

Класове

Клас – прототип, който дефинира данните и методите за всички обекти от даден вид.

1. Дефиниране на клас

```
[атрибути] [модификатори] class <име_на_клас> [:<име_на_базов_клас>]
{
    // тяло на клас
};
```

2. Членове на клас

- **поле** (член-променлива) – съхранява стойност; модификатори: **static**, **readonly** и **const**;
- **метод** (член-функция) – код за обработка на полетата;
- **свойство** – метод, който изглежда като поле за потребителя;
- **константа** – поле, чиято стойност не може да се променя;
- **индексатор** – член, който позволява да се работи с класа, сякаш самият клас е масив;
- **събитие** – предизвиква изпълнение на даден код, когато се случи;
- **оператор** – стандартен математически оператор към класа чрез предефиниране на оператори.

3. Модификатори на достъпа

- public** членът е достъпен извън дефиницията на класа и неговите производни класове;
- protected** членът е невидим извън класа, но е достъпен от производните му класове;
- private** членът е недостъпен извън класа и неговите производни класове (по подразбиране);
- internal** членът е видим само в текущата компилационна единица.

4. Метод **Main** – входна точка за приложението; трябва да бъде **static**.

- а) аргументи на командния ред – масив от низове;
- б) върнатата стойност
 - обикновено не връща стойност – **void**;
 - стойност от тип **int** – показва ниво на грешка (за конзолни приложения);
- в) може да се дефинира в няколко класа; ключът `/main:<име_на_клас>` определя класа, чийто метод **Main** ще се използва.

5. Конструктори

- а) извикват се при създаване на екземпляр на клас чрез **new**;
- б) имат същото име като името на класа;
- в) инициализират обекта;
- г) не връщат стойност.
`<клас> <обект> = new <клас> (аргументи на конструктор)`
new създава нов обект в:
 - хийпа – референтни типове;
 - стека – стойностите типове.

6. Статични членове и членове на инстанциите

- а) **член на инстанцията**
 - прави се копие на члена за всяка инстанция на класа (по подразбиране);
 - достъп: `<име_на_инстанция>.<име_на_член>`
- б) **статичен член (static)**
 - съществува само едно копие на члена;
 - създава се при зареждане на приложението;
 - съществува през цялото време;
 - членът е достъпен преди създаването на инстанция на класа;
 - достъп: `<име_на_клас>.<име_на_член>`

7. Инициализатори на конструктори

Конструкторът първо извиква конструктора на базовия клас. Инициализаторите определят на кой клас се извиква конструктора.

base (...) – извиква конструктор на базовия клас;

this (...) – извиква конструктор на текущия клас.

8. Константа

– поле, не се променя при изпълнение на приложението;

– дефинира се с **const**;

– стойността ѝ се установява при **компилация**;

– по подразбиране е статична.

9. Полета само за четене

readonly – константно поле, чиято стойност се установява в конструктора и се инициализира при **изпълнение**;

static readonly – константно поле, чиято стойност се установява в **статичен** конструктор, който по подразбиране е **public**.

Пример: Клас ТОЧКА – членове на екземплярите

```
using System;
class Point
{
    private int x, y; // Координати на точка
    // Конструктор без параметри
    public Point ()
    {
        x = 0;
        y = 0;
    }
    // Конструктор с два параметъра
    public Point (int initialX, int initialY)
    {
        x = initialX;
        y = initialY;
    }
    // Метод – изчислява разстоянието до точката other
    public double DistanceTo (Point other)
    {
        int xDiff = x - other.x;
        int yDiff = y - other.y;
        return Math.Sqrt (xDiff * xDiff + yDiff * yDiff);
    }
}
```

членове на инстанциите

```
// Предефинира метода System.Object.ToString
public override string ToString()
{
    return ("+x+", "+y+");
}
class PointApp
{
    static void Main (string[] args)
    {
        Point origin = new Point ();
        Point bottomRight = new Point (600,800);
        double distance = bottomRight.DistanceTo (origin);
        Console.WriteLine("Разстоянието между точките {0} и {1} е {2}.",
            bottomRight, origin, distance);
    }
}
```

Резултати:

Разстоянието между точките (600, 800) и (0, 0) е 1000.

Пример: Клас ТОЧКА – статични полета

```
using System;
class Point
{
    private int x, y;
    private static int objectCount = 0;
    public Point ()
    {
        x = 0;
        y = 0;
        objectCount++;
    }
    public Point (int initialX, int initialY)
    {
        x = initialX;
        y = initialY;
        objectCount++;
    }
}
```

статичен член

```
public double DistanceTo (Point other)
{
    int xDiff = x - other.x;
    int yDiff = y - other.y;
    return Math.Sqrt (xDiff * xDiff + yDiff * yDiff);
}
public override string ToString ()
{
    return ("+x+", "+y+");
}
public static int ObjectCount ()
{
    return objectCount;
}
```

статичен метод

```
class PointApp
{
    static void Main (string[] args)
    {
        Console.WriteLine ("Брой на обекти от тип ТОЧКА: {0}",
            Point.ObjectCount ());
        Point origin = new Point();
        Point bottomRight = new Point(600, 800);
        double distance = bottomRight.DistanceTo (origin);
        Console.WriteLine ("Разстоянието между {0} и {1} е {2}.",
            bottomRight, origin, distance);
        Console.WriteLine ("Брой на обекти от тип ТОЧКА: {0}",
            Point.ObjectCount ());
    }
}
```

Резултати:

Брой на обекти от тип ТОЧКА: 0
 Разстоянието между (600, 800) и (0, 0) е 1000.
 Брой на обекти от тип ТОЧКА: 2

Пример: Запазване на текущия IP адрес на работна станция (при динамично получаване на IP адрес):

- чрез **readonly** поле;
- чрез **static readonly** поле.

Клас System.Net.IPAddress – представя IP адрес.

Клас Dns – дава информация за дадена машина от Internet Domain Name System (DNS).

Метод Dns.Resolve – анализира името или адреса на инстанция от тип **IPHostEntry** (клас-контейнер, съдържащ информация за Интернет адресите). Когато името на машината е свързано с много IP адреси, методът връща само първия IP адрес.

`public static IPHostEntry Resolve (string hostName);`

Свойство IPHostEntry.AddressList – дава IP адреса на машината.

```
using System;
using System.Net;
class Workstation
{
    public const string HostName="FKSU2300A-3";
    public readonly string IPAddressString;
    public Workstation()
    {
        IPAddress ipAddress = Dns.Resolve(HostName).AddressList[0];
        IPAddressString=ipAddress.ToString();
    }
}
class GetIpAddress
{
    static void Main(string[] args)
    {
        Workstation workstation=new Workstation();
        Console.WriteLine("IP адресът на '{0}' е {1}.", Workstation.HostName,
            workstation.IPAddressString);
    }
}
```

Резултати:

IP адресът на 'FKSU2300A-3' е 81.161.244.49.

```
using System;
using System.Net;
class Workstation
{
    public const string HostName="FKSU2300A-3";
    public static readonly string IPAddressString;
    static Workstation()
    {
        IPAddress ipAddress = Dns.Resolve(HostName).AddressList[0];
        IPAddressString=ipAddress.ToString();
    }
}
class GetIpAddress
{
    static void Main(string[] args)
    {
        Console.WriteLine("IP адресът на '{0}' е {1}.", Workstation.HostName,
            Workstation.IPAddressString);
    }
}
```

Резултати:

IP адресът на 'FKSU2300A-3' е 81.161.244.49

Наследяване

Наследяване – даден клас е изграден въз основа на друг клас, като използва неговите данни и поведение.

`class <производен_клас> : <базов_клас>`

- наследяват се **public**, **protected** или **internal** членове;
- конструктор не се наследява – всеки наследник трябва да реализира свой собствен конструктор;
- **C#** не поддържа множествено наследяване чрез производни класове, а реализира много интерфейси;
- **запечатани класове (sealed)** – не могат да имат производни класове.

Пример: Класът ТРИМЕРНА ТОЧКА наследява класа ТОЧКА

```
using System;
class Point // Базов клас ТОЧКА
{ private int x,y;
  public Point ()
  { x = 0;
    y = 0;
  }
  public Point (int initialX, int initialY)
  { x = initialX;
    y = initialY;
  }
  public void Move (int dx, int dy)
  { x += dx;
    y += dy;
  }
  public override string ToString ()
  { return x+","+y; }
}
```

```
class Point3D : Point // Произведен клас ТРИМЕРНА ТОЧКА
{
  private int z;
  public Point3D () : base ()
  { z = 0;
  }
  public Point3D (int initialX, int initialY, int initialZ) : base(initialX, initialY)
  { z = initialZ;
  }
  public void Move (int dx, int dy, int dz)
  { base.Move (dx, dy);
    z += dz;
  }
  public override string ToString ()
  { return base.ToString () + "," + z;
  }
}
```

```
class PointApp
{
  static void Main (string[] args)
  {
    Point3D point = new Point3D (600,800,1000);
    point.Move (1, 1, 1);
    Console.WriteLine ("{(0)", point);
  }
}
Резултати:
(601,801,1001)
```

Структури

Структура

- стойностен тип;
- съдържа данни от различни типове;
- разглежда се като олекотена версия на клас.

1. Дефиниране на структура

```
[атрибути] [модификатори] struct <име_на_структура> [ : <интерфейси> ]
{
  // тяло на структура
}[;]
```

Структурите не се наследяват.

2. Членове на структура

- конструктори;
- константи;
- полета;
- методи;
- свойства;
- индексатори;
- оператори;
- вградени типове.

3. Ограничения

- не създава конструктор без параметри;
- не се наследява;
- получена е неявно от суперкласа на всички стойностни типове **System.ValueType**.

4. Използване

- малко количество данни;
- малък брой методи за достъп или модифициране на данните.

Пример: Дефиниране на структура RGB за често използвани цветове като статични членове.

```
using System;
struct RGB
{
  public static readonly RGB RED = new RGB(255,0,0);
  public static readonly RGB GREEN= new RGB(0,255,0);
  public static readonly RGB BLUE = new RGB(0,0,255);
  public static readonly RGB WHITE = new RGB(255,255,255);
  public static readonly RGB BLACK = new RGB(0,0,0);
  public int Red;
  public int Green;
  public int Blue;
  public RGB (int red, int green, int blue)
  { Red = red;
    Green = green;
    Blue = blue;
  }
  public override string ToString()
  { return Red.ToString("X2") + Green.ToString("X2") + Blue.ToString("X2");
  }
}
```

```
class StructApp
{
    static void PrintRGBValue(string color, RGB rgb)
    {
        Console.WriteLine("Стойността за {0} е {1}", color, rgb);
    }
    static void Main(string[] args)
    {
        PrintRGBValue ("червено",RGB.RED);
        PrintRGBValue ("зелено ",RGB.GREEN);
        PrintRGBValue ("синьо  ",RGB.BLUE);
        PrintRGBValue ("бяло  ",RGB.WHITE);
        PrintRGBValue ("черно  ",RGB.BLACK);
    }
}

```

Резултати:
 Стойността за червено е FF0000
 Стойността за зелено е 00FF00
 Стойността за синьо е 0000FF
 Стойността за бяло е FFFFFFFF
 Стойността за черно е 000000

Методи

Методи (член-функции) – определят поведението на класа.

Параметри на методите

Стойностни и референтни параметри

1. Предаване на стойностни параметри – по стойност:

- копие на стойността се предава на метода;
- измененията на стойностните параметри не влияят върху променливите, предадени от извикващия метод.

2. Предаване на референтни параметри – по адрес.

- копие на референтния параметър (т.е. друга референтна стойност към същите данни) се предава на метода;
- измененията на референтните параметри се отразява върху променливите, предадени от извикващия метод (т.е. промените се извършват върху оригиналните данни).

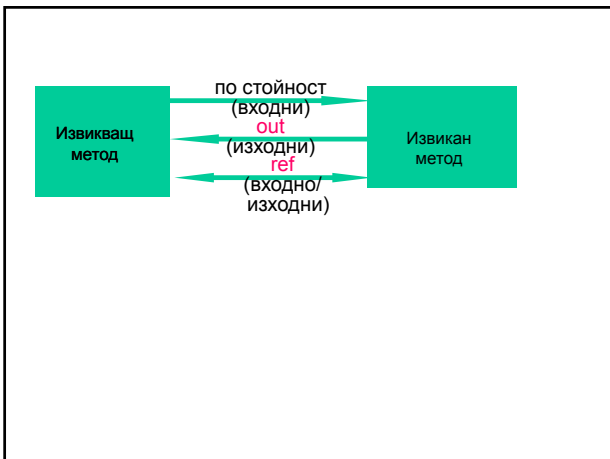
3. Връщане на много стойности

а) чрез ref параметри

- ref параметрите сочат към същата памет като променливите в извикващия код;
- измененията на ref параметрите се отразява върху променливите в извикващия метод (указатели в C++);
- ограничение – ref параметрите трябва да се инициализират преди извикване на метода.

б) чрез out параметри

- out параметрите не изискват първоначално инициализиране;
- задължително трябва да се модифицират в метода.



Пример: Предаване на стойностни параметри

```
using System;
class SomeClass
{
    public void Change(int x, ref int y, out int z)
    {
        x += 5; // входен параметър
        y *= x; // входно/изходен параметър
        z = 10 * y; // изходен параметър
    }
}
class Test
{
    static void Main()
    {
        SomeClass sc=new SomeClass();
        int v1 = 5, v2 = 5, v3;
        sc.Change(v1, ref v2, out v3);
        Console.WriteLine("v1={0}, v2={1}, v3={2}",v1,v2,v3);
    }
}

```

Резултати:
 v1=5, v2=50, v3=500

Пример: Предаване на референтни параметри.

```
using System;
class AnotherClass
{
    public int ID;
}
class SomeClass
{
    public void ChangeObject (AnotherClass x, ref AnotherClass y,
                             out AnotherClass z)
    {
        x.ID += 5;
        y.ID *= x.ID;
        z = new AnotherClass();
        z.ID = 10 * y.ID;
    }
}
```

```
class Test
{
    static void Main()
    {
        SomeClass sc = new SomeClass();
        AnotherClass r1 = new AnotherClass();
        r1.ID = 5;
        AnotherClass r2 = new AnotherClass();
        r2.ID = 5;
        AnotherClass r3;
        sc.ChangeObject (r1, ref r2, out r3);
        Console.WriteLine("r1.ID={0}, r2.ID={1}, r3.ID={2}", r1.ID, r2.ID, r3.ID);
    }
}
```

Резултати:

r1.ID=10, r2.ID=50, r3.ID=500

```
class SomeClass
{
    public void ChangeObject (AnotherClass x, ref AnotherClass y,
                             out AnotherClass z)
    {
        x = new AnotherClass();
        x.ID += 5;
        y.ID *= x.ID;
        z = new AnotherClass();
        z.ID = 10 * y.ID;
    }
}
```

Резултати:

r1.ID=5, r2.ID=25, r3.ID=250

Презареждане на методи (overloading)

Презареждане – многократно използване на едно и също име на метод с различни параметри.

1. Поведението на методите се различава слабо и зависи от типа на параметрите.
2. Списъкът от параметри е различен.
3. Типът и модификаторът за достъп на върнатия резултат трябва да са еднакви.

Презареждане на конструктори

```
class Point
{
    private int x;
    private int y;
    // Конструкторът public Point (int x, int y) неявно се компилира в
    // public Point (int x, int y) : this ().
    // this () извиква public Point ().
    public Point (int x, int y)
    {
        this.x = x;
        this.y = y;
    }
    public Point ()
    {
        x = 0;
        y = 0;
    }
}
```

Инициализатор на конструктор – извика явно конструктор:

```
public Point () : this (0, 0)
{
    ...
}
```

Наследяване и презареждане

Класът-наследник може да използва презареден метод (както в Java).

Методи с променлив брой параметри

Като аргумент на метода се използва ключовата дума `params` с масив.

Пример:

```
using System;
class Point
{
    public int x;
    public int y;
    public Point (int x, int y)
    {
        this.x=x;
        this.y=y;
    }
}
class Polygon
{
    public void DrawPolygon (params Point[] p)
    {
        Console.WriteLine("Многоъгълник с върхове: ");
        for (int i=0; i<p.GetLength(0); i++)
            Console.WriteLine ("{0},{1}", p[i].x, p[i].y);
    }
}
```

```
class TestPolygon
{
    static void Main()
    {
        Point p1 = new Point(5,10);
        Point p2 = new Point(5,15);
        Point p3 = new Point(5,20);
        Polygon p = new Polygon();
        p.DrawPolygon (p1, p2, p3);
    }
}
```

или

```
static void Main()
{
    Point[] pts = {new Point(5,10), new Point(5,15), new Point(5,20)};
    Polygon p = new Polygon();
    p.DrawPolygon (pts);
}
```

Пример: Параметри от произволен тип – params с масив от тип object.

```
using System;
class Point
{
    public int x;
    public int y;
    public Point (int x, int y)
    {
        this.x = x;
        this.y = y;
    }
}
class OpenEnded
{
    public void Foo (params object[] p)
    { for (int i=0; i<p.GetLength(0); i++)
        Console.WriteLine (p[i]);
    }
}
```

```
class TestOpenEnded
{
    static void Main()
    {
        OpenEnded oe = new OpenEnded();
        oe.Foo(123,456,"Здравей", new Point(7,8),9.0m,true,'X');
    }
}
```

Резултати:

```
123
456
Здравей
Point
9,0
True
X
```

Виртуални методи

Виртуални методи – модифицират поведението на базовия клас в производния клас.

- 1. Предефиниране на методи (overriding)** – чрез ключовата дума **new** – скрива метода в базовия клас (**new** се подразбира).

Пример: Предефиниране на метод чрез new

```
using System;
class Point
{
    protected int x = 0;
    protected int y = 0;
    public Point (int x, int y)
    {
        this.x = x;
        this.y = y;
    }
    public void Move (int dx, int dy)
    {
        x += dx;
        y += dy;
    }
    override public string ToString()
    {
        return ("x+y");
    }
}
```

```
class SlowPoint : Point
{
    private int xLimit;           // -xLimit <= x <= xLimit
    private int yLimit;           // -yLimit <= y <= yLimit
    public SlowPoint (int x, int y, int lower, int upper) : base (x, y)
    { xLimit = lower;
      yLimit = upper;
    }
    new public void Move (int dx, int dy)
    { x += dx;
      y += dy;
      x = Limit (x, xLimit);
      y = Limit (y, yLimit);
    }
    public int Limit (int d, int l)
    { // Ако координатата на точката d е по-голяма от горната граница l,
      // тя се ограничава от горната граница; ако d е по-малка от долната
      // граница -l, тя се ограничава от долната граница.
      return d > l ? l : d < -l ? -l : d;
    }
}
```

↑ предефинира Move чрез new

```
class TestOverriding
{
    static void Main (string[] args)
    {
        Point p1 = new Point (10, 20);
        p1.move (5, 5);
        Console.WriteLine (p1);
        SlowPoint p2 = new SlowPoint (100, 200, 500, 800);
        p2.move (600, 100);
        Console.WriteLine (p2);
    }
}
```

Резултати:
(15,25)
(500,300)

- 2. Полиморфизъм – много форми;**
- **предефиниране на метод в йерархията на класовете чрез ключовите думи:**
 - **virtual** в базовия клас и
 - **override** в производния клас;
 - **декларира се обект от базовия клас;**
 - **при изпълнение се извиква подходящият метод според използвания обект.**

Пример: Предефиниране на методи без полиморфизъм – чрез new

```
using System;
class Employee           // Служител
{
    private string name;           // Име
    private string address;        // Адрес
    private string phone;          // Телефонен номер
    private double payRate;        // Ставка

    public Employee (string name, string address, string phone, double payRate)
    { this.name = name;
      this.address = address;
      this.phone = phone;
      this.payRate = payRate;
    }

    public double pay()           // Плаща
    { return payRate; }

    override public string ToString()
    { return name + ", " + address + ", " + phone; }
}
```

```
class Executive : Employee // Президент на фирма
{
    private double bonus; // Допълнително възнаграждение
    public Executive (string name, string address, string phone,
                     double payRate, double bonus) :
        base(name, address, phone, payRate)
    {
        this.bonus = bonus;
    }
    new public double pay() // Предефинира Плаща
    {
        return base.pay() + bonus;
    }
}
```

```
class Hourly : Employee // Служител със заплащане на час
{
    private int hoursWorked; // Изработени часове
    public Hourly (string name, string address, string phone,
                  double payRate, int hoursWorked) :
        base (name, address, phone, payRate)
    {
        this.hoursWorked = hoursWorked;
    }
    new public double pay() // Предефинира Плаща
    {
        return base.pay() * hoursWorked;
    }
}
```



```
class TestNotPolymorphic
{
    static void Main()
    {
        Employee e;
        e = new Executive("Иван","София","1234567",400,100);
        Console.WriteLine(e + " " + e.pay());
        e = new Hourly("Мария", "Пловдив", "765432", 20, 10);
        Console.WriteLine(e + " " + e.pay());
    }
}
```

Резултати:

Иван, София, 1234567 400
Мария, Пловдив, 765432 20

Пример: Предефиниране на методи с полиморфизъм – чрез virtual и override

```
using System;
class Employee // Служител
{
    private string name;
    private string address;
    private string phone;
    private double payRate;
    public Employee (string name, string address, string phone, double payRate)
    {
        this.name = name;
        this.address = address;
        this.phone = phone;
        this.payRate = payRate;
    }
    virtual public double pay() // Виртуален метод Плаща
    {
        return payRate;
    }
    override public string ToString()
    {
        return name + ", " + address + ", " + phone; }
}
```

```
class Executive : Employee // Президент на фирма
{
    private double bonus;
    public Executive (string name, string address, string phone,
        double payRate, double bonus) :
        base(name, address, phone, payRate)
    {
        this.bonus = bonus;
    }
    override public double pay() // Предефинира виртуалния метод Плаща
    {
        return base.pay() + bonus;
    }
}
```

```
class Hourly : Employee // Служител със заплащане на час
{
    private int hoursWorked;
    public Hourly (string name, string address, string phone,
        double payRate, int hoursWorked) :
        base (name, address, phone, payRate)
    {
        this.hoursWorked = hoursWorked;
    }
    override public double pay() // Предефинира виртуалния метод Плаща
    {
        return base.pay() * hoursWorked;
    }
}
```

```
class TestPolymorphic
{
    static void Main()
    {
        Employee e;
        e = new Executive("Иван","София","1234567",400,100);
        Console.WriteLine(e + " " + e.pay());
        e = new Hourly("Мария","Пловдив","765432",20,10);
        Console.WriteLine(e + " " + e.pay());
    }
}
```

Резултати:

Иван, София, 1234567 500
Мария, Пловдив, 765432 200

Правила:

1. **override** метод трябва да има същото ниво на достъп (**protected**, **public**, **internal**), както **virtual** метода, който предефинира.
2. **virtual** член не може да се декларира **private**, защото няма да може да се предефинира. Може да се декларира **protected**, но тогава няма да може да се използва извън йерархията.

Пример: Предефиниране на методите на класа Object

```
using System;
class Point // Точка – наследява Object по подразбиране
{
    private int x, y;
    public Point ()
    {
        x = 0;
        y = 0;
    }
    public Point (int initialX, int initialY)
    {
        x = initialX;
        y = initialY;
    }
}
```

```
// Предефинира Object.Equals
public override bool Equals (Object obj)
{
    // Проверява за null и сравнява типовете по време на изпълнение
    if (obj == null || GetType () != obj.GetType ())
        return false;
    Point p = (Point)obj;
    return (x == p.x) && (y == p.y);
}
// Предефинира Object.GetHashCode
public override int GetHashCode ()
{
    // Генерира хеш код като комбинира координатите на точка чрез
    // операцията XOR
    return x ^ y;
}
// Предефинира Object.ToString
public override string ToString ()
{
    return x + ", " + y;
}
```

```
// Тримерна точка – наследник на Точка
class Point3D : Point
{
    private int z;
    public Point3D () : base ()
    {
        z = 0;
    }
    public Point3D (int initialX, int initialY, int initialZ) : base (initialX, initialY)
    {
        z = initialZ;
    }
    // Предефинира Point.Equals
    public override bool Equals (Object obj)
    {
        return base.Equals (obj) && z == ((Point3D)obj).z;
    }
    // Предефинира Point.GetHashCode
    public override int GetHashCode ()
    {
        return base.GetHashCode () ^ z;
    }
    // Предефинира Point.ToString
    public override string ToString ()
    {
        return base.ToString () + ", " + z;
    }
}
```

```
class InheritancePointApp
{
    static void Main (string[] args)
    {
        Point3D point1 = new Point3D (100, 100, 100);
        Console.WriteLine("Хеш код на т. {{0}}: {{1}}", point1, point1.GetHashCode());
        Point3D point2 = new Point3D (10, 10, 10);
        Console.WriteLine("Хеш код на т. {{0}}: {{1}}", point2, point2.GetHashCode());
        Point3D point3 = new Point3D (10, 10, 10);
        Console.WriteLine("Хеш код на т. {{0}}: {{1}}", point3, point3.GetHashCode());
        Console.WriteLine ("{{0}} and {{1}} {2}.", point1, point2,
            point1.Equals (point2) ? "са равни" : "не са равни");
        Console.WriteLine ("{{0}} and {{1}} {2}.", point2, point3,
            point2.Equals (point3) ? "са равни" : "не са равни");
    }
}
```

Резултати:
 Хеш код на т. (100,100,100): 100
 Хеш код на т. (10,10,10): 10
 Хеш код на т. (10,10,10): 10
 (100,100,100) и (10,10,10) не са равни.
 (10,10,10) и (10,10,10) са равни.

Статични методи

Статичен метод

- съществува в класа като цяло, а не в определен екземпляр на класа;
- дефинира се с ключовата дума **static**;
- извиква се без създаване екземпляр на класа:

<име_на_клас>.<име_на_метод>

1. Достъп до членове на класа

- а) има достъп само до статични класове на класа;
- б) няма достъп до членове на екземплярите.

2. Статични конструктори

- а) даден клас може да има само един статичен конструктор;
- б) няма параметри;
- в) няма достъп до нестатични членове (включително и указателя **this**);
- г) изпълнява се преди да се създаде първия екземпляр на класа;
- д) не може да има модификатор **public**;
- е) може да се използва статичен и нестатичен конструктор с една и съща сигнатура (първо се извиква статичният конструктор);
- ж) изпълнява се преди да се осъществи достъп до статичен член на класа (данни или метод).

Пример:

```
using System;
class Point
{
    private int x;
    private int y;
    private static int count;
    static Point() // Статичен конструктор
    {
        count = 0;
    }
    public Point() // Конструктор
    {
        x = 0;
        y = 0;
        count++;
    }
}
```

```
public Point (int x, int y) // Конструктор с два параметъра
{
    this.x = x;
    this.y = y;
    count++;
}
public void Move (int dx, int dy)
{
    x += dx;
    y += dy;
}
override public string ToString()
{
    return "(" + x + ", " + y + ")";
}
public static void Info() // Статичен метод
{
    Console.WriteLine("Броят на точките е " + count);
}
```

```
class TestStaticMethod
{
    static void Main()
    {
        Point.Info();
        Point p1 = new Point();
        p1.Move(5,5);
        Console.WriteLine(p1);
        Point p2 = new Point(100,200);
        p2.Move(50,50);
        Console.WriteLine(p2);
        Point.Info();
    }
}
```

Резултати:

Броят на точките е 0
(5,5)
(150,250)
Броят на точките е 2

Разширени методи

Разширен метод

- добавя методи към съществуващи типове без създаване на нов производен тип или прекомпилиране;
- специален вид статичен метод;
- извиква се чрез синтаксиса на методи на инстанциите.

Дефиниране и извикване на разширен метод

1. Дефиниране на **static class** (класът трябва да бъде видим за клиентския код).
2. Реализация на разширения метод като **static метод** с поне същата видимост като класа, в който е дефиниран.
3. Първият параметър на метода определя типа, върху който методът оперира; трябва да се предшества от модификатора **this**.
4. В извикващия код се добавя директивата **using**, за да се определи **namespace**, което съдържа класа на разширения метод.
5. Извиква се, използвайки синтаксиса на методите на инстанциите. Първият параметър не се описва в извикващия код, защото той представя типа, за който методът се прилага. Предават се аргументите за параметрите от 2-рия до n-тия.

```
using System;
namespace StringExtensionMethods
{
    public static class StringExtension
    {
        public static int WordCount(this String str)
        {
            return str.Split(new char[] { ',', '.', '?' },
                StringSplitOptions.RemoveEmptyEntries).Length;
        }
    }
}
namespace ExtensionMethods
{
    using StringExtensionMethods;
    class Program
    {
        static void Main(string[] args)
        {
            string s = "Това е текст за разширени методи";
            int count = s.WordCount();
            Console.WriteLine("Брой думи = " + count);
        }
    }
}
```

Брой думи = 6

Свързване на разширени методи по време на компилация

- разширените методи разширяват клас или интерфейс, но не ги предефинират;
- разширен метод със същото име и сигнатура като метод на интерфейс или клас никога няма да бъде извикан;
- по време на компилация компилаторът първо търси съвпадение с методите на инстанциите на класа. Ако не намери съвпадение, той търси за разширени методи, които са дефинирани за типа и се свързва към първия разширен метод, който намери.

```
using System;
namespace ExtensionMethods
{
    using StringExtensionMethods;
    class Program
    {
        private String str;
        public Program(String str)
        {
            this.str = str;
        }
        public int WordCount()
        {
            Console.WriteLine("Извикване на метод на инстанциите");
            return str.Split(new char[] { ' ', '.', '?' },
                StringSplitOptions.RemoveEmptyEntries).Length;
        }
        static void Main(string[] args)
        {
            Program s = new Program("Това е тест за разширени методи");
            int count = s.WordCount();
            Console.WriteLine("Брой думи = " + count);
        }
    }
}
```

Извикване на метод на инстанциите
Брой думи = 6

Абстрактен клас

Абстрактен клас (abstract) – дефинира клас за наследяване;

- не създава екземпляр;
- абстрактните членове се дефинират като **abstract**;
- задължително се реализират в производните класове чрез модификатора **override**;
- осигурява правилно дефиниране на производните класове.

Пример:

```
using System;
abstract class RoundShape // Абстрактен клас Кръгла фигура
{
    protected class Center // Вграден клас Център
    {
        public int x;
        public int y;
    }
    protected Center c = new Center(); // Център
    protected float radius; // Радиус
    abstract public float area(); // Абстрактен метод Площ
    public RoundShape(int x, int y, float r)
    {
        c.x = x;
        c.y = y;
        radius = r;
    }
}
```

```
class Circle : RoundShape // Кръг
{
    public Circle(int x, int y, float r) : base(x, y, r) {}
    public override float area() // Реализация на Площ
    {
        return (float)(Math.PI*Math.Pow((double)radius, 2.0));
    }
}
class Sphere : RoundShape // Сфера
{
    public Sphere(int x, int y, float r) : base(x, y, r) {}
    public override float area() // Реализация на Площ
    {
        return (float)(4*Math.PI*Math.Pow((double)radius, 2.0));
    }
}
```

```
class Shape
{
    static void Main()
    {
        Circle c = new Circle(5, 5, 10.0F);
        Sphere s = new Sphere(5, 5, 10.0F);
        Console.WriteLine("Площта на кръга е " + c.area());
        Console.WriteLine("Площта на сферата е " + s.area());
    }
}
```

Резултати:

Площта на кръга е 314,1593
Площта на сферата е 1256,637

Полиморфизъм чрез абстрактни класове

```
RoundShape shape;
shape = new Circle(5, 5, 10.0F);
Console.WriteLine("Площта на кръга е " + shape.area());
shape = new Sphere(5, 5, 10.0F);
Console.WriteLine("Площта на сферата е " + shape.area());
```

Свойства (properties)

Свойства – разглеждат се като елегантни полета от страна на потребителя, като притежават възможностите на методите за достъп.

1. Дефиниране на свойство – декларира се като поле и методи за достъп.

```
[атрибути] [модификатори] <тип> <име_на_свойство>
{
    [set
        {<тяло-на-метод-за-достъп>}
    ]
    [get
        {<тяло-на-метод-за-достъп>}
    ]
}
```

- задължително присъства един от двата метода **set** или **get**;
- **свойство за четене-запис** – дефинирани са и двата метода **set** и **get**;
- **свойство само за четене** – дефиниран е само **get**;
- **свойство само за запис** – дефиниран е само **set**;
- **свойствата не могат да се използват като параметри на методи (не са полета)**;
- **могат да се дефинират като **static**, но той не може да се комбинира с **virtual**, **abstract**, **override**, които се използват само за членове на екземплярите**;
- **осигуряват коректно разглеждане на полето.**

Пример:

```
using System;
class Address
{
    protected string city;           // Адрес
    public string City                // Населено място
    {
        get { return city; }
    }
    protected int telCode;           // Населено място
    public int TelCode                // Свойство за четене
    {
        get { return telCode; }
        set
        {
            telCode = value;
            // Обновяване на city при валиден telCode
            if (telCode == 2)
                city = "София";
        }
    }
}
```

```
class PropertyApp
{
    static void Main()
    {
        Address addr = new Address();
        addr.TelCode = 2;
        Console.WriteLine ("Населеното място с телефонен код {0} е {1}.",
            addr.TelCode, addr.City);
    }
}
```

Резултати:

Населеното място с телефонен код 2 е София.

Компиляторът създава елементи за всяко свойство:

- **get метод за достъп:** **get_City**, **get_TelCode**
- **set метод за достъп:** **set_TelCode**
- **Дефиниции на свойства (винаги):**

```
.property instance string City
{
    .get instance string Address::get_City()
} // end of property Address::City
.property instance int TelCode
{
    .get instance int Address::get_TelCode()
    .set instance void Address::set_TelCode (int)
} // end of property Address::TelCode
```

2. Наследяване и свойства

- а) предефиниране на наследени свойства** – свойството в базовия клас е с модификатор **virtual**, а в производния клас – с **override**.

Пример:

```
using System;
class Address
{
    protected string city; // Населено място
    public string City
    {
        get { return city; }
    }
    protected int telCode; // Телефонен код
    public virtual int TelCode
    {
        get { return telCode; }
        set
        { telCode = value;
          // Обновяване на city при валиден telCode
          if (telCode == 2)
              city = "София";
        }
    }
}
```

```
class FullAddress : Address // Пълен адрес
{
    private string state; // Държава
    public string State // Свойство за четене
    {
        get { return state; }
    }
    public override int TelCode // Предефинира
    // наследеното свойство
    {
        set
        {
            telCode = value;
            // Обновяване на city при валиден telCode
            if (telCode == 3592)
            {
                city = "София";
                state = "България";
            }
        }
    }
}
```

```
class PropertyApp
{
    static void Main()
    {
        FullAddress addr = new FullAddress();
        addr.TelCode = 3592;
        Console.WriteLine ("Код: {0}, населено място: {1}, държава: {2}.",
            addr.TelCode, addr.City, addr.State);
    }
}
```

Резултати:

Код: 3592, населено място: София, държава: България.

б) принудителна реализация на свойства чрез абстрактни свойства

```
abstract class <име_на_абстрактен_базов_клас>
{
    public abstract <тип> <име_на_абстрактно_свойство>
    {
        get;
        set;
    }
}
```

Пример:

```
using System;
abstract class Employee // Абстрактен Служител
{
    protected int id; // Номер
    public int Id
    { get { return id; } }
    protected int hoursWorked; // Изработени часове
    protected double hourlyCost; // Часова ставка
    public abstract double HourlyCost
    {
        get;
    }
    protected Employee (int id, int hoursWorked)
    { this.id = id;
      this.hoursWorked = hoursWorked;
    }
    public override string ToString()
    { return "Служител с номер = " + id +
      " изработка " + HourlyCost + " лв на час.";
    }
}
```

```
class ContractEmployee : Employee // Служител с договор
{
    protected double hourlyWage; // Заплащане на час
    public override double HourlyCost // Реализация на
    // часова ставка
    {
        get
        {
            return hourlyWage;
        }
    }
    public ContractEmployee (int id, int hoursWorked, double hourlyWage)
        : base(id, hoursWorked)
    {
        this.hourlyWage = hourlyWage;
    }
}
```

```
class SalariedEmployee : Employee // Служител със заплатата
{
    protected double salary; // Заплата
    public override double HourlyCost // Реализация на
    { // часова ставка
        get
        {
            return salary / hoursWorked;
        }
    }
    public SalariedEmployee(int id,int hoursWorked,double salary)
        : base(id,hoursWorked)
    {
        this.salary = salary;
    }
}
```

```
class OverrideProperties
{
    static void Main()
    {
        Employee e;
        e = new ContractEmployee(1, 40, 20);
        Console.WriteLine(e);
        e = new SalariedEmployee(2, 160, 400);
        Console.WriteLine(e);
    }
}
```

Резултати:
 Служител с номер = 1 изработва 20 лв на час.
 Служител с номер = 2 изработва 2,5 лв на час.

3. Използване на свойства

а) осигуряват високо ниво на абстракция – потребителят не се интересува дали съществува метод за достъп до даден член;

б) осигуряват родов смисъл на достъпа до членове на класа чрез стандартния синтаксис:

<име_на_екземпляр>.<име_на_поле>

в) гарантират допълнителна обработка на дадено поле, когато то е модифицирано или достъпно.

4. Автоматично реализирани свойства

```
using System
public class PhoneEntry
{
    public String Name { get; set; }
    public long Phone { get; set; }
    public static void Main()
    {
        PhoneEntry p = new PhoneEntry ();
        p.Name = "Maria";
        p.Phone = 359888881000;
        Console.WriteLine(p.Name + " " + p.Phone);
    }
}
```

Компиляторът автоматично генерира поле от тип String (според типа на свойството Name), поле от тип long (според типа на свойството Phone) и методите get_Name, set_Name, get_Phone и set_Phone, които връщат/установяват стойностите на полетата.

```
using System;
class Rational
{
    public int Nominator { get; set; }
    public int Denominator { get; set; }
    public Rational(int nominator, int denominator)
    {
        Nominator = nominator;
        Denominator = denominator;
    }
    public Rational() {}
    public override string ToString()
    { return Nominator + "/" + Denominator; }
}
class Program
{
    static void Main(string[] args)
    {
        Rational r = new Rational(1, 5);
        float number = (float)r.Nominator / (float)r.Denominator;
        Console.WriteLine(r + " = " + number);
    }
}
```

1/5 = 0.2

5. Инициализатори на обекти

Rational r = new Rational() { Nominator=1, Denominator=5 };
което е идентично на:
 Rational r = new Rational();
 r.Nominator=1;
 r.Denominator=5;

Можем да пропуснем скобите:
 Rational r = new Rational { Nominator=1, Denominator=5 };

С един оператор можем да конструираме обект от тип Rational, да извикаме неговия конструктор, да инициализираме двете му публични свойства и да извикаме методите ToString и ToUpper за резултантния израз – кодиране в контекст на израз.

String s=new Rational{Nominator=1, Denominator=5}.ToString().ToUpper();

6. Анонимни типове

```
var o = new { свойство1 = израз1, ..., свойствоN = изразN }
```

Компиляторът прави заключение за типа на всеки израз, създава: **private** полета за тези типове; **public** свойства само за четене за всяко от тези полета; конструктор, който приема всички тези изрази; и предефинира методите: **Equals**, **GetHashCode** и **ToString**.

```
var r1 = new { Nominator=1, Denominator=5 };
Console.WriteLine(r1); // {Nominator=1, Denominator=5 }
```

Компиляторът може да генерира анонимен тип, като прави заключение за имената на свойствата и техните типове от променливите:

```
int Nominator=1;
int Denominator=5;
var r2 = new {Nominator, Denominator};
Console.WriteLine(r2); // {Nominator=1, Denominator=5 }
Console.WriteLine(r1.Equals(r2)); // True
```

Предефиниране на оператори

Предефиниране на оператори

- предефинират се съществуващи оператори с операнди (единият или и двата) от тип **class** или **struct**;
- друго средство за извикване на метод;
- добавя абстракция – един от най-важните аспекти на обектно-ориентираното програмиране.

Дефиниране на предефиниран оператор

op – чрез operator

```
public static <тип_на_върната_стойност> operator op
    (<аргумент1>[,<аргумент2>])
{
    <тяло-на-предефиниран-оператор>
    return върната_стойност;
}
```

- трябва да бъде **public** и **static**;
- **<тип_на_върната_стойност>** – произволен тип (най-често е от тип **class** или **struct**);
- за унарнен оператор – **<аргумент1>** от тип **class** или **struct**;
- за бинарен оператор – **<аргумент1>** от тип **class** или **struct**, **<аргумент2>** – от произволен тип.

Правила и ограничения

1. Две категории оператори за предефиниране:

- унарни + - ~ ++ -- true false ;
- бинарни + - * / % & | ^ << >> == != > < >= <= .

2. Не се предефинират операторите:

- , [] () && || ? ;
- недефинирани оператори в C# (напр. **);
- дефинирани при изпълнение:
 - достъп до член (.);
 - извикване на метод;
 - присвояване (=);
 - new.

3. Предефинират се по двойки (предефинира ли се единият, трябва да се предефинира и другият):

- == и != (трябва да се предефинират методите **Equals** и **GetHashCode**);
- < и > .

4. Операторът за присвояване не се предефинира, но ако предефинираме бинарен оператор, неговият съставен еквивалент за присвояване се предефинира неявно – напр. предефинирането на + води до неявно предефиниране на +=.

Пример: Предефиниране на оператора + за класа РАЦИОНАЛНО ЧИСЛО

```
using System;
class Rational // Рационално число
{
    private int numerator; // Числител
    private int denominator; // Знаменател
    public Rational (int numer, int denom)
    {
        numerator = numer;
        denominator = denom;
        Reduce();
    }
    private void Reduce() // Привежда под общ знаменател
    {
        int common = Gcd (Math.Abs(numerator), denominator);
        numerator /= common;
        denominator /= common;
    }
}
```



```
private int Gcd (int n1, int n2)           // НОД
{
    while (n1 != n2)
        if (n1 > n2)
            n1 -= n2;
        else
            n2 -= n1;
    return n1;
}
// Предефинира оператор +
public static Rational operator+ (Rational op1, Rational op2)
{
    int commonDenominator = op1.denominator*op2.denominator;
    int sum = op1.numerator*op2.denominator +
        op1.denominator*op2.numerator;
    return new Rational (sum, commonDenominator);
}
public override string ToString()
{
    return numerator + "/" + denominator;
}
```

```
// Предефинира принудително преобразуване от Rational във float
public static explicit operator float(Rational op)
{
    return (float)op.numerator / op.denominator;
}
// Предефинира подразбиращо се преобразуване от Rational в double
public static implicit operator double(Rational op)
{
    return (double)op.numerator / op.denominator;
}
```

```
class TestRational
{
    static void Main()
    {
        Rational x, y, z;
        x = new Rational (1, 4);
        y = new Rational (1, 3);
        z = x + y;
        Console.WriteLine (x + "+" + y + "=" + z);
        Console.WriteLine (z);
        z += y;
        Console.WriteLine ("+=" + y + "=" + z);
        float f = (float)z;
        Console.WriteLine(f);
        double r = z;
        Console.WriteLine(r);
    }
}
```

Резултати:

```
1/4+1/3=7/12
7/12+=1/3=11/12
0.58333333
0.5833333333333333
```