

CURS 12

LINQ

Proiect realizat de: Bălan Andrei, Călugărițoiu Teodora,
Pui Andreea, Podea Maria, Russu Mihaela, Sasai Danissa

Cuprins

- 1.** Introducere în LINQ
- 2.** Metode utilizate
- 3.** XML

LINQ

- Set de tehnologii a căror scop este integrarea unei **sintaxe query-like** direct în C#.
- Se folosesc 2 tipuri de sintaxă:
 1. Query Syntax
 2. Method Syntax

Interfețe și Clase

- Pentru a putea folosi LINQ:
 - a. Pe date din memorie <-> interfața **IEnumerable<T>**
 - b. Pe date remote (ex: din baza de date) <-> interfața **IQueryable<T>**
- Funcțiile utilizate se regăsesc în 2 clase: **Enumerable** și **Queryable**.

Deferred & Immediate

1. Execuție immediate = obținerea rezultatului are loc **în momentul apelului query-ului**
2. Execuție deferred = obținerea rezultatului este **amânată până când valoarea acestuia este cerută**

Execuția deferred <-> lazy evaluation

Deferred & Immediate

- Execuția deferred este **mai eficientă** (overhead distribuit uniform) => se încearcă folosirea **cât mai des** a acesteia.
- Funcțiile ce întorc **o singură valoare/o listă** (ex: Count(), ToList(), ToArray(), etc.) forțează **execuția imediată**.

Deferred

```
IEnumerable<string> listaAnimale = ["Raton", "Ratusca", "Leu", "Caine", "Rata", "Iepure"];
var rez :IEnumerable<string> = listaAnimale
    .Select(s :string => s) // IEnumerable<string>
    .OrderBy(s :string => s) // IOrderedEnumerable<string>
    .Where(s :string => s.StartsWith('R'));
```

Immediate

```
IEnumerable<string> listaAnimale = ["Raton", "Ratusca", "Leu", "Caine", "Rata", "Iepure"];
int rez = listaAnimale
    .Select(s :string => s) // IEnumerable<string>
    .Count(s :string => s.EndsWith('a') || s.EndsWith('e'));
```

Iterarea unui sir infinit folosind yield return și LINQ

```
IEnumerable<ulong> FibonacciNumbers()
{
    yield return 0;
    yield return 1;

    ulong previous = 0, current = 1;
    while (true)
    {
        ulong next = checked(previous + current);
        yield return next;
        previous = current;
        current = next;
    }
}

var firstTenOddFibNumbers :IEnumerable<ulong> = FibonacciNumbers().Where(n :ulong =>n%2 == 1).Take(10);
foreach (var num :ulong in firstTenOddFibNumbers)
{
    Console.WriteLine(num);
}
```

Avantaje LINQ

- Codul devine **mai clar și mai lizibil**
(se elimină buclele for/foreach)
- Sintaxă unificată pentru surse diferite de date
ex: aceleași operații (**Where**, **Select**, **Join**) se folosesc pentru:
 1. colecții în memorie
 2. baze de date
 3. XML
- Type Safety → erorile de tip sunt detectate la compilare
- Suport IntelliSense

Dezavantaje LINQ

- Performanță inferioară în anumite cazuri
- Debugging mai dificil
(mai greu de urmărit pas cu pas față de foreach)
- Funcționalitățile din SQL sunt **limitate** în LINQ

Prim exemplu LINQ

PROBLEMĂ: Să se determine toate sirurile de caractere de lungime 3.

Varianta clasică

```
string[] words = {"one", "two", "three"};
foreach (string word in words)
{
    if (word.Length == 3)
        Console.WriteLine(word);
}
```

Varianta LINQ

```
var selectQuery :IEnumerable<string> = from word :string in words
                                         where word.Length == 3
                                         select word;
var selectMethod :IEnumerable<string> = words.Where(w :string => w.Length == 3);
foreach (var word :string? in selectQuery)
    Console.WriteLine(word);
```

→ sintaxa QUERY

→ sintaxa METHOD

Aggregate

- Aplică o operație pentru **fiecare element** al colecției, **ținând rezultatul** în continuare.
- Are 3 parametrii:
 - **seed** (**optional**), valoarea inițială pentru accumulator
 - **func** (**obligatoriu**), funcția care va fi apelată pentru fiecare element
 - **resultSelector** (**optional**), funcție care va modifica accumulatorul în rezultat

Aggregate

```
string[] duckNames = { "Petra", "Rudky", "Rata", "Marghiola", "Jim" };
string longestName = duckNames.Aggregate((longest :string , next :string ) => longest.Length > next.Length ? longest : next);
Console.WriteLine("Longest Name: {0}", longestName);
```

Va afişa:

Longest Name: Marghiola

```
int[] numbers = { 476, 523, 432, 326 };
int nr = numbers.Aggregate(0,(result:int , next:int) => result + SumaCifre(next));
Console.WriteLine("Suma tuturor cifrelor: {0}", nr);
```

Va afişa:

Suma tuturor cifrelor: 47

```
public static int SumaCifre(int a)
{
    int x = 0;
    while (a != 0)
    {
        x = x + a % 10;
        a /= 10;
    }
    return x;
}
```

Any

- Verifică dacă un element în colecție **satisfacă o condiție**.

```
int[] speeds = { 23, 12, 32, 54, 2 };
bool result = speeds.Any(speed => speed > 50);
Console.WriteLine("Is there a speed above 50 : {0}", result);
```

Va afișa:

Is there a speed above 50 : True

Concat

- Returnează **rezultatul concatenării** a celor două colecții

```
int[] numbers1 = { 1, 2, 3 };
int[] numbers2 = { 4, 5, 6 };
var result :IEnumerable<int> = numbers1.Concat(numbers2);

foreach (var i:int in result)
{
    Console.WriteLine(i);
}
```

Va afișa:

1
2
3
4
5
6

SelectMany

- Proiectează *fiecare element* al colecției și combină secvențele rezultate într-o singură secvență. Asemănător cu *cross join* în SQL.
- Are 2 parametrii:
 - *collectionSelector*, o funcție de transformare care este aplicată fiecărui element din colecție
 - *resultSelector (optional)*, o funcție de transformare care se aplică fiecărui element intermediar

SelectMany

```
string[] fruits = { "Grape", "Orange", "Apple" };
int[] amounts = { 1, 2, 3 };
var result :IEnumerable<{Fruit,Amount}> = fruits.SelectMany(f :string => amounts, (f :string , a :int ) => new
{
    Fruit = f,
    Amount = a
});
Console.WriteLine("Selecting all values from each array, and mixing them:");
foreach (var o :{Fruit,Amount} in result)
    Console.WriteLine(o.Fruit + ", " + o.Amount);
```

Va afişa:

```
Selecting all values from each array, and mixing them:
Grape, 1
Grape, 2
Grape, 3
Orange, 1
Orange, 2
Orange, 3
Apple, 1
Apple, 2
Apple, 3
```

Exemplul 1

To Dictionary

- se folosește pentru a transforma o **coleție(List, Array)** într-o structură de date de tip dicționar(Dictionary).
- metoda `toDictionary` are 2 argumente:
 - Selectorul de Cheie (obligatoriu)**-functie care specifică proprietatea unică ce va servi drept cheie de identificare
 - Selectorul de Valoare (optional)**-funcție care ne permite să specificăm ce date să fie stocate în dicționar (întregul obiect sau doar o anumită proprietate)

To Dictionary-Exemplu

PROBLEMĂ: Se dă o listă de produse.

- 1.) Transformați lista într-un dicționar ce are cheia primară id-ul produsului.
- 2.) Transforimați lista lista într-un dicționar ce are cheia primară id-ul produsului și valoarea atribuită cheii să fie doar numele produsului.

To Dictionary-Exemplu

```
public class Produs
{
    □ 5 usages
    public int Id { get; set; }
    □ 5 usages
    public string Nume { get; set; }
    □ 4 usages
    public decimal Pret { get; set; }
}

List<Produs> listaProduse = new List<Produs>
{
    new Produs { Id = 1, Nume = "Laptop", Pret = 3500 },
    new Produs { Id = 2, Nume = "Mouse", Pret = 100 },
    new Produs { Id = 3, Nume = "Tastatura", Pret = 250 }
};
```

To Dictionary-Exemplu

Sintaxa QUERY

```
//Creeaza un Dictionary<int, Produs>
var dictionarProduse :Dictionary<int,Produs> = (from p :Produs in listaProduse select p)
    .ToDictionary(p :Produs => p.Id);
```

```
//Creeaza un Dictionary<int, Produs>
var dictionarProduse :Dictionary<int,Produs> = (from p :Produs in listaProduse select p)
    .ToDictionary(p :Produs => p.Id);
```

*sau putem pune direct listaProduse.ToDictionary(...)

To Dictionary-Exemplu

Sintaxa METHOD

```
//Creeaza un Dictionary<int, Produs>
var dictionarProduse :Dictionary<int,Produs> = listaProduse.ToDictionary(p :Produs => p.Id);
```

```
//Creeaza un Dictionary<int, string>
//Primul lambda este Cheia (p.Id), al doilea este Valoarea (p.Nume)
var dictionarNume :Dictionary<int,string> = listaProduse.ToDictionary(p :Produs => p.Id, p :Produs => p.Nume);
```

Group By

- se folosește pentru a grupa elementele unei colecții în funcție de o proprietate(**Key=criteriu de grupare**)
- returnează o colecție de grupuri

PROBLEMĂ: Se dă o listă de produse. Grupați produsele în funcție de prețul lor

Group By-Exemplu

```
List<Produs> listaProduse = new List<Produs>
{
    new Produs { Id = 1, Nume = "Laptop", Pret = 3500 },
    new Produs { Id = 2, Nume = "Mouse", Pret = 100 },
    new Produs { Id = 3, Nume = "Tastatura", Pret = 250 },
    new Produs { Id = 4, Nume = "Monitor", Pret = 3500 },
    new Produs { Id = 5, Nume = "MousePad", Pret = 100 },
    new Produs { Id = 6, Nume = "Casti", Pret = 250 }
};
```

Group By-Exemplu

Sintaxa QUERY

```
var grupuri:IEnumerable<IGrouping<...>> = from p :Produs in listaProduse group p by p.Pret;
```

Sintaxa METHOD

```
var grupuri:IEnumerable<IGrouping<...>> = listaProduse.GroupBy(p :Produs => p.Pret);
```

Distinct

- se folosește pentru a elimina duplicatele dintr-o colecție, returnând o nouă secvență care conține **doar elementele unice**

PROBLEMĂ: Se dă o listă de numere. Să se formeze o nouă listă ce conține doar elementele unice din lista originală.

Distinct-Exemplu

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

Sintaxa QUERY

```
var numereDistincte :IEnumerable<int> = (from nr:int in numere select nr).Distinct();
```

Sintaxa METHOD

```
var numereDistincte :IEnumerable<int> = numere.Distinct();
```

*sau putem pune pentru sintaxa QUERY numere.Distinct()

Except

- se folosește pentru a **elimina** toate elementele dintr-o colecție care **există și în altă colecție**
- Except **elimină duplicatele** din prima colecție indiferent dacă acestea apar sau nu în cea de-a doua colecție

PROBLEMĂ: Se dau 2 liste de numere. Să se formeze o nouă listă ce conține doar elementele unice din prima listă ce **nu apar** în a doua listă.

Except-Exemplu

```
int[] numere1 = { 1, 2, 1, 1, 3, 3, 5, 6 };
int[] numere2 = { 3, 4, 5 };
```

Sintaxa QUERY

```
var rezultat :IEnumerable<int> = (from n:int in numere1 select n).Except(numere2);
```

Sintaxa METHOD

```
var rezultat :IEnumerable<int> = numere1.Except(numere2);
```

*sau putem pune pentru sintaxa QUERY numere1.Except(numere2)

Exemplul 2

Element At Or Default

- elementul de pe poziția k (dacă există poziția k în listă)
- valoarea default (dacă nu există poziția k în listă)
(prima poziție se consideră 0)

```
string[] flyingDucks = { "Duck1", "Duck3", "Duck4", "Duck6", "Duck7" };
```

```
var resultIndex3 :string? = flyingDucks.ElementAtOrDefault(3);
```

```
var resultIndex11 :string? = flyingDucks.ElementAtOrDefault(11);
```

```
Console.WriteLine(resultIndex3); ?
```

```
Console.WriteLine(resultIndex11); ?
```

Join

- foarte similar cu INNER JOIN-ul din SQL
- combină/asociază date din 2 colecții **pe baza unei chei comune**

```
string[] swimmingDucks = { "Duck1", "Duck2", "Duck3", "Duck5", "Duck7" };
string[] flyingDucks = { "Duck1", "Duck3", "Duck4", "Duck6", "Duck7" };
```

PROBLEMĂ: Să se determine rațele de tip **flying** și **swimming**

Join - Exemplu

Sintaxa QUERY

```
var swimmingAndFlyingDucks1:IEnumerable<string> = (from sDuck:string in swimmingDucks  
join fDuck:string in flyingDucks on sDuck equals fDuck  
select fDuck);
```

Sintaxa METHOD

```
var swimmingAndFlyingDucks2:IEnumerable<string> = swimmingDucks.Join  
(flyingDucks,  
    sDuck:string => sDuck,  
    fDuck:string => fDuck,  
    (sDuck:string, fDuck:string) => fDuck  
);
```

Order By

- ordonează o colecție după un anumit parametru

PROBLEMĂ: Se dă o listă de persoane. Să se ordoneze lista
lista descrescător după numele persoanelor.

Person → obiect cu 2 parametrii: nume (string), prenume (string)

```
Person[] people =  
{  
    new Person( name: "Popescu",   firstName: "Andrei") ,  
    new Person( name: "Mana" ,   firstName: "Malina") ,  
    new Person( name: "Mana" ,   firstName: "Ana") ,  
    new Person( name: "Zarin" ,   firstName: "Mihnea") ,  
};
```

Order By - Exemplu

Sintaxa QUERY

```
var sortedPeople :IOrderedEnumerable<Person> = from p :Person in people  
    orderby p.Name descending  
    select p;
```

Sintaxa METHOD

```
var sortedPeople2 :IOrderedEnumerable<Person> = people  
    .OrderByDescending(x :Person => x.Name);
```

Then By

- se folosește după sortarea cu OrderBy pentru a sorta încă o dată lista, după un alt parametru

PROBLEMĂ: Se dă o listă de persoane. Să se ordoneze lista
lista descrescător după numele persoanelor,
iar apoi, crescător după prenume.

Then By - Exemplu

Sintaxa QUERY

```
var sortedPeople3 :IOrderedEnumerable<Person> = from p :Person in people  
orderby p.Name descending, p.FirstName ascending  
select p;
```

Sintaxa METHOD

```
var sortedPeople4 :IOrderedEnumerable<Person> = people // Person[]  
.OrderByDescending(person => person.Name)  
.ThenBy(person => person.FirstName);
```

Exemplul 3

XML (eXtensible Markup Language

- este un **format de stocare** și transport al datelor, bazat pe **tag-uri**.
- este un mod standard de **a descrie datele** într-un format clar și organizat, astfel încât:
 - oamenii să-l poată citi
 - **aplicațiile diferite** să se înțeleagă între ele

Structura XML în C# cu LINQ to XML

```
XDocument (Documentul XML complet)
└── XElement (Element principal/root)
    ├── XAttribute (Atribut al elementului)
    └── XElement (Sub-element / copil)
        └── XAttribute (Atribut optional)
```

Explicații

- **XDocument** – reprezintă întregul document XML în memorie, adică **rădăcina** tuturor elementelor și structurii XML. Este **“containerul principal”** care ține tot XML-ul.
- **XElement** reprezintă un **element individual** din XML, adică o etichetă cu conținut și eventual sub-elemente.
- **XAttribute** reprezintă un **atribut asociat** unui element XML, adică o proprietate sau metadată care descrie elementul.

Ferma Vesela

10 Mai 2024

Lacul Albastru - Zona de Nord



Donald

Varsta: 3 ani

Greutate: 2.4 kg



Daisy

Varsta: 2 ani

Greutate: 2.0 kg



Scrooge

Varsta: 6 ani

Greutate: 3.1 kg

Exemplu rezolvat

```
<?xml version="1.0" encoding="UTF-8"?>
<Ferma data="2024-05-10" nume="Ferma Vesela">

    <Lac>
        <Nume>Lacul Albastru</Nume>
        <Locatie>Zona de Nord</Locatie>
    </Lac>

    <Rate>
        <Rata nume="Donald">
            <Varsta>3</Varsta>
            <Greutate>2.4</Greutate>
            <Culoare>Alb</Culoare>
        </Rata>
        <Rata nume="Daisy">
            <Varsta>2</Varsta>
            <Greutate>2.0</Greutate>
            <Culoare>Galben</Culoare>
        </Rata>
        <Rata nume="Scrooge">
            <Varsta>6</Varsta>
            <Greutate>3.1</Greutate>
            <Culoare>Alb</Culoare>
        </Rata>
    </Rate>
</Ferma>
```

Exemplu rezolvat(2)

- **XDocument**: Întregul fișier **ferma.xml**
- **XElemente**:
 - **<Ferma>** - element principal (root)
 - **<Lac>** - sub-element pentru **<Ferma>**
 - **<Rata>** - sub-element pentru **<Rate>**
 - **<Varsta>** - sub-element pentru **<Rata>**
- **XAttribute**:
 - **<Rata nume="Donald">** - atributul nume

Cum funcționează în spate?

1. Aplicația **deschide** fișierul XML
2. Un parser XML îl **citește** caracter cu caracter
3. **Verifică** dacă respectă regulile XML
4. **Creează** o structură internă de tip arbore, unde fiecare tag devine un nod(astfel de creează obiectele C# în memorie)

Accesarea obiectelor din memorie

- Elementul root:
 - XElement ferma = doc.Root;
- Atributul unui element:
 - string numeFerma = ferma.Attribute("nume").Value;
- Navigarea către un sub-element:
 - XElement lac = ferma.Element("Lac");
 - string numeLac = lac.Element("Nume").Value;

Accesarea obiectelor din memorie(2)

- Citirea unei valori numerice
- Modificarea obiectelor din memorie

```
int varsta = (int)ferma  
    .Element("Rate")  
    .Element("Rata")  
    .Element("Varsta");
```

```
ferma.Element("Rate")  
    .Element("Rata")  
    .SetElementValue("Varsta", 4);
```

Alte reguli pentru XML

- XML trebuie să aibă **un singur element root**
- Fiecare tag deschis trebuie închis
- XML este **case-sensitive**
- Atributele nu trebuie duplicate
- Structura trebuie să fie **logică și clară**
- Inițial toate atrbutele sunt de tip **string**

Fisiere XML - scrierea

- Fiecare tag, atribut sau valoare se scrie individual.
- Funcții relevante:
 1. **XmlWriter.Create(pathFișier, setări)** – unde pathFișier este de tip **string**, iar setări de tip **XmlWriterSettings**
 2. **WriteStartDocument()** – scrie în fișier antetul specific

Fisiere XML - scrierea

3. **WriteStartElement(numeElement)** – scrie tagul de început al unui element
4. **WriteAttributeString(numeAtribut, valoareAtribut)**
 - scrie în interiorul tagului de început atributul cu numele și valoarea precizate
5. **WriteString(string)** – scrie valoarea unui element simplu

Fisiere XML - scrierea

6. **WriteEndElement(numeElement)** – scrie tagul de final al unui element
7. **WriteEndDocument()** – scrie finalul documentului XML
8. **Close()**

Fișiere XML - citire

Concept: Spre deosebire de scriere, citirea presupune încărcarea structurii în memorie pentru a fi interogată.

1. **XDocument.Load(pathFișier)** – unde pathFișier este de tip `string`, încarcă fișierul XML de la calea specificată într-un obiect
2. **Descendants(numeTag)** – returnează o colecție (`IEnumerable`) cu toate elementele care au numele specificat, indiferent unde se află în ierarhie.

Fișiere XML - citire

Odată ajuns la un nod (element), trebuie să extragem datele din el.

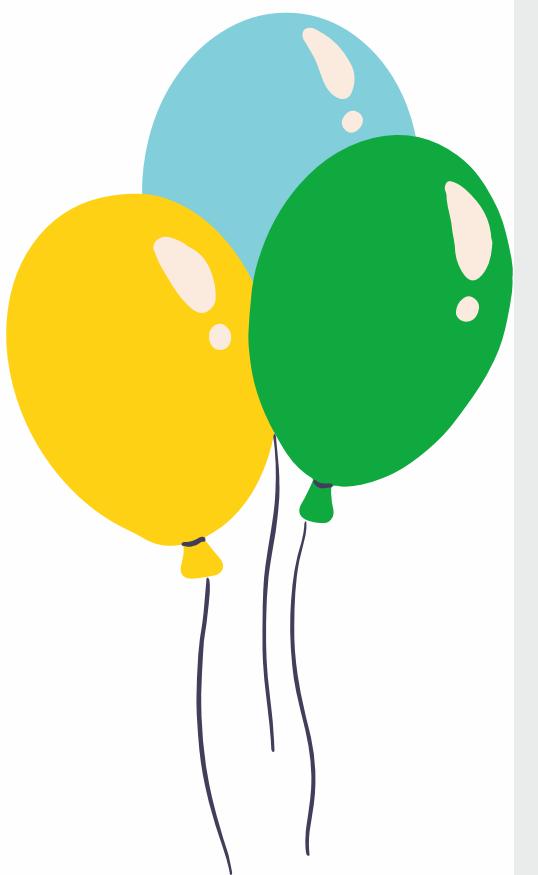
1. **Element(numeElement).Value** - returnează textul din interiorul unui tag copil specificat.
2. **Attribute(numeAtribut).Value** - returnează valoarea atributului specificat din tag-ul

Fișiere XML - citire

Datele citite sunt de obicei transformate în obiecte C# folosind LINQ

```
var cazuriCritice =  
    from c in docUrgente.Descendants("Copil")  
    where c.Attribute("status")?.Value == "Critic"  
    select new  
    {  
        Nume = c.Element("Nume")?.Value,  
        Oras = c.Element("Oras")?.Value,  
        Mesaj = c.Element("Mesaj")?.Value  
    };
```

Timpul pt Menti!



**MULTUMIM PENTRU
ATENTIE!**