Програмите на C# и система от типове в .NET модела.

Съдържание на лекцията

1. **Програмите на C# - елементарен пример**
2. **Система от типове в .NET модела.**
3. **Идентификатори, декларации, операции, изрази, идентификатори, конструкции, коментари в C#**
4. **Вход и изход от конзолата**
5. **Структури, класове, интерфейси, делегати - кратък преглед**
6. **Методите на System.Object**

Програмите на C#

- Представляват съвкупност от дефиниции на класове, структури и други типове
- Някой от класовете съдържа метод **`Main()`** – входна точка за програмата
- Физически, типовете са разположени във файлове (ансамбли). C# приложенията могат да се състоят от много файлове.
- В един файл може да има няколко класове, структури и други типове.
- Класовете логически се разполагат в пространства от имена (namespaces). Един namespace може да е разположен в няколко файла.
- Пълното име на типа (напр. тип FileStream) се образува от пътя до него в йерархията от namespace-и и собственото му име, например

  System.IO.FileStream fs = new System.IO.FileStream(...);
- Можем да работим с кратките имена на типа

  using System.IO;

  FileStream fs = new FileStream(...);

Елементарна програма на C#

```
using System;
                                                    /*Директивата using System;
                                                    като #include в C++
                                                    като import в Java
                                                    като uses в Delphi */
public class proba
                                                    /*Декларацията на клас
                                                    * ключова дума class */
{static void Main()
                                                    /*Методът static void Main()
                                                    - входна точка на програмата
                                                    - когато завърши, завършва и програмата */
   {
      Console.WriteLine("Hello Students!!!");
      Console.ReadLine();
                                                    /* Класът System.Console
                                                    - осигурява средства за вход и изход от конзолата  */
   }
}
```

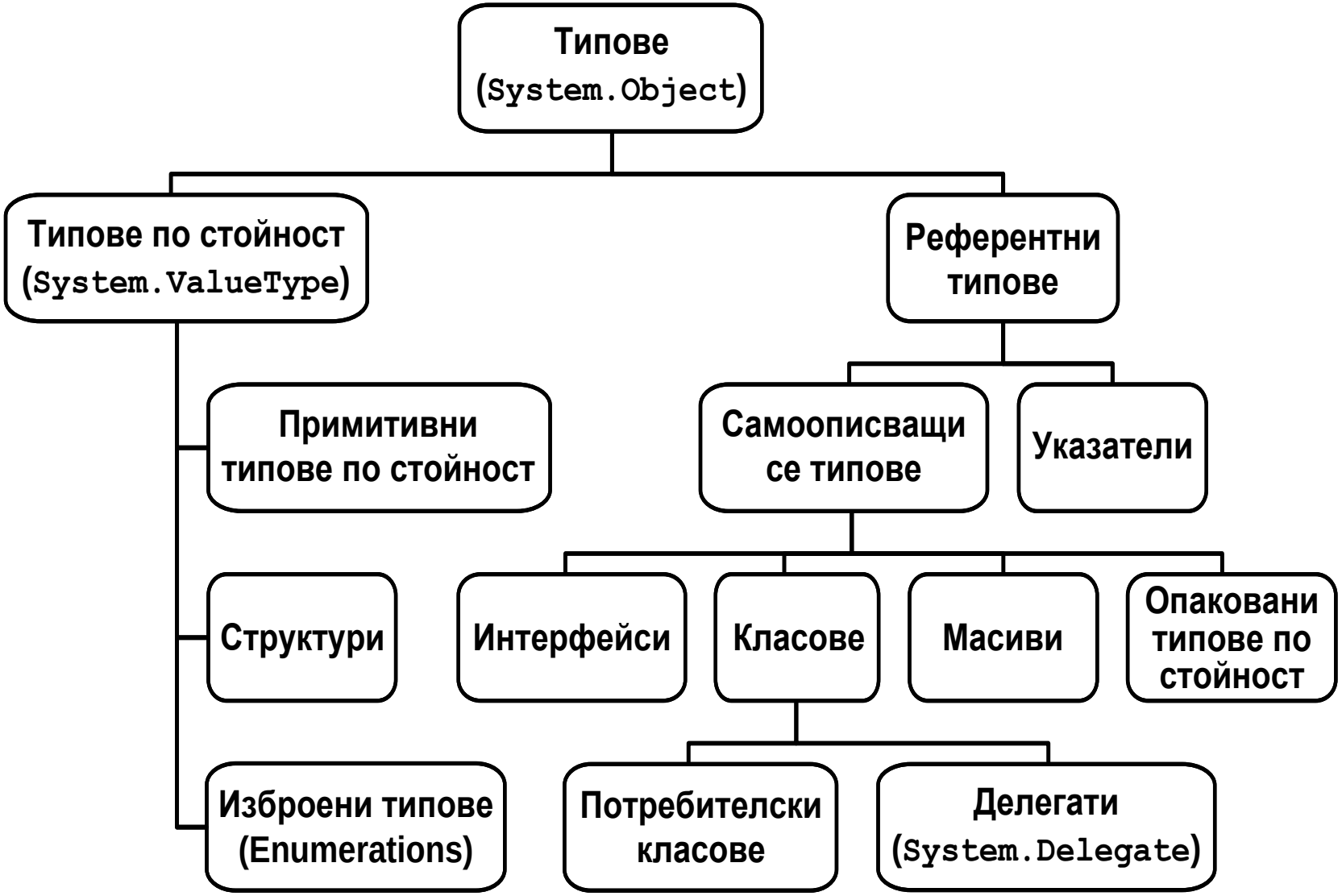
Как работи програмата?

- Директивата `using System;`
  - като `#include` в C++
  - като `import` в Java
  - като `uses` в Delphi
- Декларацията на клас
  - ключова дума `class`
- Методът `static void Main()`
  - входна точка на програмата
  - когато завърши, завършва и програмата
- Класът `System.Console`
  - осигурява средства за вход и изход от конзолата

Типове данни в C#

- Типовете данни биват два вида
  - Типове по стойност (value types)
  - Типове по референция (reference types)
- Типовете биват още
  - примитивни (вградени, built-in) типове
  - Типове, дефинирани от потребителя

Типове данни в C# (класическа фигура)



Типове данни в C#

- Типове по стойност (value types)
  - Произлизат от System.ValueType, който от своя страна наследява System.Object.
  - Заделят необходимата за съхраняване на стойността си памет в стека в момента на декларирането и директно съдържат стойността, която им се присвоява.
  - Не могат да приемат стойност null (защото не са адреси)
  - Когато биват предавани като параметри на метод – те се предават по стойност (извиканият метод работи с копието на стойността на променливата).
  - Стойностни са вградените примитивни типове: sbyte, byte, short, ushort, int, long, ulong, float, double, decimal, char и bool (описани в таблицата на следващи слайдове) както и дефинираните от потребителя типове структури и изброяване.

Типове данни в C#

- Типове по референция (reference types)
  - Директно наследяват System.Object.
  - Тези типове заделят необходимата за съхраняване на стойността си памет динамично (в heap-а), по време на изпълнение на програмата, чрез използване на new. При това: в стека се съхраняват менажирани указатели (референции) към динамичната памет (heap), където се съхраняват техните стойности.
  - Когато биват предавани като параметри на метод - референтните типове се предават по референция (адрес) и се унищожават от garbage collector автоматично, по някое време, когато не се използват повече от програмата.
  - Вижда се, че референтните типове в C# са аналогични на указателите в C++, но се различават по начина по който работят , а именно: тук са менажирани указатели към стойност в динамичната памет (heap) и се унищожават от garbage collector автоматично.
  - За разлика от стойностните, референтните типове, могат да получават стойност null, което означава липса на стойност.

Типове данни в C#

- Типове по референция (reference types) са:
  - вградените примитивни типове object и string (виж таблица 3.1), опакованите стойностни типове и дефинираните от потребителя типове:
  - Класове:

```
class Foo: Bar, IFoo {...}
```

  - Интерфейси

```
interface IFoo: IBar {...}
```

  - Масиви

```
string[] a = new string[5];
```

  - Делегати

```
delegate void Empty();
```

Пример - Reference & Value type

```
using System;
    class MyClass { public int mValue; }  // Reference type
    struct MyStruct { public int mValue; }  // Value type
    class TestTypes
    {     static void Main()
          {          MyClass class1 = new MyClass();
                     class1.mValue = 10;
                     Console.WriteLine(class1.mValue); // Отпечатва 10
                     MyClass class2 = class1;
                     class2.mValue = 20;
                     Console.WriteLine(class1.mValue); // Отпечатва 20
                     MyStruct struct1 = new MyStruct();
                     struct1.mValue = 10;
                     Console.WriteLine(struct1.mValue); // Отпечатва 10
                     MyStruct struct2 = struct1;
                     struct2.mValue = 20;
                     Console.WriteLine(struct1.mValue); // Отпечатва 10
          }
    }
```

Типове данни в C#

- На всички типове в . Net езиците съответстват типове от CTS на .NET Framework, например на:
- int в C# -> System.Int32
- integer в VB.Net -> System.Int32

Вградени Примитивни типове

Тип	Размер в байт	Тип .NET	Описание
			Базов тип
object		System.Object	Съхранява всеки тип, т.е явява се общ родител (null – по подразбиране); (реф.тип)
			Логически тип
bool	1	System.Bodean	true или false; (false – по подразбиране); (стойн. тип)
			Цели типове
sbyte	1	System.SByte	Цял със знак (от -128 до 127); (0 – по подразбиране); (стойн. тип)
byte	1	System.Byte	Цял без знак (от 0 до 255); (0 – по подразбиране); (стойн. тип)
short	2	System.Int16	Цял със знак (от -32768 до 32767); (0 – по подразбиране); (стойн. тип)
ushort	2	System.UInt16	Цял без знак (от 0 до 65535); (0 – по подразбиране); (стойн. тип)
int	4	System.Int32	Цял със знак (от -2147483648 до 2147483647); (0 – по подразбиране); (стойн. тип)
uint	4	System.UInt	Цял без знак (от 0 до 4 294 967 295); (0u – по подразбиране); (стойн. тип)
long	8	System.Int64	Цял със знак (от -9223372036854775808 до 9223372036854775807); (0L – по подразбиране); (стойн. тип)
ulong	8	System.UInt64	Цял без знак (от 0 до 18446744073709551615); (0u – по подразбиране); (стойн. тип)

Вградени Примитивни типове

Тип	Размер в байт	Тип .NET	Описание
			Типове за Реални числа
float	4	System.Single	Число с плаваща точка и двойна точност. Съдържа стойности в приблизителния диапазон от -1.5×10^{-45} до $+3.4\times10^{38}$ при 7 значещи цифри (след дес.точка); Суфикс f, F; Пример: float f1=45.5f; (0.0f – по подразбиране); (стойн. тип)
double	8	System.Double	Число с плаваща точка и двойна точност. Съдържа стойности в приблизителния диапазон от -5.0×10^{-314} до $+1.7\times10^{308}$ при 15-16 значещи цифри (след дес.точка); (0.0 – по подразбиране); Суфикс – няма; (стойн. тип)
decimal	16	System.Decimal	Число, с фиксирано положение на десетичната точка (до 28 значещи цифри): от -1.0×10^{-28} до $+7.9\times10^{28}$. Обикновено се използва във финансови отчети. Изисква суфикс m, M; Пример: decimal d2 = 45.5m; (0.0m – по подразбиране); (стойн. тип)
			Символен тип
char	2	System.Char	Unicode символ; Диапазон: От '\u0000' до '\uffff'; ('\u0000' – по подразбиране); (стойн. тип)
			Стринг
string		System.String	Поредица от Unicode-символи; (null – по подразбиране); (реф.тип)

Преобразуване на типовете (Conversions)

- Два типа преобразувания на примитивните типове
  - По подразбиране (**implicit conversion**)
    - позволява се, когато е безопасно
    - Например: int → long, float → double, long → float, byte → short

    Примери:

```
float pi=3.1415f;
double PI = pi;    // implicit conversion
byte b = 255;
short sh = b;      // implicit conversion
```

Преобразуване на типовете (**conversion**)

- Изрично (**explicit conversion**)
  - когато преобразуваме към по-малък тип или типовете не са директно съвместими
  - Например **long → int, double → float, char → short, int → char, sbyte → uint**
- В C# има специална ключова дума “checked”, която се използва за да се получи System.OverflowException при препълване, вместо грешен резултат.
- Може да се използва и Convert клас за изрично преобразуване:

```
float PI = Convert.ToSingle(pi);
```

- Ключовата дума **unchecked** действа в противоположна посока

Преобразуване на типовете

```
...short sh = i;
//short – 2 байта, byte - 1 байт
checked {
byte b = (byte) sh;
/* explicit conversion
при препълване се хвърля изключение System.OverflowException,
вместо грешен резултат.*/
          }
unchecked
          {
uint k = 1234567890; // uint - 4 байта
sbyte sb = (sbyte) k; //sbyte - 1 байт
/* explicit conversion
Възможно е да се получи грешен резултат,
при препълване не се хвърля изключение */
          }
```

Преобразуване на стойностни типове към референтни - boxing и обратно – unboxing

Променливи от стойностен тип могат да се съхраняват в променливи от референтен тип чрез опаковане (boxing):

```
float pi = 3.1415f; //тип стойност
object b = pi;      // pi се опакова в b
Console.WriteLine("pi={0}, b={1}",pi,b);
                                  //pi=3.1415, b=3.1415
```

Процедура при опаковане:

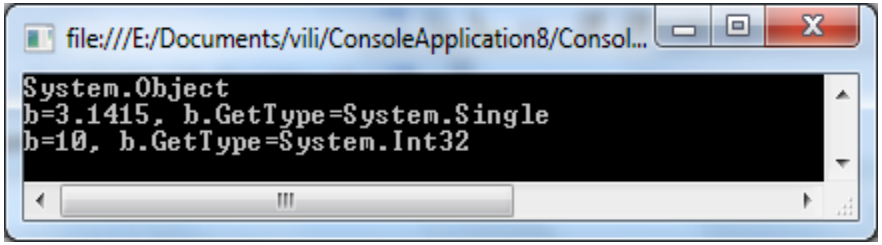
– Заделя се динамична памет за създаване на обект

– Копира се съдържанието на променливата от стека в заделената динамична памет

– В стека се връща референция към създадения в динамичната памет обект.

Разопаковането (unboxing) изисква изрично преобразуване на обекта към стойностния тип:

```
float pi = 3.1415f;  //тип стойност
object b = pi;       // a се опакова в b
float c = (float)b;  //b се разопакова в c
Console.WriteLine("pi={0}, b={1}, c={2}",pi,b,c);
                                  //pi=3.1415, b=3.1415, c=3.1415
```




```
…
static void Main()
    {
float pi = 3.1415f;            //тип стойност
object b=new object();    //референтен тип-object
Console.WriteLine("{0}", b.GetType()); //System.Object
b=pi;                                  // pi се опакова в b
Console.WriteLine("b={0}, b.GetType={1}", b, b.GetType());
    //b=3.1415,… System.Single
int i = 10; b = i; // i се опакова в b
Console.WriteLine("b={0}, b.GetType={1}", b, b.GetType());
    //b=10,… System.Int32
}
```

Преобразуване (Casting)

- Casting Up (Implicit Cast): **от породен Class към базов Class**
- Casting Down (Explicit Cast): **базов Class към породен Class**

```
public class Animal {…   } …
class Dog : Animal {…  } …
static void Main()
 {
Dog myDog = new Dog(); //обект myDog от тип Dog
Animal anim = new Animal();//обект anim от тип Animal
anim = myDog;          // Casting Up (Implicit Cast)
Dog newDog=(Dog)anim;
                       // Casting Down(Explicit Cast)
  }
```

Оператори за работа с типове в C#: is, as, typeof

- **Оператор is** – проверява дали даден обект е инстанция на даден тип

```
...
class Base { } ...
Object objBase = new Base();
//сега проверяваме дали objBase е обект от тип Base
if (objBase is Base)
Console.WriteLine("objBase is Base");
// objBase is Base
```

Оператори за работа с типове в C#: is, as, typeof

- **Оператор as – преобразува даден референтен тип (в примера var2 от тип object) в друг референтен тип (в примера – в string), като при неуспех не предизвиква изключение, а връща стойност null**

```
/* Създаваме string и го присвояваме на променлива var2 от
   тип object */
      string var1 = "carrot";
      object var2 = var1;
 // Опитваме се да преобразуваме var2 към string.
      string var3 = var2 as string; //връща null при неуспех!
      if (var3 != null)
      {
         Console.WriteLine("have string variable");
      } // have string variable
```

Оператори за работа с типове в C#: is, as, typeof

- **typeof(тип)** - използва се за получаване на обект от тип **системния тип** за даден тип.

Пример:

```
System.Type mytype = typeof(int);
Console.WriteLine("{0}", mytype.FullName);
                                    //System.Int32
if (mytype.IsPrimitive)
                Console.WriteLine("Is a primitive.");
if (mytype.IsPointer)
                Console.WriteLine("Is pointer.");
                                    //Is a primitive
```

Типове дефинирани от потребителя: изброими типове (enumerations)

- Изброимият тип в C# е **стойностен** тип (наследява System.ValueType).
- Състои се от множество именовани константи и се дефинира се със запазената дума **enum**.

Пример: Дефиниране на изброим тип Season:

```
public enum Season { Winter, Spring, Summer, Fall };
```

Дефиниране и инициализация на променлива от тип Season:

```
Season now = Season.Spring;
Console.WriteLine("now is {0}, ",now); //now is Spring
```

Типове дефинирани от потребителя: изброени типове (enumerations)

- Както се вижда, променливите от изброен тип могат да приемат една от дефинираните в **enum** стойности.
- Дефиницията, която представихме може да се даде и така:

```
public enum Season: int { Winter, Spring, Summer, Fall };
```

- Вътрешно изброимият тип се представя с **int** (като номерацията започва от 0), но за базов може да бъде избран и друг тип, например:

```
public enum Season:
 ushort { Winter, Spring, Summer, Fall };
```

Пример:

```
using System;
class Test_Enum
{    enum gradus : byte {    min = 0,krit = 72,max = 100 }
     static void Main()
     {    Console.WriteLine("minimal temperature={0}",gradus.min);
          Console.WriteLine("critic temperature={0}",gradus.krit);
          Console.WriteLine("maxima temperature={0}",gradus.max);
          Console.WriteLine("--------------------------");
          Console.WriteLine("minimal
temperature={0}",(byte)gradus.min);
          Console.WriteLine("critic
temperature={0}",(byte)gradus.krit);
          Console.WriteLine("maxima
temperature={0}",(byte)gradus.max);
     }
}
```

Результат:

```
minimal temperature=min
critic temperature=krit
maxima temperature=max
--------------------------
minimal temperature=0
critic temperature=72
maxima temperature=100
```

Запазени думи в C#

abstract	as	base	bool	break	byte
case	catch	char	checked	class	const
continue	decimal	default	delegate	do	double
else	enum	event	explicit	extern	false
finally	fixed	float	for	foreach	goto
if	implicit	in	int	interface	internal
is	lock	long	namespace	new	null
object	operator	out	override	params	private
protected	public	readonly	ref	return	sbyte
sealed	short	sizeof	stackalloc	static	string
struct	switch	this	throw	true	try
typeof	uint	ulong	unchecked	unsafe	ushort
using	virtual	void	volatile	while	

Идентификатори в C#

- Състоят се от букви, цифри и знак за подчертаване.
- Винаги започват с буква или подчертаване.
- Малките и главните букви се различават

- Microsoft препоръчва се следната конвенция
  - **PascalCase** – имена на класове, namespace-и, структури, типове, методи, свойства, константи
  - **camelCase** – имена на променливи и параметри

Декларации

- Декларациите на променливи в C# могат са няколко вида (почти като в C++, Java и Delphi):
  - локални променливи (за даден блок)
    - `int count;`
  - член-променливи на типа
    - `public static int mCounter;`
    - могат да имат модификатори
- Константите:
    - compile-time константи – декларират се като променливи със запазената дума `const`
    - runtime константи – декларират се като полета на класовете, с модификатор `readonly`

Оператори

- **Много близки до операторите в C++ и Java, със същите приоритети**
- **Аритметични: `+`, `-`, `*`, `/`, `%`, `++`, `--`**
- **Логически: `&&`, `||`, `!`, `^`, `true`, `false`**
- **Побитови операции: `&`, `|`, `^`, `~`, `<<`, `>>`**
- **За слепване на символни низове: `+`**
- **За сравнение: `==`, `!=`, `<`, `>`, `<=`, `>=`**
- **За присвояване: `=`, `+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `|=`, `^=`, `<<=`, `>>=`**
- **За работа с типове: `as`, `is`, `sizeof`, `typeof`**
- **Други: `.`, `[]`, `()`, `?:`, `new`, `checked`, `unchecked`, `unsafe`**

Изрази (expressions)

- Изразите, които се изчисляват до някаква стойност се наричат expressions
- Имат синтаксиса на C++ и Java
- Например:

```
a = b = c = 20; // израз със стойност 20
(a+5)*(32-a)%b  // израз с числова стойност
"Иван" + " Иванов" // символен израз (string)
Math.Cos(Math.PI/x) // израз с реална стойност
typeof(obj) // израз от тип System.Type
(int) arr[idx1][idx2] // израз от тип int
new Student() // израз от тип Student
(currentValue <= MAX_VALUE) // булев израз
```

Конструкции (statements)

- **Програмните конструкции (statements) имат синтаксиса на C++ и Java**
- **Биват няколко вида:**
  - **Елементарни програмни конструкции**
  - **Съставни конструкции – { … }**
  - **Програмни конструкции за управление**
    - **Условни конструкции**
    - **Конструкции за повторение и цикъл**
    - **Конструкции за преход**
    - **За управление на изключенията**
  - **Специални конструкции (напр. `checked`)**

Конструкции (statements)

- Елементарни програмни конструкции
  - Най-простите елементи на програмата. Примери:

```
sum = (a+b)/2; // присвояване (<променлива> = <израз>)
PrintReport(report); // извикване на метод
student = new Student("Ivan Ivanov", 123456);
// създаване на обект
```

- Съставни конструкции
  - Състоят се от няколко други конструкции, оградени в блок. Например:

```
{
   Report report = GenerateReport(period);
   report.Print();
}
```

Конструкции за управление

- Условни конструкции (conditional statements)
  - `if` и `if-else`

```
if (orderItem.Ammount > ammountInStock)
{
    MessageBox.Show("Not in stock!", "error");
}

if (Valid(order))
{
    ProcessOrder(order);
}
else
{
    MessageBox.Show("Invalid order!", "error");
}
```

Конструкции за управление

- Условни конструкции (conditional statements)
  - `switch` и `case`
  - позволено е използване на `string` и `enum`

```
switch (characterCase)
{
    case CharacterCasing.Lower:
        text = text.ToLower();
        break;
    case CharacterCasing.Upper:
        text = text.ToUpper();
        break;
    default:
        MessageBox.Show("Invalid case!", "error");
        break;
}
```

Конструкции за управление

- Конструкции за повторение (iteration statements)
  - for-цикъл

```
// Отпечатваме числата от 1 до 100 и техните квадрати
for (int i=1; i<=100; i++)
{
    int i2 = i*i;
    Console.WriteLine(i + " * " + i + " = " + i2);
}
```

  - while-цикъл

```
// Изчисляваме result = a^b
result = 1;
while (b > 0)
{
    result = result * a;
    b--;
}
```

Конструкции за управление

- Конструкции за повторение (iteration statements)
  - цикъл `do-while`

```
// Четем символи до достигане на край на ред
do
{
    ch = ReadNextCharacter(stream);
}
while (ch != '\n');
```

  - цикъл `foreach` – за обработка на колекции

```
string[] names = GetNames();

// Отпечатваме всички елементи на масива
foreach (string name in names)
{
    Console.WriteLine(name);
}
```

Конструкции за преход

- Конструкции за преход
  - `break`, `continue` – използват се в цикли
  - `goto` – безусловен преход
  - `return` – за връщане от метод

```
using System;
class Break_Continue_Test
{static void Main()
      {for (int i = 1; i <= 100; i++)
            { if (i == 5) break;
            else
            if    (i == 3) continue;
              Console.WriteLine(i); //1 2 4
            }
      }
}
```

Други конструкции

- Конструкции за управление на изключенията
  - `throw` – предизвикване на изключение
  - `try-catch` – прихващане на изключение
  - `try-finally` – сигурно изпълнение на завършваща секция
  - `try-catch-finally` – прихващане на изключение със завършваща секция
- Специални конструкции
  - `lock` – синхронизирано изпълнение
  - `checked`, `unchecked` – контрол на аритметичните препълвания
  - `fixed` – фиксиране на местоположението в паметта при работа с неуправляван код

Коментари в програмата

- Коментарите биват няколко вида
  - Коментар за част от програмен ред

```
// Съдържа всички поръчки на потребителя
```

  - Блоков коментар

```
/* Изтриваме всички поръчки, за които някой артикул не
е наличен в необходимото количество. Изтриване реално
не се извършва, а само се променя статуса на Deleted */
```

XML документация

- **Представлява съвкупност от коментари, започващи с `///`**
- **Значително улеснява поддръжката – документацията е част от кода, а не стои във външен файл**

- **Може да съдържа различни XML тагове:**
  - **`<summary>`…`</summary>` – кратко описание за какво се отнася даден тип, метод, свойство и т.н.**
  - **`<remarks>`…`</remarks>` – подробно описание на даден тип, метод, свойство и т.н.**
  - **и др.**

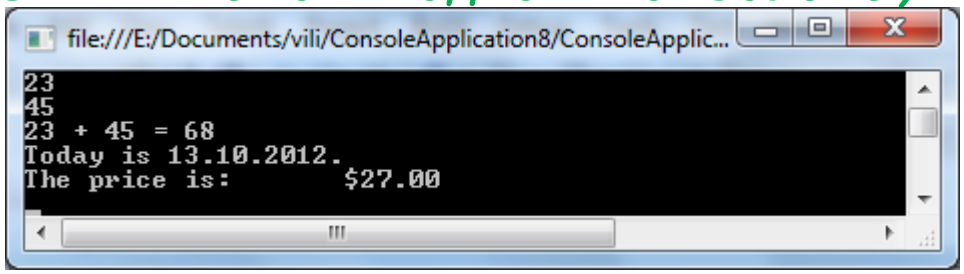
- **C# компилаторът може да извлича XML документацията, като XML файл за по-нататъшна обработка**

Вход и изход от конзолата

- Използват се стандартни класове от BCL
- Вход от конзолата
  - `Console.ReadLine()` – чете цял символен ред и връща `string`
  - `Console.Read()` – чете единичен символ и връща `char`
- Изход към конзолата
  - `Console.Write(…)` – печата на конзолата подадените като параметри данни (приема `string`, `int`, `float`, `double`, …)
    - приема параметрични форматиращи низове
  - `Console.WriteLine(…)` – работи като `Conlsole.Write()`, но преминава на нов ред

Вход и изход от конзолата – пример

```
int a = Int32.Parse(Console.ReadLine());
int b = Int32.Parse(Console.ReadLine());
Console.WriteLine("{0} + {1} = {2}", a, b, a+b);
// (въвеждаме съответно 3 и 2 като вход от конзолата)
// 2 + 3 = 5
```




```
Console.WriteLine(
    "Today is {0:dd.MM.yyyy}.", DateTime.Now);
// Today is ...13.9.2019.

Console.WriteLine("The price is: {0,12:C}", 27);
// The price is:     27,00 лв
// (точният формат зависи от текущите езикови
настройки)
//12 – извеждане в 12 позиции, с дясно подравняване
//-12 – извеждане в 12 позиции, с ляво подравняване
```

Структури

- Структурата е **стойностен тип** (наследява System.ValueType).
- Дефинира се с ключовата дума **struct**, но описанието на структура е подобно на описанието на един клас.
- Може да има методи, свойства и полета
- Може да имплементира интерфейси
- Динамичната памет е с голям обем, но е скъп ресурс:
  - по бавен достъп и сложен механизъм за освобождаване на памет.
- Статичната памет е с бърз достъп, но е ограничена като обем.
- Използвайте структура, когато трябва да дефинирате тип, който:
  - има малък брой методи и няма такива, които променят член данните;
  - не се изисква да наследява друг тип и типът няма да бъде наследяван от други типове;
  - обектите са ще са малки (до 16 байта).

```
using System;
namespace Test
{  struct President
   {     public string FirstName;
         public string LastName;
         public string TwoNames()
         {return FirstName + " " + LastName;     }
   }
   class Class1
   {     static void Main()
         {        President p; //President p=new President();
                  p.FirstName="Ivan";
                  p.LastName="Ivanow";
    Console.WriteLine("{0}, {1}, {2} {3}",
   p.FirstName,p.LastName,p.TwoNames(),p.ToString());
         }
   }
}   //Ivan, Ivanov, Ivan Ivanov, Test.President
```

Класове

- Класът е референтен тип и в C# се дефинира се с ключовата дума class.
- Класовете в .Net са класическа реализация на структурата от данни „клас” от обектно-ориентираното програмиране и са много подобни на класовете в C++ и Java.
- Класовете имат видимост (public или internal – по подразбиране) и членове (class members):
  - полета,
  - константи,
  - методи,
  - свойства,
  - индексатори,
  - събития,
  - оператори,
  - конструктори и др.,

  които могат да бъдат статични (общи за типа) или за дадена инстанция (екземплярни) и с различна видимост:

Видимост на членовете

- public – членът е достъпен за всички други класове във всички ансамбли (default за enum, interface) на дадено приложение
- protected - членът е достъпен само за класовете, които наследяват този клас (наследниците могат да са и извън ансамбъла)
- internal - членът е достъпен само за класовете, в ансамбъла
- private - членът е достъпен само за този клас (default за class, struct)
- protected - internal - членът е достъпен само за класовете, които наследяват този клас в дадения ансамбъл

Пример:

```
public class Point
 {  public int x, y;
    public Point(int p1, int p2)
    {  x = p1;  y = p2;  }
 }
```

Интерфейси

- Интерфейсът в C# е референтен тип, който се дефинира с ключовата дума interface.
- Интерфейсите имат име и описания на методи и свойства, но **нямат имплементиране**.
- Всеки клас, наследяващ интерфейс трябва да имплементира всички методи, свойства и събития на интерфейса.
- Чрез интерфейса всъщност се осигуряват връзки между многото класове, които го имплементират – **получава се ефект на множествено наследяване**.
- Използвайте Casting в Client Code за да използвате даден интерфейс ...вж. примера

ефект на множествено наследяване

```
interface IFoo
{void DoSomething1();}

interface IBar
{void DoSomething2();}

class MyObject : IFoo, IBar
  {...}
```

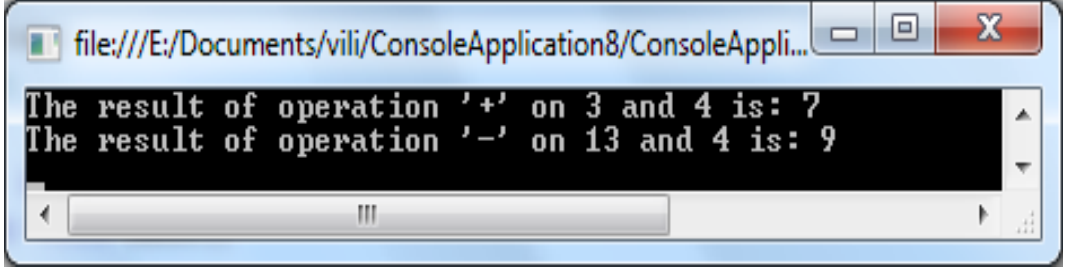
Пример: Дефинирани са два интерфейса FirstName и LastName, които се имплементират от клас Name.

```
using System;
namespace Test_Interface
{     interface FirstName
      { string GetName();}
      interface LastName
      { string GetName();}
      class Name : FirstName, LastName
      {     string FirstName.GetName()
            { return "Tom"; }
            string LastName.GetName()
            { return "Tailor"; }
      }
      class Class1
      {           public static void Main()
            {     Name my = new Name();
                  // Грешка ще бъде записа: my.GetName();
                  FirstName first_name = (FirstName)my;
                  LastName second_name = (LastName)my;
Console.WriteLine("{0} {1}", first_name.GetName(),
second_name.GetName());
                  //Tom Tailor
                  Console.ReadLine();
            }
      }
}
```

Делегати

- Делегатите в C# са типове, които пазят референция към метод.
- Дефинират с ключовата дума delegate и имат много широка сфера на приложение, в това число и за реализация на събитийния модел.
- Когато един делегат се асоциира с даден метод (например метод Add()) той се държи точно като него.
- Всеки метод, който има същата сигнатура (например метод Substra()) може да се присвои на делегата.
- При използване на делегат за указване на статичен метод – делегатът се прикрепя само към указвания метод (както в примера по долу), а при използване на делегат за указване на не-статичен метод – делегатът се прикрепя към екземпляр на класа и самия метод.

Пример:




```
using System;
class Test
{ delegate int MyDelegate(int p, int q);
    static void Main()
    {
MyDelegate arithMethod = new
MyDelegate(Add);
int r1 = arithMethod(3, 4);
Console.WriteLine("The result of
operation '+' on 3 and 4 is: {0}",r1);
int r2 = Substra(13, 4);
Console.WriteLine("The result of
operation '-' on 13 and 4 is: {0}", r2);
Console.ReadLine();
    }
    static int Add(int a, int b)
    {       return a + b;   }
    static int Substra(int a, int b)
    {      return a -b; }
}
```

Методите на System.Object

- Основна концепция на Common Type System в .Net е, че всичко е обект и всички типове наследяват директно или индиректно System.Object.
- Типът System.Object дефинира за своите наследници обща функционалност, реализирана чрез няколко метода, част от които са виртуални и следователно - могат да бъдат препокривани:

Методите на System.Object

- **virtual bool Equals (object obj)** – виртуален метод, който сравнява текущия обект с обекта **obj**, предаден в качестото на аргумент. Връща true ако текущия обект е еднакъв с obj, иначе - false. Може да се предефинира.

Методът има и статична версия:

- **static bool Equals(object objA,object objB)**, която сравнява двата обекта objA и objB, подадени като параметри.
- Реализация по подразбиране:
  - За **референтни типове** - сравняване за **идентичност** (сравняване по референция), извиква се на практика ReferenceEquals().
  - За **стойностни типове** - сравнява по **стойност**.
- Два обекта могат да бъдат проверени за равностойност както чрез метода Equals(), така и операторите за сравнение == и !=.

Методите на System.Object

Пример:

```
string s1 = "Maria";
string s2 = "Tom";
Console.WriteLine(Equals(s1, s2));//False
Console.WriteLine(Equals(s1, "Maria"));//True
Console.WriteLine(s1.Equals(s2));//False
Console.WriteLine(s1.Equals("Maria"));//True
```

Методите на System.Object

- **virtual string ToString()** – виртуален метод, който представя обекта във вид на символен низ (стринг).
- За аритметическите типове връща стойността, преобразувана в стринг.
- Подразбиращото се поведение на ToString, за референтните типове е да върне името на типа.
- Затова е хубаво да се предефинира, за да връща нещо по-смислено.

Пример:

```
int x = 255;
Console.WriteLine("type {0} is {1} , value is {2}", "x",
    x.GetType(), x.ToString());
// type x is System.Int32, value is 255
```

Методите на System.Object

- **virtual string ToString()** – Пример за препокриване

```
using System;
    class Name
    {       public string FirstName; //Поле FirstName
            public string LastName;  //Поле LastName
            public override string ToString()  //Метод
            { return FirstName + " " + LastName; }
            // Препокриване на виртуален метод ToString()
    } //class Name
    class Class1
    {       static void Main()
            {           Name my_name = new Name();
                        my_name.FirstName = "Tom";
                        my_name.LastName = "Tailor";
            Console.WriteLine("My name is {0}", my_name.ToString());
                        Console.ReadLine();
            }
} // Резултат:  My name is Tom Tailor
```

- **virtual int GetHashCode()** – виртуален метод за извличане на **хеш кода** (уникален идентификатор от тип int) на обекта. Използва се при реализацията на някои структури от данни, например хеш-таблици.

static bool ReferenceEquals(object obj1, object obj2) – сравнява двата обекта obj1 и obj2 по референция (проверка за „**идентичност**”): сравнява се дали обектите сочат към едно и също място в динамичната памет (резултатът е true ако obj1 и obj2 сочат на едно и също място). „Идентичността” позволява да се провери дали два обекта са всъщност един и същи обект (все едно да сравним адресите на обектите при C++).

```
object a = 4;   object b=a;
Console.WriteLine(ReferenceEquals(a,b));
//True
```

Стойност null и Nullable-типове

- Една от разликите между референтните и стойностните типове е, че променливите от референтен тип могат да приемат стойност null, а стойностните - не.

Тоест, коректно е:

  - **object o = null;**
  - **string s = null;**

- Има ситуации обаче, при които е удобно числовите типове да могат да имат стойност null, тоест да могат да бъдат неопределени.

- Стандартният пример: при работа с база от данни (често БД работят с други системи от типове, различни от тази на C#), типовете, които за C# са стойностни, в БД биха могли да съдържат стойност null. При това, ние не можем предварително да знаем, какво ще получим от БД като стойност за тези типове – конкретна стойност или null (тоест неопределена стойност).
- За тази цел в C# са създадени Nullable типове.
- Ако напишем знак ? след типа, създаваме Nullable тип. Например:
  - **int? z = null;**
  - **bool? enabled = null;**

- Но фактически запис ? се явява опростена форма на използване на структурата System.Nullable<T>.
- Параметър T в ъгловите скоби представлява универсален параметър, вместо който в конкретната задача се поставя конкретен тип данни.
- Следващите видове дефиниции на променливи ще бъдат еквивалентни:
  - **int? z1 = 5;**
  - **bool? enabled1 = null;**
  - **Double? d1 = 3.3;**
    - **Nullable<int> z2 = 5;**
    - **Nullable<bool> enabled2 = null;**
    - **Nullable<System.Double> d2 = 3.3;**

Nullable типове - свойства

- За всички Nullable типове са валидни 2 свойства:
  - **Value**, което **представлява стойността на обекта**, и
  - **HasValue**, което **връща true, ако Nullable обекта съхранява някаква стойност**.
- При това свойство Value съхранява обект от типа, чрез който се типизира Nullable:

```
class Program
{  static void Main()
   {
int? x = 7;    Console.WriteLine(x.Value);          // 7
Nullable<State> state = new State() { Name = "Nana" };
Console.WriteLine(state.Value.Name);          // Nana
 Console.ReadLine();
   }
}
struct State
{    public string Name { get; set; }
}
```

Nullable типове - свойства

Но, ако се опитаме да получим стойност на променлива, която е равна на null, то ще се получи грешка, т.е:

```
int? x = null;
Console.WriteLine(x.Value);                    // Грешка
```

- В този случай е необходимо да се изпълнява проверка за наличие на стойност, с помощта на свойство **HasValue**:

```
int? x = null;
if(x.HasValue)
Console.WriteLine(x.Value);
else
Console.WriteLine("x is equal to null");
```

Така също, структура Nullable с помощта на метода **GetValueOrDefault()** позволява да се използва стойноста по подразбиране за типа (за числовите типове това е 0), ако стойност по подразбиране не е зададена:

```
...
int? figure = null;
Console.WriteLine(figure.GetValueOrDefault());
// по подразбиране, стойността int е 0
Console.WriteLine(figure.GetValueOrDefault(10));
// по подразбиране се използва 10
```

Равенство на обекти

- При проверка на Nullable обектите за равенство следва да се отчита, че те са **равни не само когато имат равни ненулеви стойности, но и когато и двата обекта са равни на null**:

```
int? x1 = null; int? x2 = null;
if (x1 == x2)
    Console.WriteLine("равни обекти ");
else
    Console.WriteLine("неравни обекти ");
```

Преобразуване на Nullable типове

Ето възможните преобразования:

- явно преобразуване от T? към T

```
int? x1 = null;
if(x1.HasValue)
{
    int x2 = (int)x1;
    Console.WriteLine(x2);
}
```

- неявно преобразуване от T към T?

```
int x1 = 4;
int? x2 = x1;
Console.WriteLine(x2);
```

- неявно разширяващо преобразуване от V към T?

```
int x1 = 4;
long? x2 = x1;
Console.WriteLine(x2);
```

- явно свиващо преобразуване от V към T?

```
long x1 = 4;
int? x2 = (int?)x1;
```

- По подобен начин работи и преобразуването от V? към T?

```
long? x1 = 4;
int? x2 = (int?)x1;
```

- явно преобразуване от V? към T

```
long? x1 = 4;
int x2 = (int)x1;
```

Оператор ??

- Оператор ?? се нарича оператор за null-обединение. Той се използва за задаване на стойност по подразбиране на стойностни типове и на референтни типове, които допускат стойност null.

Оператор ??

- Оператор ?? връща левия операнд, ако този операнд не е равен на null. Иначе връща десния операнд, при което левият операнд е длъжен да е null. Да видим примера:

```
int? x = null;
int y = x ?? 100; // y=100
// връща 100, тъй като x е  null
 int? z = 200;
int t = z ?? 44; // t=200
// връща 200, тъй като z не е равен null
```

Оператор ??

- Но ние не можем да напишем следното:

```
int x = 44;
int y = x ?? 100;
```

- Причината е, че x не е тип Nullable и не може да приема стойност null, следователно не може да се използва в качество на ляв операнд в операция ??.