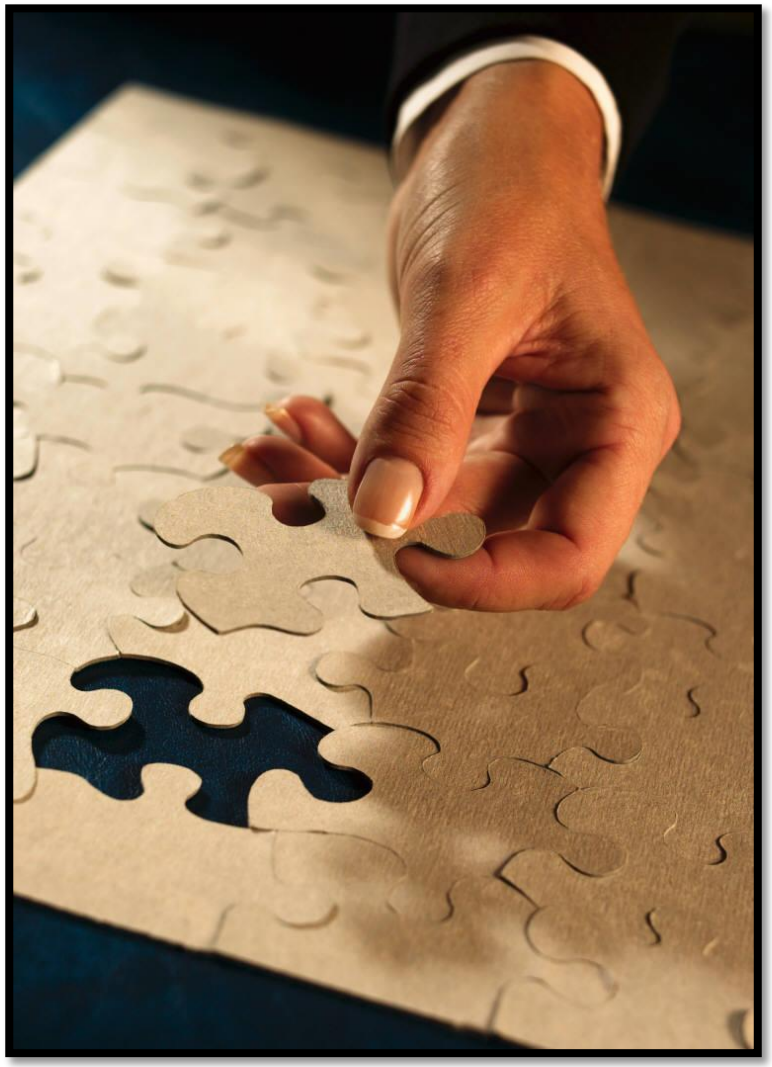

# Modulul 13

## C# 3.0, 4.0 și 5.0

# Overview

- Parametri cu nume și parametri opționali
- Inițializarea la instanțiere a obiectelor
- Tipuri anonime
- Legare dinamică
- Metode extensie
- Expresii lambda
- Caller Information



# Ce reprezintă?

- Reprezintă îmbunătățiri aduse limbajului acolo unde era nevoie de mult cod pentru a implementa o idee simplă
- În anumite situații, elimină necesitatea de a utiliza construcții suplimentare
- Facilitează implementarea unor concepte deja existente
  - Unele lucruri pot fi scrise mai simplu
  - Complexitatea aplicațiilor care pot fi scrise rămâne aceeași

# Parametri opționali

- Sunt parametri care au asignată o valoare default
- Funcțiile se comporta la fel la apel
  - Execută același cod
  - Pot fi precizați toți parametrii
  - Trebuie precizați cel puțin parametrii obișnuiți

```
static int sum(int val1, int val2, int val3 = 0)
{
    return val1 + val2 + val3;
}
public static void Main( )
{
    int suma = sum(1,2);        //1+2+0 => 3
    suma = sum(1,2,3);          //1+2+3 => 6
}
```

# Parametri cu nume

- Oferă noi posibilități de a apela funcții
  - Parametrii sunt precizați după numele pe care îl au
  - Poate fi inversată ordinea parametrilor
- Util atunci când
  - Există mai multi parametri cu același tip, care pot fi confundați
  - Trebuie specificat doar unul dintre parametrii opționali

```
static void SetTime(int hours, int min, int sec=0, int msec=0)
{ ... }
public static void Main( )
{
    SetTime(min : 7, hours : 3);      //03:07:00.000
    SetTime(0, 0, msec : 260);        //00:00:00.260
}
```

# Inițializarea la instanțiere a obiectelor

- Există posilitatea de a inițializa obiecte de orice tip asemănător variabilelor cu tip de bază sau a vectorilor

```
int numar = 5;
int[] vector = { 1, 2, 3 };
```

- La obiecte, se inițializează variabilele publice
  - Se pot specifica, după nume, doar variabilele publice ale clasei din care face parte obiectul

```
class Counter
{ public int initialValue;
  ...
}
public static void Main( )
{
    Counter val = new Counter { initialValue = 0 };
}
```

# Tipuri anonime

- Reprezintă obiecte "read only", care nu aparțin de nici o clasă
  - Se declară folosind cuvântul cheie **var**
- Tipul obiectelor
  - Este stabilit atunci când sunt create, în funcție de modul în care sunt inițializate
  - Este valabil doar în domeniul de vizibilitate, în momentul creării

```
public static void Main( )
{
    var jucator = new { nume = "Andrei", loc = 2 };
    Console.WriteLine("{0} - locul {1}",
                                jucator.nume, jucator.loc);
    //Andrei – locul 2
}
```

# Legare dinamică

- Este un mod de a apela dinamic câmpuri și metode ale unui obiect al cărui tip nu este cunoscut în momentul compilării
  - Se foloseste cuvântul cheie **dynamic**
- Oferă posibilitatea de a utiliza aceeași metodă pentru obiecte care aparțin unor clase diferite pe lanțul de moștenire
  - Nu este necesar a se mosteni o interfață care să introducă o informație referitoare la existența unor metode în acele clase
- Trebuie avut în vedere că eventualele probleme vor fi semnalate la rulare și nu în momentul compilării

# Legare dinamică

- Mecanismul de apel în cadrul legării dinamice îl reprezintă numele metodelor
  - La rulare, metoda este căutată chiar după nume, deoarece, în momentul compilării, nu se știa nimic despre structura obiectului

```
public static void PrintLength(dynamic obj)
{
    Console.WriteLine("Lungime: " + obj.Length);
}
public static void Main()
{
    int[] vector = new int[10];
    string cuvant = "mamaliga";
    PrintLength(vector);  //Lungime: 10
    PrintLength(cuvant);  //Lungime: 8
}
```

# Metode extensie

- Oferă posibilitatea de a adăuga noi metode la o clasă deja existentă, fără a fi necesară utilizarea moștenirii
  - Nu afectează structura logică bazată pe clase a unei aplicații
  - Funcționează acolo unde clasele sunt declarate cu "sealed" (nu pot fi moștenite).

- După ce au fost create, metodele extensie pot fi apelate la fel ca orice metodă existentă în acea clasă
  - Metodele extensie reprezintă un mod de a adăuga cod pentru o clasă fără a avea acces direct la codul acelei clase
  - La compilare, codul clasei arată la fel cum ar arăta dacă metoda ar fi fost declarată clasic

# Definirea metodelor extensie

- Metodele extensie trebuie definite într-o clasă statică
  - Preferabil a se utiliza o singură clasă statică pentru definirea de metode extensie referitoare la aceeași clasă propriu-zisă
  - Clasa trebuie să fie declarată imediat în "namespace" (clasă "top level")
- Metodele extensie sunt definite ca funcții statice
- Primul parametru trebuie să aibă tipul clasei pentru care se face extensia și să fie precedat de cuvântul cheie "this"
- Următorii parametri sunt opționali și reprezintă parametrii cu care va fi apelată metoda, prin intermediul unui obiect

# Exemplu metode extensie

Se va extinde clasa "String" cu metoda "ContainsVocals"

- Observați că "StringExtension" este în afara clasei "Program"
- Observați că metoda se apelează fără nici un parametru

```
public static class StringExtension
{
    public static bool ContainsVocals(this string obj)
    {
        return (obj.IndexOfAny("aeiou".ToCharArray()) != -1);
    }
}
public class Program
{
    public static void Main()
    {
        Console.WriteLine("abcdefghijk".ContainsVocals()); //True
        Console.WriteLine("msvcrt.dll".ContainsVocals());  //False
    }
}
```

# Expresiile lambda

- Reprezintă un mod mai simplu de a scrie funcții care efectuează calcule

- Funcțiile astfel create nu au nume, dar se pot reține prin intermediul unei variabile delegat corespunzătoare

- Sunt foarte utile atunci când este nevoie să se paseze o funcție de calcul ca argument pentru o altă funcție
  - O lambda expresie se poate folosi împreună cu "System.Array.Sort" pentru a sorta descrescător un vector
  - Se evită crearea de funcții statice folosite o singură dată

# Definirea expresiilor lambda

- Ca orice procedură, expresiile lambda primesc valori și returnează rezultatul obținut în urma unui calcul
- Sintaxă
  - Parametrii de intrare
  - Operatorul lambda "=>"
  - Expresia care calculează rezultatul

```
( variabile ) => expresie

sau

( variabile ) => {expr1; expr2; ... exprFinala;}
```

  - În al doilea caz, rezultatul final este obținut prin utilizarea instrucțiunii "return", la fel ca la metode

# Exemple de expresii lambda

- Funcție care incrementează valoarea argumentului cu 1

```
x => x + 1
```

- Funcție care verifică dacă cele 2 argumente sunt egale

```
(x, y) => x == y
```

- Funcție care verifică dacă un "string" este mai lung decât o anumită valoare

```
(int x, string s) => s.Length > x
```

# LINQ- Language-Integrated Query

- LINQ – permite utilizatorilor să formuleze interogări în cadrul limbajului C#
  - Interogările pot rula asupra unor surse de date diferite: structuri de date, documente XML, Ado.Net etc.

```
int[] numere= new int[] {5, 7, 1, 4, 9, 3, 2, 6, 8};

var numereMici= from n in numere
                where n <= 5
                orderby n
                select n;

foreach(var n in numereMici) {
    Console.WriteLine(n);
}
```

# LINQ și expresiile lambda

- Expresiile lambda sunt foarte utile în scrierea cererilor Linq

```
int[] numere= new int[] {5, 7, 1, 4, 9, 3, 2, 6, 8};

var numereMici = numere.Where (n => n <= 5).OrderBy(n=>n)

foreach(var n in numereMici) {
   Console.WriteLine(n);
}
```

# Caller Information - Informații despre apelant

- Atributele Caller Information sunt folosite pentru a obține informații despre apelantul unei metode:
  - CallerFilePath – calea fișierului sursă care conține apelantul
  - CallerLineNumber – numărul liniei, din fișierul sursă, de unde este apelată metoda
  - CallerMemberName – numele metodei sau al proprietății care apelează

# Exemplu de utilizare a atributelor Caller Information

```
public void Execută()
{
        LogareInformații(“Noi informații.”);
}

public void LogareInformații (string mesaj,
                                [CallerMemberName] string apelant = "",
                                [CallerFilePath] string caleFișier = "",
                                [CallerLineNumber] int numărLinie = 0)
{
        Console.WriteLine(“ Mesaj: “ + mesaj);
        Console.WriteLine(“ Apelant: “ + apelant);
        Console.WriteLine(“ Cale fișier sursă: “ + caleFișier);
        Console.WriteLine(“ Număr linie: “ + numărLinie );
}
```

# Sumar

- Parametri cu nume și parametri opționali
- Inițializarea obiectelor la creare
- Tipuri anonime
- Legare dinamică
- Metode extensie
- Expresii lambda
- Caller Information