

Лекция №2 Базови елементи на езика.

- **Пространства на имената (*Namespaces*)**

Чрез тях .NET осигурява контейнери за код на приложенията, така че съдържащите се в този код елементи да могат да бъдат еднозначно идентифицирани. Пространствата на имената се използват още и за категоризиране на елементите на .NET Framework, повечето от които са дефиниции на типове (например `System.Int32`, `System.String`).

По подразбиране кодът се разполага в глобалното пространство на имената (*global namespace*) и е достъпен само по име (без квалифициране). С помощта на ключовата дума `namespace` можем да дефинираме *namespace* за блок от код, заграден във фигурални скоби. Имената от това пространство трябва да бъдат квалифицирани, ако се използват извън него. Квалифицираните имена използват точка (`.`) за разделител между различните нива на влагане на пространствата

```
System.Collections.Generic.List<string> list = new List<string>();
```

С помощта на оператора `using` може да се ползват имената от дадено пространство, ако неговият код е свързан по някакъв начин към проекта (или е дефиниран в сорс код към проекта, или е включен в References на проекта)

```
using System.Collections.Generic;
List<string> list;
```

Visual Studio създава *namespace* за всеки нов проект.

Новост в C# 6 е операторът `using static`. С него се осигурява директен достъп до статични членове, например статичният метод `public static void WriteLine`, който е част от статичния клас `System.Console` може да бъде извикван директно с краткото си име:

```
using static System.Console;
WriteLine("It's OK");
```

- **Преобразуване на типовете**

Всички данни, независимо от своя тип, се записват в поредици от битове (нули и единици). Типът казва как да се интерпретират тези битове. Например типът `char` (`System.Char`) използва 16 бита, за да представи *unicode* символ чрез число между 0 и 65535. Това число се записва по същия начин, както и неотрицателно цяло число от типа `ushort` (`System.UInt16`). По принцип обаче различните типове използват различни схеми за представяне на данните, което означава, че дори да е възможно да разположим поредица от битове в променлива от друг тип, не винаги ще получим това, което очакваме. Затова вместо директно съпоставяне на редици от битове трябва да използваме преобразуване на типовете. Има два вида преобразуване – косвено (*implicit*) и явно (*explicit*).

При косвеното преобразуване от тип A към тип B е възможно във всички случаи и правилата за такова преобразуване са достатъчно прости, за да се доверим на компилатора:

```
char ch ='a';
ushort us = ch;          //implicit conversion
Console.WriteLine(us);   //a
us = 1048;
ch = (char) us;          //explicit conversion
Console.WriteLine(ch);   //и
```

Виждаме, че косвено преобразуване от `ushort` към `char` не е възможно.

Implicit Numeric Conversions Table: https://msdn.microsoft.com/en-us/library/y5b434w4.aspx

From	To
sbyte	short, int, long, float, double, or decimal
byte	short, ushort, int, uint, long, ulong, float, double, or decimal
short	int, long, float, double, or decimal
ushort	int, uint, long, ulong, float, double, or decimal
int	long, float, double, or decimal
uint	long, ulong, float, double, or decimal
long	float, double, or decimal
char	ushort, int, uint, long, ulong, float, double, or decimal
float	double
ulong	float, double, or decimal

Правилото за косвено преобразуване е следното: Всеки тип A, чийто диапазон на допустими стойности попада изцяло в диапазона допустими стойности на тип B, може да бъде косвено преобразуван до тип B.

Синтаксисът за явно преобразуване е следният:

```
(<destinationType>) <sourceVar>
```

Преобразува стойността на `<sourceVar>` към тип `<destinationType>`.

При такова преобразуване е възможна загуба на данни:

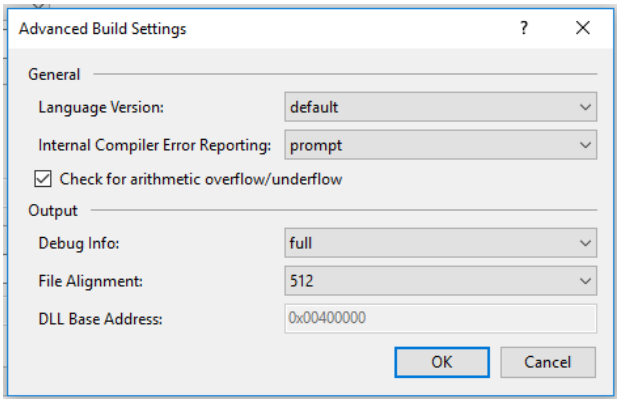
```
byte dest;          //8-bit unsigned integer
short source = 281; //16-bit signed integer
dest = (byte) source;
Console.WriteLine($"source: {source}");     //source: 281
Console.WriteLine($"destination: {dest}");  //destination: 25
```

Програмистът трябва да се погрижи да провери дали стойността на `<sourceVar>` може да се събере в `<destinationType>` или да накара системата да провери за препълване (*overflow*) по време на изпълнение на програмата. Това става с ключовата дума `checked`.

```
dest = checked ((byte) source);
```

Изпълнението на този код ще предизвика грешка „An unhandled exception of type 'System.OverflowException' occurred…„. Ако използваме вместо това ключовата дума `unchecked`, програмата ще се държи както в по-горния пример (без проверка).

Могат да се променят настройките на проекта за проверка за препълване (*overflow checking*), така че по подразбиране всеки такъв израз да е `checked`, освен ако изрично не се укаже `unchecked`. С десния бутон върху името на проекта в Solution Explorer, Properties/Build/, бутона Advanced… и избираме чекбокса Check for arithmetic overflow/underflow:



Явно преобразуване може да се прави и с командите за преобразуване от статичния клас `System.Convert`. Сигурно вече сте ги ползвали при въвеждане на числови данни от клавиатурата:

```
string snum = Console.ReadLine();
double num = System.Convert.ToDouble(snum);
Console.WriteLine(num);
```

Ясно е, че такива преобразувания няма да работят с всякакви въведени стойности. В случая въведеният низ трябва да отговаря на изискванията за записване на числа – може да има знак и да е в експоненциална форма.

Характерна особеност е, че такива преобразувания винаги са `checked` и ключовите думи, както и настройките на проекта не влияят на това.

- **Функции**

Сигнатура на функция – името и параметрите, без типа на резултата.

Досега сте работили само със статични функции.

Имената на функциите обикновено се пишат по конвенцията *PascalCase*.

Кратки функции, които изпълняват например един ред код могат да се пишат като *expression-bodied methods* (възможност, налична в C# 6). Използва се => (*lambda arrow*).

```
public static int Product(int a, int b) => a * b;
```

вместо

```
public static int Product(int a, int b)
{
    return a * b;
}
```

Общият вид на дефиниция на функция е следният:

```
[<static>] <тип_на_резултата> <Име>(<тип> <параметър>, ...)
{
```

```
        ...
        return <стойност>;
}
```

Параметри:

В спецификацията на езика C# се разграничават понятията „параметър“ и „аргумент“. Параметрите са част от дефиницията на функцията, а аргументите – част от обръщението към нея (извикването ѝ). Някъде се наричат съответно формални и фактически параметри. Те трябва да си съответстват (по типове, брой и подредба).

Всеки параметър е достъпен в тялото на функцията като променлива.

Пример: Намиране на най-голямата стойност в масив от цели числа.

```
static int MaxValue(int[] intArray)
{
    int max = intArray[0];
    for (int i = 1; i < intArray.Length; i++)
    {
        if (intArray[i] > max)
            max = intArray[i];
    }
    return max;
}

static void Main(string[] args)
{

    int[] myArray = { 1,8,3,6,2,5,9,3,0,2 };
    int maxVal = MaxValue(myArray);
    Console.WriteLine($"The maximum value in myArray is {maxVal}");
    // Може и така:
    Console.WriteLine($"{MaxValue(new int[] { 1,8,3,6,2,5,9,3,0,2 })}");
}
```

Функции с променлив брой аргументи

C# позволява функцията да съдържа точно един специален параметър, който трябва да бъде последен в списъка с параметри. Той се нарича *parameter array* и се разглежда като масив от параметри. Този параметър се отбелязва с ключовата дума `params`. С негова помощ една и съща функция може да бъде извиква с различен брой аргументи.

Ако променим сигнатурата на функцията `MaxValue`, като добавим пред параметъра ѝ `params`,

```
static int MaxValue(params int[] intArray)
```

ще можем да я извикваме така:

```
int maxVal = MaxValue(1, 8, 3, 6, 2, 5, 9, 3, 0, 2);
```

Тази функция може да бъде извиквана и с аргумент масив, както в предишния ѝ вид:

```
int maxVal = MaxValue(myArray);
```

Още един пример – функция, която намира сумата от аргументите си – цели числа:

```
static int SumInts(params int[] intArray)
{
```

```
            int sum = 0;
            foreach (int x in intArray)
                sum += x;
            return sum;
        }
```

Тя се извиква така:

```
Console.WriteLine(SumInts(1, 2, 3, 4, 5));                  //15
Console.WriteLine(SumInts(1, 8, 3, 6, 2, 5, 9, 3, 0, 2));   //39
Console.WriteLine(SumInts(myArray));                        //39
```

Предаване на параметрите по референция и по стойност

В примерите дотук параметрите на функциите са параметри стойности (*value parameters*). Това означава, че стойността на аргумента (с който се извиква функцията) се копира в променливата, която използваме за параметър и функцията работи с това копие, а не с оригиналния параметър. Всякакви промени на параметъра не се отразяват на аргумента, например:

```
static void Swap(int x, int y)
{
    int t = x;
    x = y;
    y = t;
}

static void Main()
{
    int a = 5, b = 10;
    Swap(x, y);
    Console.WriteLine($"a = {a}, b = {b}");       // interpolated string expression
    Console.WriteLine("a = {0}, b = {1}", a, b);  // вместо да се използва това
}
```

За да се извърши размяната, параметрите трябва да се предадат по референция, тоест функцията да работи с обектите, с които е била извикана, а не с техни копия. Това става със спецификатора `ref` за съответния параметър и аргумент:

```
static void Swap(ref int x, ref int y)
```

Извикването в `Main()` също изисква спецификатора:

```
Swap(ref x, ref y);
```

Предаването на параметър по референция има две ограничения:

- аргументът не може да бъде константа;
- аргументът трябва да бъде **инициализирана** променлива.

out параметри

Това е още една възможност за управление на начина, по който се предават параметрите. Използва се модификатора `out`. Той е подобен на `ref`, защото отново в края на изпълнението на функцията стойността на параметъра се записва в променливата-аргумент, но има две съществени разлики:

- може да се използва неинициализирана променлива като *out* аргумент;
- функцията третира *out* параметъра като неинициализиран. Тоест дори аргумента да има някаква стойност в момента на извикването, тази стойност се губи, когато започне изпълнението на функцията.

Пример: Ще променим функцията MaxValue, така че да намира и индекса на елемента с най-голяма стойност. Ще променим малко и алгоритъма, като използваме полето MinValue на типа int.

```
static int MaxValue(int[] intArray, out int maxIndex)
{
     int max = int.MinValue; //най-малката стойност, допустима за типа
     maxIndex = 0;
     for (int i = 0; i < intArray.Length; i++)
     {
         if (intArray[i] > max)
         {
             max = intArray[i];
             maxIndex = i;
         }
     }
     return max;
 }
```

И в Main:

```
int[] myArray = { 1, 8, 3, 6, 2, 5, 9, 3, 0, 2 };
int maxIndex;
Console.WriteLine($"The maximum is {MaxValue(myArray, out maxIndex)}");
Console.WriteLine($"The first occurrence of this value is at pos {maxIndex + 1}");
```

Отново при извикването се добавя спецификатора out. В C# 7 *out* параметрите могат да се декларират директно (inline) в обръщението към функцията. Областта им на действие е съдържащия ги блок:

```
int[] myArray = { 1, 8, 3, 6, 2, 5, 9, 3, 0, 2 };
Console.WriteLine($"The maximum is {MaxValue(myArray, out int maxIndex)}");
Console.WriteLine($"The first occurrence of this value is at pos {maxIndex + 1}");
```

Предефиниране на функции

Това е възможността да имаме няколко функции с едно и също име, но с различни параметри (по брой и тип). Коя функция да бъде изпълнена решава компилатора в зависимост от аргументите, с които е извикана.

```
static int Sum(int a, int b)
{
    return a + b;
}
static double Sum(double a, double b)
{
    return a + b;
}
static int Sum(params int[] args)
{
    int s = 0;
    foreach (int x in args)
        s += x;
    return s;
}
static void Main(string[] args)
{
    Console.WriteLine(Sum(1,2));            // 3
    Console.WriteLine(Sum(1,2,3,4,5));      // 15
    Console.WriteLine(Sum(3.5, 4.5));       // 8
}
```

Възможно е параметрите на функциите да се различават само по това дали параметрите им се предават по стойност или по референция:

```
static void ShowDouble(ref int val)
{
     //...
}
static void ShowDouble(int val)
{
     //...
}
```

Коя от двете версии ще се изпълни зависи дали в обръщението се съдържа `ref`.

Параметри с подразбиращи се стойности (*Optional Parameters*)

Позволява асоцииране на параметъра с **константна** стойност в декларацията на метода. Това позволява извикване на метода с пропускане на подразбиращите се параметри.

```
public static int power(int a, int n=2)
{
    int res = a;
    for (int i = 1; i < n; i++)
    {
        res *= a;
    }
    return res;
}
static void Main(string[] args)
{
    Console.WriteLine(power(2,3));  // 8
    Console.WriteLine(power(5));    // 25
}
```

Параметрите с подразбиращи се стойности трябва да са разположени след изискваните параметри (тези, които нямат подразбиращи се стойности).

Именувани аргументи

При извикването на функцията може изрично да се посочи името на параметъра, на който да се присвои стойността, а не да се разчита на съответствие в подредбата на аргументите и параметрите. Това е удобно при функции с много параметри с подразбиращи се стойности – така можем да прескочим няколко от тях. Недостатък е, че при промяна на името на параметъра трябва да се променя и клиентския код. Затова се препоръчва да не се преименуват параметри.

```
static void Main(string[] args)
{
    DisplayGreeting(lastName: "Donchev", firstName:"Ivaylo");

}
public static void DisplayGreeting(
    string firstName,
    string middleName = default(string),
    string lastName = default(string))
{
    Console.WriteLine($"Hello, {firstName} {middleName} {lastName}!");
}
```

- **Делегати**

Делегатът (*delegate*) е тип, чиито стойности са референции към функции. Типичното приложение на делегатите е при работата със събития (*event handling*), която ще разгледаме по-късно в курса. Декларирането на делегат е подобно на това на функция, но без тяло. Използва се ключовата дума `delegate`. Декларацията определя тип на резултата и списък с параметри на делегата.

След декларирането на делегата могат да се декларират променливи от тип този делегат и те да се инициализират с референции на функции, които имат същия тип на резултата и списък с параметри. След това променливата-делегат може да се използва като функция. Това позволява чрез делегати функции да се предават като параметри на други функции.

Пример: Извикване на библиотечни и потребителски функции чрез делегат:

```
class Program
{
   delegate double Calculation(double a);  // декларация на делегат
   static double Square(double x) => x * x;
   static void Proccess(Calculation calc, double arg)  // функция с аргумент делегат
   {
       Console.WriteLine(calc(arg));   // извикване на ф-я чрез променлива-делегат
   }
   static void Main()
   {
       Console.WriteLine("Enter the angle in degrees: ");
       double x = Convert.ToDouble(Console.ReadLine()) * Math.PI / 180;
       Proccess(Math.Sin, x);
       Proccess(Math.Cos, x);
       Console.WriteLine("Enter the number to square: ");
       x = Convert.ToDouble(Console.ReadLine());
       Proccess(Square, x);    // потребителската функция като параметър

       Calculation c;  // дефиниране на променлива-делегат
       c = n => 2 * n; // ламбда функция, която пасва на делегата
       Console.WriteLine(c(x));   // 2*x

       Console.ReadKey();
   }
}
```

Присвояването на референция на функция на променливата-делегат може да стане и така:

```
c = new Calculation(n=>2*n);
c = new Calculation(Math.Sin);
```

Въпрос на личен избор е кой от двата синтаксиса да се използва.

- **Изброявания (*enums*)**

Има ситуации, в които бихме искали да ограничим множеството от стойности, които може да заема дадена променлива, както и да зададем имена на тези стойности. В такива случаи е удачно да се използва изброяване.

Пример: Посоките на света.

```
enum Orientation { North, South, East, West }
```

Изброяването дефинира тип от краен брой стойности, които ние задаваме. След това могат да се дефинират променливи от този тип и да им се присвояват стойности от изброяването.

```
Orientation orientation = Orientation.North;
```

Изброяванията имат базисен тип, използван за съхраняване на стойностите. По подразбиране той е `int`. Може да се избере друг базисен тип при декларирането на изброяването:

```
enum <typeName> : <underlyingType>
{
    <value1>,
    <value2>,
    <value3>,
    ...
    <valueN>
}
```

```
enum Orientation : byte { North, South, East, West }
```

Базисни типове могат да бъдат `byte`, `sbyte`, `short`, `ushort`, `int`, `uint`, `long`, и `ulong`. По подразбиране на всяка стойност на изброяването се присвоява кореспондираща стойност на базисния тип, започвайки от нула. С оператора = могат да се задават други стойности. Стойностите може да се повтарят. Ако при тези присвоявания се получи цикъл, това предизвиква грешка:

```
enum <typeName> : <underlyingType>
{
     <value1> = <value2>,
     <value2> = <value1>
}
```

Възможно е преобразуване на типовете от изброяване към базисен и обратно:

```
Orientation orientation = (Orientation) 2;
Console.WriteLine(orientation);     // East
orientation = Orientation.South;
Console.WriteLine((byte)orientation); // 1
```

Възможно е и преобразуване от `string` до изброяване. Използва се командата `Enum.Parse`. Синтаксисът е малко по-сложен:

```
Orientation myDirection = (Orientation) Enum.Parse(typeof(Orientation),"West");
```

Трябва да се внимава, защото ако се използва низ, който не съвпада със стойност на изброяването, ще възникне грешка. Тези стойности са case sensitive.

Лекция №3 Създаване и управление на класове и обекти.

Класовете ни дават удобен механизъм за моделиране на субекти (*entities*) в програмите. Entity може да бъде абстракция на реален обект, например клиент (*customer*) или нещо по-абстрактно като транзакция (*transaction*). Дизайнът на всяка система включва откриване на тези *entities*, които са важни за процесите, които имплементира системата и след това анализирането им, за да се определи каква информация трябва да съдържат и какви операции да извършват. Информацията се съхранява е полета (*fields*), а операциите се реализират чрез член-функции (методи).

➢ Капсулиране

Капсулирането е важен принцип при дефинирането на класове. Идеята е клиентите на класа да не се интересуват от това как той работи, за да предоставя очакваното поведение. За реализацията на методите си класът може да се нуждае от допълнителна информация, представяща вътрешното му състояние и допълнителна функционалност (методи), които остават

скрити за клиентите. Затова капсулирането понякога се нарича и „скриване на информацията“ (*information hiding*).

- **Дефиниране и използване на класове**

Класове се дефинират с ключовата дума `class`. Членовете на класа – полета (*fields*), свойства (*properties*) и методи (*methods*) се разполагат в неговото тяло – блок, ограден от фигурни скоби:

```
class Circle
{
    int radius;
    double Area()
    {
        return Math.PI * radius * radius;
    }
}
```

Този клас съдържа едно поле с данни и един метод.

С дефинирането на клас се създава нов тип данни, който се използва по аналогичен на останалите типове начин:

```
Circle c;           //дефинира променлива от тип Circle
c = new Circle();   //инициализира променливата
```

С ключовата дума `new` се създава нова инстанция (обект) на класа.

```
Circle d;        //друга променлива от същия тип
d = c;           //възможно е присвояване между обекти на един и същ клас
```

Важно е да се разграничават двете понятия – „клас“ и „обект“.

- **Контрол на достъпа**

Така дефиниран, класът `Circle` не е особено полезен. По подразбиране членовете на класа са скрити за външния свят. Полетата (като `radius`) и методите (като `Area()`) са видими за методите на класа, но не и за други функции извън класа – те са частни (`private`) за класа. И макар че можем да създадем обект на `Circle` в програмата, нямаме достъп до неговото поле `radius` и неговия метод `Area`. За да управляваме достъпа до елементите на класа, използваме ключовите думи `public`, `private`, `protected` и `internal`, които се наричат модификатори на достъпа (*access modifiers*). Тези модификатори могат да се прилагат както към членове (*members*) на класа, така и към цели класове и типове.

- `public`: елементът (тип или член на клас или структура) е достъпен за всеки код в същото асембли или друго асембли, което го референсира;
- `private`: елементът е достъпен само за код от същия клас или структура;
- `protected`: елементът е достъпен за код от същия клас или структура или производен клас;
- `internal`: елементът е достъпен за всеки код от същото асембли, но не и за други асемблита.

Има още един модификатор – `protected internal`, който дава достъп на всеки код от същото асембли, както и на производни класове в друго асембли. Ще стане ясно, след като изучим наследяването.

Да променим класа `Circle`, като добавим модификатори на достъпа:

```
public class Circle
{
    private int radius;
    public double Area()
    {
        return Math.PI * radius * radius;
    }
}
```

Сега остана проблемът, че няма как да инициализираме радиуса, за да можем да пресмятаме повърхнината на кръга. Това обикновено се прави от специален метод, наречен конструктор, но ще разгледаме конструкторите след малко. Ако искаме да запазим полето `radius` капсулирано, трябва да осигурим достъп до него през `public` метод. В него може да се следи за коректността на данните:

```
public class Circle
    {
        private int radius;
        public double Area()
        {
             return Math.PI * radius * radius;
        }
        public void SetRadius(int r)
        {
             if (r > 0)
                   radius = r;
             else
                   throw new ArgumentOutOfRangeException("Incorrect radius value");
        }
    }
```

Виждаме, че методите на класа `Area()` и `SetRadius()` имат достъп по `private` полето `radius`.

```
c.SetRadius(5);
Console.WriteLine($"The area of circle is {c.Area()}");
Circle[] circles = new Circle[5];
for (int i = 0; i < circles.Length; i++)
{
    circles[i] = new Circle();   // Защото масивът е от референции.
    circles[i].SetRadius(i + 1); // Всяка от тях трябва да сочи обект
}
foreach (var circle in circles)
    Console.WriteLine($"Circle with area {circle.Area()}");

Console.WriteLine();
```

Достъпът до данните и методите на класа се осъществява чрез **оператора точка** (`.`). Той не е нужен единствено в случай, че съответният метод се извиква в тялото на друг метод от същия клас.

За разлика от локалните променливи, дефинирани в тялото на функция, които не се инициализират по подразбиране, полетата на класа се инициализират до `0`, `false` или `null`, в зависимост от техния тип. Въпреки това е добра практика инициализирането да се прави явно.

➢ Конструктори

Когато използваме `new` за създаване на обект, средата (*runtime*) трябва да изгради този обект, използвайки дефиницията на класа. Трябва да задели парче памет от операционната

система, да го запълни с дефинираните в класа данни и да извика конструктор, който да извърши необходимата инициализация.

Конструкторът (*instance constructor*) е специален метод, който се изпълнява автоматично при създаване на инстанция на класа. Името му съвпада с името на класа, може да има параметри, но не може да връща стойност, дори `void`. Всеки клас трябва да има конструктор. Ако програмистът не напише такъв, компилаторът автоматично създава подразбиращ се (*default*) `public` конструктор, който инициализира полетата с подразбиращи се стойности и извиква подразбиращия се конструктор на директния базов клас (ще стане ясно, когато почнем да учим наследяване).

Добавяме подразбиращ се конструктор за примерния клас `Circle`:

```
public Circle()  // default constructor
{
    radius = 0;
}
```

Резултатът от изпълнението му няма да се различава от изпълнението на създадения от компилатора конструктор, който би изглеждал така:

```
public Circle() : base() {}
```

Знаем, че полета от тип `int` се инициализират по подразбиране със стойност `0`.

Ако декларираме конструктора като `private`, обекти на класа не могат да бъдат създавани от функции извън класа. Това е полезно в някои ситуации, които няма да разглеждаме тук. Ще посочим само, че създаването на обекти може да се делегира на друг метод на класа, например:

```
private Circle() {}
public static Circle CreateCircle() => new Circle();
```

Тогава класът се инстанцира така

```
Circle c = Circle.CreateCircle();
```

Опитът за директно инстанциране извън класа (например в `Main()`) води до грешка

```
error CS0122: 'Circle.Circle()' is inaccessible due to its protection level
```

Конструктори с параметри. Предефиниране на конструктори

Конструкторите също са методи на класа, макар и малко по-специални, и като такива могат да бъдат предефинирани. Един клас може да има много конструктори, като единственото ограничение е всички те да се различават по брой или тип на параметрите си.

При наличието на няколко конструктора възниква въпросът кой от тях ще се извика при създаването на обект. Този проблем се решава както и при предефинираните методи – компилаторът автоматично избира на базата на подадените параметри. Прилага се принципът на най-доброто съвпадение.

Добавяме конструктор с параметър към класа `Circle`:

```
public Circle(int r)
{
    radius = r;
}
```

Обекти с него създаваме така:

```
Circle c2 = new Circle(45);
```

При дефинирането на конструктори трябва да се съобразяваме с една важна особеност – ако дефинираме собствен конструктор (с или без параметри), компилаторът не създава автоматично *default* конструктор. Така че, ако сме дефинирали конструктор с параметри, но искаме да имаме и *default* такъв, трябва да дефинираме и него, а не да разчитаме на компилатора да стори това.

Ключовата дума `this` и методите на класовете

При дефинирането на конструктор може да се получи ситуация, в която името на параметъра съвпада с името на поле на класа:

```
public Circle(int radius)
{
   radius = radius;
}
```

Този код ще се компилира, макар и с предупреждение от Visual Studio, че "*Assignment made to the same variable*". Казахме, че в тялото на методите параметрите могат да се използват като всички останали локални променливи. В този случай присвояване на параметъра собствената му стойност.

В тялото на методите може да се използва ключовата дума `this`, за да се квалифицира името на поле или метод, когато това име се скрива от друго, както в горния пример:

```
public Circle(int radius)
{
    this.radius = radius;
}
```

В C# 7 (Visual Studio 2017) можем да използваме и ламбда синтаксиса в конструктори. Тогава горният конструктор може да се запише по-кратко така:

```
public Circle(int radius) => this.radius = radius;
```

Когато се използва в контекста на обектите, `this` е референция на текущата инстанция на класа (обекта, за който е извикан методът). Друго типично приложение, освен при скриване на име, е за предаване на обекта като параметър към друга функция. Тази Ключова дума се използва в още няколко случая с различно значение, които ще разгледаме по-късно:

- като модификатор на първия параметър на *extension* метод;
- за дефиниране на индексатор (*indexer*).

Един конструктор може да извика друг върху същия обект чрез ключовата дума `this`. Тя може да се използва с или без параметри, в зависимост от това кой конструктор искаме да извикаме.

```
public Circle() : this(0) {}
public Circle(int radius)
{
    this.radius = radius;
}
```

Ключовата дума `this` може да се използва не само в конструктори. Ще пренапишем метода `SetRadius()`:

```
public void SetRadius(int radius)
{
    if (radius > 0)
        this.radius = radius;
```

```
            else
                throw new ArgumentOutOfRangeException("Incorrect radius value");
        }
```

readonly полета

В декларациите на полета може да се използва модификатора `readonly`. Стойност на такива полета може да се присвоява само в декларацията им или в конструктор. Ключовата дума `readonly` се различава от `const` по своето действие – `const` полета могат да се инициализират само в декларацията им, докато `readonly` могат и в конструктор. Така те могат да имат различна стойност в зависимост от това кой конструктор ги инициализира. Също така `const` полетата са *compile-time* константи, докато `readonly` поле може да получи стойност по време на изпълнение на програмата, както в следния пример:

```
public class Car
{
    string model;
    readonly int year;
    public readonly int Age;
    public Car(string model, int year)
    {
        this.model = model;
        this.year = year;
        this.Age = DateTime.Now.Year - year;
    }
    public void WriteLine()
    {
        Console.WriteLine($"Car {this.model},
              {this.year}, ({this.Age} years old)");
    }
}
```

И в `Main()`:

```
Console.WriteLine();
Car myCar = new Car("Citroen C5", 2009);
myCar.WriteLine();
```

Ако направим опит повторно да присвоим стойност на `public` полето `Age`, например

```
myCar.Age = 5;
```

ще получим съобщение за грешка

```
error CS0191: A readonly field cannot be assigned to (except in a constructor or a variable initializer)
```

- **Деструктори**

Деструкторите се използват от .NET за разчистване след обектите. Рядко се налага да пишем деструктор за наш клас, защото *garbage collector*-а управлява заделянето и освобождаването на памет за нашите обекти. Когато обаче нашето приложение държи *unmanaged* ресурси като прозорци, файлове, мрежови връзки, трябва да се използва деструктор, за да освободи тези ресурси.

Особености на деструкторите

- деструктори се дефинират само за класове (не и за структури);
- един клас може да има само един деструктор;
- името на деструктора съвпада с името на класа с добавен отпред знак ~ (тилда);

- деструкторите не могат да бъдат наследени или предефинирани;
- деструктори не могат да бъдат извиквани. Те се извикват автоматично;
- деструкторите нямат параметри и модификатори на достъпа.

Ще дефинираме деструктор за класа `Circle`, макар точно за такъв клас той да не е необходим.

```
~Circle()
{
    Console.WriteLine($"Circle with radius {this.radius} destruction...");
}

static void Main(string[] args)
{
    Circle c1, c2;
    c1 = new Circle();
    c2 = new Circle(5);
    c1.SetRadius(10);
    Console.WriteLine(c1.Area());
    Console.WriteLine(c2.Area());
}
```

За обектите `c1` и `c2` деструкторът ще се изпълни автоматично при приключване на изпълнението на програмата. Програмистът няма контрол върху това кога ще се извика деструктора, защото това се определя от garbage колектора, който проверява за обекти, които вече не се използват. Ако определи някой обект като приемлив за изтриване, garbage колектора извиква деструктора му (ако има такъв) и освобождава паметта, използвана за съхранението на обекта.

Програмистът може да инициира принудително *garbage collection*, извиквайки метода `GC.Collect()`, но в повечето случаи това трябва да се избягва, защото може да предизвика проблеми с производителността.

```
static void Main(string[] args)
{
    ...
    Console.WriteLine(c1.Area());
    Console.WriteLine(c2.Area());
    GC.Collect();
    Console.ReadKey();
}
```

За да се демонстрира, че това работи, програмата трябва да се билдне в *Release* версия.

Всеки деструктор имплицитно извиква метода `Finalize()` на своя базов клас. Това става рекурсивно нагоре по йерархията. Тоест компилаторът ще преобразува нашия деструктор в метод `Finalize()`.

```
protected override void Finalize()
{
    try
    {
        // Почистващ код
    }
    finally
    {
        base.Finalize();
    }
}
```

Ние не можем да извикваме пряко `Finalize()`.

Ако приложението използва скъпи външни ресурси, има начин за явно (*explicit*) освобождаване на ресурса преди *garbage* колектора да освободи обекта. Това се прави с имплементиране на метода `Dispose()` от интерфейса `IDisposable`. Това може чувствително да подобри производителността на приложението. Дори и при такъв явен контрол върху ресурсите, деструкторът гарантира, че ресурсите ще бъдат освободени, ако извикването на `Dispose()` се провали.

- **Статични методи, данни и класове**

Досега сте използвали много статични данни и функции. Статична е и функцията `Main()`. Статичен е класът `Math`, както и всички негови функции (`Abs()`, `Exp()`, `Log()` , `Sin()` , `Cos()` , `Pow()`, `Sqrt()`) и константи (`PI`, `E`). Статичен е класът `System.Console` и неговите методи, например `WriteLine()`. Характерно за статичните елементи е, че те не са естествено свързани с конкретен обект. Ако математическите функции бяха реализирани като инстантни методи, вместо като статични, за да ги използваме трябваше да създаваме обект на класа `Math` и чрез него да извикваме метода:

```
Math m = new Math();
double d = m.Sqrt(42.24);
```

Статичните членове на класа (полета, свойства и методи) могат да се разглеждат като общи за всички негови обекти. Статичните полета и свойства позволяват достъп до данни, които не зависят от конкретни инстанции, а статичните методи позволяват да се изпълняват команди, свързани с класа, но не и с конкретни негови обекти.

Не се позволява достъп до статични членове чрез обекти на класа. Достъпът отново е с операцията точка, но приложена към името на класа. Статичните методи пък нямат достъп до членове на класа, които не са статични. Инстантните методи имат достъп до статичните членове на същия клас без да се налага да използват квалифицирано име.

Ще добавим към класа Circle възможност да броим инстанцираните обекти. За целта дефинираме статично поле `circlesCount` , инициализирано със стойност `0`. Конструкторите увеличават тази стойност с 1, а деструкторите я намаляват.

```
public class Circle
{
    static int circlesCount = 0;
    int radius;

    public Circle() : this(0) {}
    public Circle(int radius)
    {
        this.radius = radius;
        circlesCount++;
    }
    ~Circle()
    {
        circlesCount--;
    }
    public static int GetCirclesCount() => circlesCount;
}
```

Подразбиращия се конструктор разчита на извикания от него конструктор с параметър да увеличи броя на инстанциите. Ето и обръщение към статичния метод:

```
Console.WriteLine($"{Circle.GetCirclesCount()} circles");
```

Полета от ограничен брой типове (числови, низове и изброявания) могат да се декларират с ключовата дума `const`. Тези полета също са статични (въпреки, че не се използва `static`), но тяхната стойност не може да се променя. Така например в класа `Math` е дефинирана константата PI:

```
public const double PI = 3.1415926535897931;
```

Статични класове. Статични конструктори.

Класове, които имат само статични членове могат да се декларират като статични. Предназначението на такива класове е да бъдат контейнери на помощни методи и полета с данни. Такива класове не могат да се инстанцират. Ако е необходима някаква инициализация, статичният клас може да има подразбиращ се конструктор (без параметри), който също се декларира като статичен и не може да има модификатори на достъпа. Други конструктори не са позволени. Забранено е и наследяването на статични класове. Статичните данни могат да бъдат инициализирани и директно в рамките на декларацията им, но някога може да се наложи по-сложна инициализация, която е по-подходящо да се изпълни в рамките на конструктора.

Статичен конструктор могат да имат и класове, които не са статични (съдържат инстантни методи и данни). Този конструктор се изпълнява еднократно преди да бъде създадена първата инстанция на класа или преди да бъде референсиран някой статичен член. Изискванията са същите, както за статичен конструктор на статичен клас – не може да има параметри и модификатори на достъпа. По принцип този конструктор трябва да инициализира само статичните данни на класа, но може и да извърши някоя операция, която трябва да бъде изпълнена само веднъж. Статичен конструктор не може да бъде извикван директно и потребителят няма контрол кога ще се изпълни той.

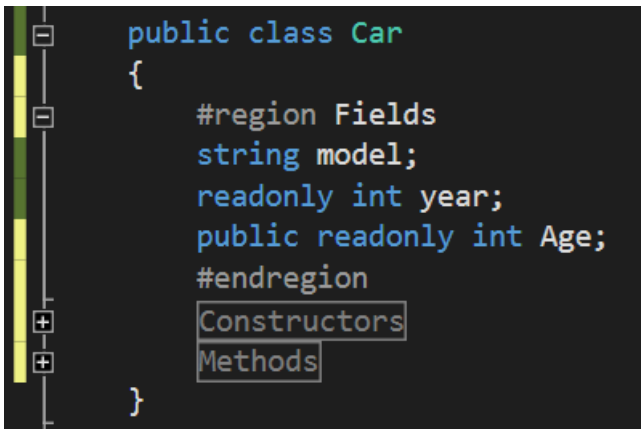
Ще променим класа `Circle`, така че статичното поле `circlesCount` да се инициализира в статичния конструктор.

```
public class Circle
{
   static int circlesCount; // без инициализация тук
...

   static Circle()     // статичен конструктор
   {
       circlesCount = 0;
   }
```

- **Частично дефинирани класове (*Partial Class Definitions*)**

Дефинирането на класове с множество полета, свойства и методи може да направи сорс файловете много големи и да затрудни четенето на кода. Едно решение е да се използват региони. Ето как ще изглежда класът `Car` с 3 региона, единият от които е отворен:



Фиг. 1. Региони

Алтернатива на този подход е разделянето на дефиницията на класа в няколко файла - *partial class definitions*. Това става с ключовата дума `partial`, която обявява класа за частично дефиниран. Тя трябва да се използва във всеки файл, съдържащ частична дефиниция на класа. Например полетата и конструкторите може да са в един файл, а методите – в друг.

```
public partial class Car
{

    string model;
    readonly int year;
    public readonly int Age;

    public Car(string model, int year)
    {
        this.model = model;
        this.year = year;
        this.Age = DateTime.Now.Year - year;
    }

}
```

И в другия файл:

```
public partial class Car
{
    public void WriteLine()
    {
        Console.WriteLine($"Car {this.model},
              {this.year}, ({this.Age} years old)");
    }
}
```

Класът е обединение на всички частични дефиниции.

Тази възможност се използва от *Microsoft Visual Studio* например за *Windows Presentation Foundation (WPF)* и *Windows Store* приложения, при които сорс кода, който програмиста може да редактира се поддържа в отделен файл от кода, генериран от *Visual Studio*, когато подредбата на формата се променя.

Възможно е да има и частични дефиниции на методи (*partial methods*), но ще ги разгледаме по-късно.

- **Анонимни типове (анонимни класове)**

Анонимен клас е клас, който няма име. Такива класове са удобни за капсулиране на множество от *read-only* свойства в един обект, без явно да дефинираме неговия тип. Името на класа (типа) се генерира от компилатора и не е достъпно на ниво сорс код. Типът на всяко поле (свойство) се извлича от компилатора по стойността, с която е инициализирано. Анонимен клас се създава с оператора `new` и инициализатор на обект:

```
var anonymousPerson = new { Name = "John", Age = 47 };
```

Така създаденият клас съдържа две `public` полета – `Name`, инициализирано с низа `"John"` и `Age`, инициализирано с `int` стойността 47). Тъй като няма как да знаем името на анонимния клас, дефинираме променлива от този тип с ключовата дума `var`. Така типът на обекта се определя от инициализатора.

Достъпът до полетата става по същия начин – с операцията точка:

```
Console.WriteLine($"{anonymousPerson.Name}, {anonymousPerson.Age}");
```

Ако дефинираме още един такъв обект, той ще е от същия тип

```
var anonymousPerson2 = new { Name = "Ivan", Age = 23 };
```

И дори е възможно присвояване между тях:

```
anonymousPerson = anonymousPerson2;
```

Анонимни типове се използват често в *select* клаузи на *query* изрази, които връщат подмножество от полета за всеки обект в дадена колекция.

Ще подчертаем, че анонимните класове могат да съдържат само `public` *read-only* свойства и никакви други членове като методи или събития. Изразът, който се използва за инициализиране на свойство не трябва да бъде `null`, анонимна функция или указател.

Ако направим опит да променим стойност на свойство,

```
anonymousPerson.Age = 27;
```

ще получим грешка:

```
error CS0200: Property or indexer '<anonymous type: string Name, int Age>.Age' cannot be assigned to -- it is read only
```

- **Свойства (*properties*)**

В „чистото“ ООП всички функции са методи (както в езика *Smalltalk*). В C# методите са само един от видовете членове-функции (*function members*). Според спецификацията на езика *function members* са членове, които съдържат изпълними оператори. Те са разделени на 8 категории:

- методи (*methods*)
- свойства (*properties*)
- събития (*events*)
- индексатори (*indexers*)
- предефинирани операции (*user-defined operators*)
- конструктори (*instance constructors*)
- статични конструктори (*static constructors*)
- деструктори (*destructors*)

Свойството е член, който предоставя достъп до характеристика на обект или клас. Можем да кажем, че капсулира част от състоянието на обекта. За примери можем да посочим дължина на низ, размер на шрифт, заглавие на прозорец, име на клиент и т.н. Свойствата са естествено разширение на полетата (*fields*) – и двете са именувани членове с асоциирани типове и синтаксисът за достъп до тях е същият. За разлика от полетата обаче свойствата не обозначават места за съхранение на данните (*storage locations*). Вместо това свойствата имат аксесори (*accessors*), които определят оператори, които да бъдат изпълнени, когато техните данни се четат или записват. По този начин свойствата предоставят механизъм за асоцииране на действия с четенето и записването на атрибути на обекта и нещо повече – позволяват тези атрибути да се изчисляват, както ще видим в примерите по-нататък.

По принцип е по-добре да се предоставят свойства вместо полета за достъп до състоянието на обекта, защото така имаме повече контрол върху различни ситуации (поведение) – може да се извърши предварителна обработка, преди да се промени състоянието.

Декларацията на свойство в най-простия ѝ вид изглежда така:

```
модификатор тип Име
{
    get
    {
        // read accessor code
    }
    set
    {
        // write accessor code

    }
}
```

Модификаторът е за достъпа – `public`, `private`…

Свойството съдържа два блока с код, които започват с ключовите думи `get` и `set`. `get` блокът съдържа оператори, които се изпълняват, когато свойството се чете, а `set` блокът – когато се записва в свойството. Типът на свойството определя типа на данните, които се четат и записват с аксесорите. `get` блокът трябва да върне с `return` стойност от тип типа на свойството. Обикновено простите свойства се асоциират с единично `private` поле, контролирайки достъпа до него. Така `set` блокът може да зададе на съответното поле получената от потребителя на свойството стойност чрез ключовата дума `value`. Да разгледаме пример:

```
public class ScreenPosition
{
    private int x, y;
    public ScreenPosition(int x, int y)
    {
        this.x = rangeCheckedX(x);
        this.y = rangeCheckedY(y);
    }
    public int X
    {
        get
        {
            return x;
        }
        set
        {
            x = rangeCheckedX(value);
        }
```

```
        }
        public int Y
        {
            get
            {
                return y;
            }
            set
            {
                y = rangeCheckedY (value);
            }
        }
        private static int rangeCheckedX(int x)
        {
            if (x >= 0 && x <= 1920)
                return x;
            else
                throw (new ArgumentOutOfRangeException("X", "argument is out of
range"));
        }
        private static int rangeCheckedY(int y)
        {
            if (y >= 0 && y <= 1080)
                return y;
            else
                throw (new ArgumentOutOfRangeException("Y","argument is out of
range"));
        }
    }
```

Тук `set` аксесорите, както и конструкторът, използват статичните методи `rangeCheckedX()`, за да проверяват коректността на данните. `public` свойствата `X` и `Y` са свързани с `private` полетата `x` и `y`. Ето как можем да използваме тези свойства:

```
ScreenPosition pos = new ScreenPosition(50, 150);
Console.WriteLine($"point at possition ({pos.X},{pos.Y})");
// Let's move that point to new position
pos.X = 100; pos.Y = 200;
Console.WriteLine($"point at possition ({pos.X},{pos.Y})");
```

Полетата `x` и `y` са недостъпни за методи извън класа, понеже са `private`, но достъпът през свойствата е възможен.

Аксесорите могат да имат собствени модификатори на достъпа, като възможните режими зависят от модификатора на свойството. Те могат да бъдат само по-ограничаващи от режима на пропъртито.

Що се отнася до именуването на свойствата, препоръчва се да се използва конвенцията *PascalCase*, както и при методите.

Възможно е да се декларират **статични свойства**, подобно на статичните полета. Достъпът до тях се осъществява чрез името на класа, а не чрез конкретен обект.

Read-only и *write-only* свойства

Допустимо е свойствата да имат само по един аксесор. Ако искаме *read-only* свойство, трябва да му дефинираме само `get` аксесор. Всеки опит да се присвои стойност на такова свойство ще води до грешка по време на компилация.

Пример: Ще добавим *read-only* свойство `Diameter` на класа `Circle`:

```
private int radius;
public int Diameter
{
    get
    {
        return radius * 2;
    }
}
```

Ако работим с C# 6, можем да ползваме за такива ситуации *expression based properties*, подобно на *expression based methods*. Тогава същото свойство се имплементира на един ред така:

```
public int Diameter => radius * 2;
```

Ако дефинираме само `set` аксесор ще имаме *write-only* свойство. Такива свойства са полезни за защита на данните например за пароли. Приложението трябва да ни позволи да зададем паролата, но никога не трябва да ни дава да я прочетем отново. При опит за логване потребителят предоставя парола и *logon* методът може да сравни тази парола със съхранената такава и само да върне индикация дали двете пароли съвпадат.

Да се върнем на класа `ScreenPosition`. В C# 7 можем да дефинираме неговите пропъртита така:

```
public int X { get => x; set => x = rangeCheckedX(value); }
public int Y { get => y; set => y = rangeCheckedY(value); }
```

Ограничения, свързани със свойствата

- стойност на свойство на клас или структура може да се присвои само след като класа или структурата са инициализирани;

Пример:

```
ScreenPosition location;
location.X = 5; // error CS0165: Use of unassigned local variable 'location'
```

- свойство не може да се предава като `ref` или `out` аргумент на функция (докато поле може);

Това е така, защото пропъртито в действителност не сочи към обект в паметта, а към функция-аксесор.

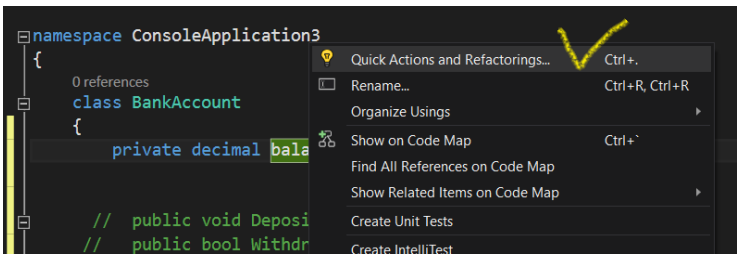
- свойство не може да съдържа други методи, свойства или полета, освен един `get` и един `set` аксесор;
- аксесорите не могат да имат параметри;
- не могат да се дефинират `const` свойства.

Рефакториране на членовете

Една полезна техника е възможността средата автоматично да генерира код за деклариране на свойство от поле. Нека имаме следния клас:

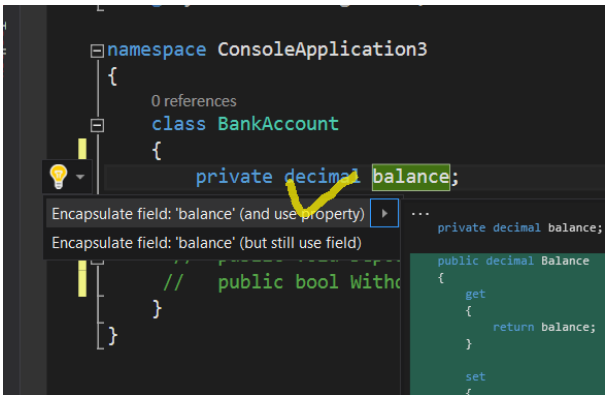
```
class BankAccount
{
    private decimal balance;
    //...
```

Искаме свойство, което да управлява достъпа до полето `balance`.Това с тава с десен бутон върху името му и избираме Quick Actions and Refactoring.



Фиг. 2. Рефакториране - автоматично генериране на свойство

Избираме първото от предложените ни действия:



Фиг. 3. Генериране на свойство от поле

Автоматично ще се добави код и класът ще изглежда така:

```
class BankAccount
{
    private decimal balance;
    public decimal Balance {get => balance; set => balance = value;}
//.....................................................................
}
```

Друга възможност за рефакториране е замяната на методи със свойства. Например за класа `Circle` може вместо методите `SetRadius()` и `GetRadius()` да се дефинира свойство `Radius`, свързано с `private` полето `radius`, което да контролира достъпа до него.

Автоматични свойства

Автоматично генерираният код от предишния пример е доста често срещана ситуация - `private` поле, достъпвано чрез `public` свойство. Затова към езика е добавена още едно улеснение – автоматичните свойства. Идеята е да се опрости синтаксиса – ние декларираме свойството частично, а компилаторът допълва останалото. По-специално, компилаторът декларира `private` поле, което ще се използва за съхранение на данните и го използва за `get` и `set` блоковете на пропъртито.

```
class BankAccount
 {
     public decimal Balance { get; set; }
     //...
```

Дефинираме само достъпа, типа и името на свойството, но не и имплементация на аксесорите и прилежащото `private` поле. За декларирането на автоматично свойство можем да използваме *code snippet* – написваме `prop` и натискаме два пъти клавиша *Tab*.

При автоматичните свойства нямаме достъп до `private` полето, защото не знаем неговото име. Можем да контролираме достъпа чрез модификатори на аксесорите, например:

```
public Decimal Balance { get; private set; }
```

Автоматичното свойство задължително има `get` аксесор, а `set` аксесора може да се пропусне. В такъв случай пропъртито се нарича *getter-only auto-property* и прилежащото му поле се счита `readonly`. Това означава, че на пропъртито може да се присвои стойност само в тялото на конструктор на класа, в който е декларирано. Допустимо е автоматичното свойство да има инициализатор:

```
public class Point
{
    public int X { get; set; } = 0;
    public int Y { get; set; } = 0;
}
```

Стойността директно се присвоява на прилежащото поле.

И един пример с *getter-only*:

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

Този код е еквивалентен на:

```
public class ReadOnlyPoint
{
    private readonly int __x;
    private readonly int __y;
    public int X { get { return __x; } }
    public int Y { get { return __y; } }
    public ReadOnlyPoint(int x, int y) { __x = x; __y = y; }
}
```

Като малко отклонение ще покажем как чрез рефлексия всъщност можем да променяме стойностите на *getter-only* свойства, използвайки прилежащите им полета. Нека имаме първия вариант на класа `ReadOnlyPoint`. Тогава:

```
ReadOnlyPoint rop = new ReadOnlyPoint(1, 2);
Console.WriteLine($"({rop.X},{rop.Y})");    // (1,2)
var fields = typeof(ReadOnlyPoint).GetFields(
             System.Reflection.BindingFlags.NonPublic |
             System.Reflection.BindingFlags.Instance);
foreach(var field in fields)
    Console.WriteLine(field.Name);  // <X>k__BackingField и т.н.
fields[0].SetValue(rop, 10);
fields[1].SetValue(rop, 20);
Console.WriteLine($"({rop.X},{rop.Y})"); // (10,20)
```

Пример: Класове `Point` и `Line`, представящи съответно точка и отсечка в равнината:

Файл Point.cs:

```
using System;

namespace DefiningClasses
```

```
{
    public class Point
    {
        private double x, y;
        public Point(double x, double y)
        {
            this.x = x;
            this.y = y;
        }
        public Point() : this(0.0, 0.0) { }
        public void SetPoint(double x, double y)
        {
            this.x = x;
            this.y = y;
        }
        public override string ToString() => "(" + x + "," + y + ")";
        public double DistanceTo(Point p) => Math.Sqrt(Math.Pow(x - p.x, 2) +
Math.Pow(y - p.y, 2));
        public double X => x;  // read-only property
        public double Y => y;  // read-only property
    }
}
```

Файл Line.cs:

```
namespace DefiningClasses
{
    class Line
    {
        private Point A, B;
        public double Length => A.DistanceTo(B);   // read-only property
        public Line(Point p1, Point p2)
        {
            A = new Point(p1.X, p1.Y); // за да имаме копие на точката p1 (Point е
референтен тип!)
            B = new Point(p2.X, p2.Y); // не трябва да се пише директно A = p1; B =
p2;
        }
        public Line(double x1, double y1, double x2, double y2)
        {
            A = new Point(x1, y1);  // A и B са референции, които трябва да сочат към
            B = new Point(x2, y2);  // действителен обект
        }
        public override string ToString() => $"Line from point {A} to point {B}";
    }
}
```

Файл Program.cs

```
using System;
namespace DefiningClasses
{
    class Program
    {
        static void Main(string[] args)
        {
            Point A = new Point();
            Point B = new Point(5, 10);
            Console.WriteLine($"Point A: {A}");   // (0,0)
            Console.WriteLine($"Point B: {B}");   // (5,10);

            Line L = new Line(A, B);
```

```
            Console.WriteLine(L);
            Console.WriteLine($"Length: {L.Length}");

            A.SetPoint(1, 1);
            Console.WriteLine(L);

            Line LL = new Line(1, 2, 3, 4);
            Console.WriteLine(LL);
            Console.ReadKey();
        }
    }
}
```

За упражнение променете класа `Point`, така че свойствата `X` и `Y` да са автоматични (без съответни полета x и y). Добавете метод , `Clone()` който връща като резултат копие на точката (нова точка със същите координати.

Изпробвайте поведението на следния код:

```
Point A = new Point(1, 2);
Point B = A.Clone(); //Point B = A;
A.X = A.Y = 0.0;
Console.WriteLine($"A: {A}, B: {B}");
```

Добавете метод `Deconstruct()` и демонстрирайте използването на *tuple* за деконструкция на класа `Point`. (C# 7)