

# Curs 4

LINQ – Language Integrated Query



# LINQ

- Motivatie LINQ
  - Aplicatiile vor accesa date la un anumit moment in timpul executiei
  - Datele pot fi regasite in fisiere XML, bd relationale, colectii in memorie, siruri
- LINQ – introdus in .NET 3.5 – ofera un modalitate puternic tipizata de a accesa date in diverse formate prin intermediul unui layer de abstractizare

# Structuri sintactice specifice

- LINQ poate fi inteles ca un limbaj de interogare puternic tipizat incorporat in gramatica C#
- Putem construi expresii asemanatoare cu interogările SQL, dar interogările SQL se pot aplica unor surse de date diferite inclusiv celor care nu au nimic de a face cu bd relationale
- Desi interogările LINQ sunt similare cu interogările SQL, sintaxa nu este identica. De fapt multe interogari LINQ folosesc un format diferit de cel utilizat la bd
- Nu incercam sa mapam sintaxa LINQ to SQL, ci mai degraba le privim ca interogari care “prin coincidenta” seamana cu SQL

# Mecanisme LINQ

- Cand LINQ a fost introdus in .NET, limbajul C# continea o serie de mecanisme pe care se bazeaza tehnologia LINQ
- Astfel, limbajul C# utilizeza urmatoarele mecanisme care se afla la baza LINQ:
  - Variabile cu tip implicit
  - Sintaxa de initializare a obiectelor/colectiilor
  - Expresii Lambda
  - Metode extinse
  - Tipuri anonime

# Variabile cu tip implicit

- Cuvantul cheie **var** permite declararea unei variabile locale fara a specifica explicit tipul variabilei
- Totusi, variabila este puternic tipizata, deoarece compilatorul va determina tipul de date corect bazandu-se pe atribuirea valorii

```
static void DeclareImplicitVars()
{
    // Implicitly typed local variables.
    var myInt = 0;
    var myBool = true;
    var myString = "Time, goes on...";
    // Print out the underlying type.
    Console.WriteLine("myInt is a: {0}", myInt.GetType().Name);
    Console.WriteLine("myBool is a: {0}", myBool.GetType().Name);
    Console.WriteLine("myString is a: {0}", myString.GetType().Name);
}
```

- Acest feature este foarte util cand utilizam LINQ – multe interogari LINQ vor returna o secventa de tipuri de date, care nu sunt cunoscute pana la momentul compilarii, deci nu vom putea declara tipul unei variabile explicit

# Sintaxa de initializare a obiectelor/colectiilor

- Sintaxa de initializare colectie pentru a umple o lista `List<T>` cu obiecte `Rectangle`, fiecare avand doua obiecte `Point` care reprezinta o pozitie determinata de doua coordonate (x,y):

```
List<Rectangle> myListOfRects = new List<Rectangle>
{
    new Rectangle {TopLeft = new Point { X = 10, Y = 10 },
    BottomRight = new Point { X = 200, Y = 200}},
    new Rectangle {TopLeft = new Point { X = 2, Y = 2 },
    BottomRight = new Point { X = 100, Y = 100}},
    new Rectangle {TopLeft = new Point { X = 5, Y = 5 },
    BottomRight = new Point { X = 90, Y = 75}}
};
```

- Aceasta sintaxa combinata cu variabilele cu tip implicit ne permit declararea de tipuri anonime, utile la crearea de proiectii LINQ
- Proiectii LINQ- transformarea unui obiect intr-o forma noua care de obicei consta intr-un subset de proprietati. Se pot realiza proiectii si cu obiectul original fara modificari

# Expresii Lambda

- Utilizam expresii lambda pentru a crea o functie anonima
- operator ( $\Rightarrow$ ) permite construirea unei expresii lambda- separa lista de parametrii de corpul functiei
- $( \text{ArgumentsToProcess} ) \Rightarrow \{ \text{StatementsToProcessThem} \}$
- Expresiile lambda simplifica modul de lucru in .NET, reducand numarul de linii de cod care trebuie scrise

# Metode extinse

- Metode extinse permit adaugarea de functionalitati la clasele existente fara a folosi mostenirea
- La crearea unei metode extinse primul parametru este calificat cu cuvantul cheie **this** marcand tipul care se extinde.
- Metodele extinse trebuie definite in cadrul unei clase statice si trebuie declarate folosind cuvantul cheie **static**

```
namespace MyExtensions
{
    public static class IntExtensions {
        public static bool IsGreaterThan(this int i, int value)
        { return i > value; }
    }
}

using MyExtensions;

class Program {
    static void Main(string[] args)
    { int i = 10;
      bool result = i.IsGreaterThan(100); Console.WriteLine(result);
    } }
```

Cand creem interogari LINQ, utilizam metode extinse definite in .NET.





# Tipuri anonime

- Generarea definitiei unei clase la compilare prin specificarea unui set de perechi nume-valoare
- Pentru a defini un tip anonim declaram o variabila cu tip implicit si specificam datele folosind sintaxa de initializare a obiectelor

// Creaza un tip anonim compus din alt tip anonim

```
var purchaseItem = new {  
    TimeBought = DateTime.Now,  
    ItemBought = new {Color = "Red", Make = "Saab", CurrentSpeed = 55},  
    Price = 34.000};
```

- LINQ foloseste tipuri anonime cand dorim sa proiectam noi feluri de date “on the fly”
- O colectie de obiecte Persoana – dorim sa utilizam LINQ pentru a obtine info despre varsta si CNP  
Folosind o proiectie LINQ, permitem compilatorului sa genereze un nou tip anonim care contine informatia dorita

# Termenii LINQ

- Utilizand LINQ putem crea expresii de interogare folosind limbajul C#
- LINQ poate fi folosit in mai multe cazuri iar termenii folositi difera in functie de acest lucru:
  - *LINQ to Objects*: utilizarea interogarilor LINQ la siruri si colectii
  - *LINQ to XML*: utilizarea LINQ pentru a manipula si interoga documente XML
  - *LINQ to Entities*: utilizarea interogarilor LINQ in cadrul Entity Framework.
  - *Parallel LINQ (PLINQ)*: procesarea paralela a datelor returnate de o interogare LINQ

# Interogari

- Specifica informatiile care trebuie incarcate din sursa de date
- Optional specifica cum trebuie sortate/grupate
- Sunt stocate intr-o variabila de interogare si initializate cu o expresie de interogare

```
var result = from matchingItem in container select matchingItem;
```

- Interogarile sunt separate de executia acestora

# Sintaxa de baza

- Corectitudinea sintactica a unei interogarii LINQ este validata la compilare => ordinea operatorilor este importanta
- Expresiile LINQ sunt construite utilizand operatorii **from**, **in**, **select**
- Template-ul general:  

```
var result = from matchingItem in container select matchingItem;
```
- Identificatorul aflat dupa operatorul **from** reprezinta un item care corespunde criteriilor introgarii LINQ – poate avea orice nume.
- Identificatorul aflat dupa operatorul **in** reprezinta containerul de date in care se face cautarea (sir, colectie, document XML, etc.).

# Selectarea itemilor din container

```
static void
SelectEverything(ProductInfo[]
products)
{
    Console.WriteLine("All product
details:");
    var allProducts = from p in
products select p;
    foreach (var prod in
allProducts)
    {
        Console.WriteLine(prod.ToString
());
    }
}
```

```
static void
ListProductNames(ProductInfo[]
products)
{
    Console.WriteLine("Only
product names:");
    var names = from p in products
select p.Name;
    foreach (var n in names)
    {
        Console.WriteLine("Name: {0}",
n);
    }
}
```

# Obtinerea de subseturi de date

- Template-ul general:

```
var result = from item in container where BooleanExpression select item;
```

```
static void GetOverstock(ProductInfo[] products)
{
    Console.WriteLine("The overstock items!");
    var overstock = from p in products where p.NumberInStock > 25
    && p.ExpDate=DateTime.Today.AddDays(7) select p;
    foreach (ProductInfo c in overstock)
    {
        Console.WriteLine(c.ToString());
    }
}
```

# Proiectarea unor noi tipuri de date

```
static void
GetNamesAndDescriptions(ProductInfo[]
products)
{
    Console.WriteLine("Names and
    Descriptions:");
    var nameDesc = from p in products
    select new { p.Name, p.Description
    }; // tip de date anonim
    foreach (var item in nameDesc)
    {
        Console.WriteLine(item.ToString());
    }
}
```

- Cand interogarea LINQ creaza o proiectie nu va cunoaste tipul de date- e obligatorie utilizarea **var**

```
static var
GetProjectedSubset(ProductInfo[]
products)
{
    var nameDesc = from p in products
    select new { p.Name, p.Description
    };
    return nameDesc; // Nu!
}

• Nu putem scrie metode care
returneaza tipuri implicite
return nameDesc.ToArray(); return
type Array
```

# Numarul de itemi returnati

```
static void GetCountFromQuery()
{
    string[] currentVideoGames = {"Morrowind", "Uncharted 2", "Fallout 3", "Daxter", "System Shock 2"};
    int numb = (from g in currentVideoGames where g.Length > 6 select g).Count();
    Console.WriteLine("{0} items for the LINQ query.", numb);
}
```

- Count() - metoda extinsa a clasei Enumerable



# Inversarea ordinii din setul obtinut

```
static void ReverseEverything(ProductInfo[] products)
{
    Console.WriteLine("Product in reverse:");
    var allProducts = from p in products select p;
    foreach (var prod in allProducts.Reverse())
    {
        Console.WriteLine(prod.ToString());
    }
}
```

- Reverse() - metoda extinsa a clasei Enumerable

# Sortarea

```
static void AlphabetizeProductNames(ProductInfo[] products)
{
    // produse in ordine alfabetica.
    var subset = from p in products orderby p.Name select p;
    Console.WriteLine("Ordered by Name:");
    foreach (var p in subset)
    {
        Console.WriteLine(p.ToString());
    }
}
```

- var subset = from p in products orderby p.Name **ascending** select p;
- var subset = from p in products orderby p.Name **descending** select p;

# Gruparea

```
var queryCustomersByCity =  
    from cust in customers  
    group cust by cust.City;  
  
foreach (var customerGroup in queryCustomersByCity)  
{  
    Console.WriteLine(customerGroup.Key);  
    foreach (Customer customer in customerGroup)  
    {  
        Console.WriteLine(" {0}", customer.Name);  
    }  
}
```

# Join

Gasim clientii si distribuitori care se afla in aceeasi locatie

```
var innerJoinQuery =  
    from cust in customers  
    join dist in distributors on cust.City equals dist.City  
    select new { CustomerName = cust.Name, DistributorName =  
dist.Name };
```

- creeaza asocieri intre secvente care nu sunt explicit modelate in sursa de date
- Clauza join se aplica colectiilor de obiecte nu direct tabelelor din bd
- Cheile straine din model – colectie de item-uri
- from order in Customer.Orders

# Diferenta intre doua containere

```
static void DisplayDiff()
{
    List<string> myCars = new List<String> {"Yugo", "Aztec", "BMW"};
    List<string> yourCars = new List<String>{"BMW", "Saab", "Aztec" };
    var carDiff = (from c in myCars select c).Except(from c2 in yourCars
    select c2);
    Console.WriteLine("Here is what you don't have, but I do:");
    foreach (string s in carDiff)
        Console.WriteLine(s);}
}
```

- Except() - metoda extinsa a clasei Enumerable

# Intersectia a doua containere

```
static void DisplayIntersection()
{
    List<string> myCars = new List<String> { "Yugo", "Aztec", "BMW" };
    List<string> yourCars = new List<String> { "BMW", "Saab", "Aztec"
};
    // Get the common members.
    var carIntersect = (from c in myCars select c).Intersect(from c2 in
yourCars select c2);
    Console.WriteLine("Here is what we have in common:");
    foreach (string s in carIntersect)
        Console.WriteLine(s);}
• Intersect() - metoda extinsa a clasei Enumerable
```



# Reuniunea

```
static void DisplayUnion()
{
    List<string> myCars = new List<String> { "Yugo", "Aztec", "BMW" };
    List<string> yourCars = new List<String> { "BMW", "Saab", "Aztec"
};
    // Get the union of these containers.
    var carUnion = (from c in myCars select c).Union(from c2 in
yourCars select c2);
    Console.WriteLine("Here is everything:");
    foreach (string s in carUnion)
        Console.WriteLine(s);
}
```

- Union() - metoda extinsa a clasei Enumerable



# Concatenarea

```
static void DisplayConcat()
{
    List<string> myCars = new List<String> { "Yugo", "Aztec", "BMW" };
    List<string> yourCars = new List<String> { "BMW", "Saab", "Aztec" };
    var carConcat = (from c in myCars select c).Concat(from c2 in yourCars
    select c2);
    foreach (string s in carConcat)
        Console.WriteLine(s);
}
```

- Concat() - metoda extinsa a clasei Enumerable



# Eliminarea duplicatelor

```
static void DisplayConcatNoDups()
{
    List<string> myCars = new List<String> { "Yugo", "Aztec", "BMW" };
    List<string> yourCars = new List<String> { "BMW", "Saab", "Aztec" };
    var carConcat = (from c in myCars select c).Concat(from c2 in
yourCars select c2);
    foreach (string s in carConcat.Distinct())
        Console.WriteLine(s);
}
```

- Distinct() - metoda extinsa a clasei Enumerable

# Operatori de agregare LINQ

```
static void AggregateOps()
{
    double[] winterTemps = { 2.0, -21.3, 8, -4, 0, 8.2 };
    // Exemple de agregare
    Console.WriteLine("Max temp: {0}", (from t in winterTemps select
    t).Max());
    Console.WriteLine("Min temp: {0}", (from t in winterTemps select
    t).Min());
    Console.WriteLine("Average temp: {0}", (from t in winterTemps
    select t).Average());
    Console.WriteLine("Sum of all temps: {0}", (from t in winterTemps
    select t).Sum());
}
```

# Cu LINQ vs. Fara LINQ

```
static void
QueryOverStrings()
{
    string[] currentVideoGames =
    {"Morrowind", "Uncharted 2",
    "Fallout 3", "Daxter",
    "System Shock 2"};
    var subset = from g in
    currentVideoGames where
    g.Contains(" ") orderby g
    select g;
    foreach (string s in subset)
    Console.WriteLine("Item:
    {0}", s);
}
```

```
static void QueryOverStringsLongHand()
{
    string[] currentVideoGames = {"Morrowind",
    "Uncharted 2", "Fallout 3", "Daxter", "System
    Shock 2"};
    string[] gamesWithSpaces = new string[5];
    for (int i = 0; i < currentVideoGames.Length;
    i++)
    { if (currentVideoGames[i].Contains(" "))
    gamesWithSpaces[i] = currentVideoGames[i];
    }
    Array.Sort(gamesWithSpaces);
    foreach (string s in gamesWithSpaces)
    { if( s != null)
    Console.WriteLine("Item: {0}", s);
    }}
}
```

# Relatii intre tipurile interogarilor LINQ

```
List<string> names =  
    new List<string>{"John", "Rick", "Maggie", "Mary"};  
  
IEnumerable<string> nameQuery = from name in names  
                                where name[0] == 'M'  
                                select name;  
  
foreach (string str in nameQuery)  
{  
    Console.WriteLine(str);  
}
```

The diagram illustrates the relationships between LINQ types in the provided code. It features three numbered arrows: Arrow 1 connects the `List<string>` type to the `name` variable in the query. Arrow 2 connects the `name` variable to the `string` type in the `foreach` loop. Arrow 3 connects the `string` type in the `foreach` loop to the `IEnumerable<string>` type.

Interogari care nu transforma sursa de date

1. Argumentul tip al sursei de date determină tipul identificatorului.
2. Tipul obiectului care este selectat determină tipul variabilei de interogare. Aici `name` este un `string`. Prin urmare, variabila de interogare este un `IEnumerable<string>`.
3. Variabila de interogare este iterata în instrucțiunea `foreach`. Deoarece variabila de interogare este o secvență de string-uri, variabila de iterație este, de asemenea, un șir.

# Relatii intre tipurile interogarii LINQ

Interogari care transforma sursa de date: Interogarea preia o secvență de obiecte Customer ca intrare și selectează numai proprietatea Nume din rezultat

1. Argumentul tip al sursei de date determină tipul identicatorului.
2. Instrucțiunea select returnează proprietatea Name în loc de obiectul Customer complet - argumentul de tip al custNameQuery este string, nu Customer.
3. Deoarece custNameQuery este o secvență de string-uri, variabila de iterație a buclei foreach trebuie să fie, de asemenea, un string.

```
Table<Customer> Customers = db.GetTable<Customers>();  
  
IQueryable<string> custNameQuery =  
    from cust in Customers  
    where cust.City == "London"  
    select cust.Name;  
  
foreach (string str in custNameQuery)  
{  
    Console.WriteLine(str);  
}
```

# Relatii intre tipurile interogarii LINQ

```
Table<Customer> Customers = db.GetTable<Customers>();
```

1

```
var namePhoneQuery =  
    from cust in Customers  
    where cust.City == "London"  
    select new { name = cust.Name,  
                phone = cust.Phone };
```

2

3

```
foreach (var item in namePhoneQuery)  
{  
    Console.WriteLine(item);  
}
```

1. Argumentul tip al sursei de date este întotdeauna tipul identificatorului.
2. Deoarece instrucțiunea select produce un tip anonim, variabila de interogare trebuie introdusă implicit folosind var.
3. Deoarece tipul variabilei de interogare este implicit, variabila de iterație din bucla foreach trebuie, de asemenea, să fie implicită.

# Relatii intre tipurile interogarii LINQ

```
var Customers = db.GetTable<Customers>();  
var custQuery = from cust in Customers  
                where cust.City == "London"  
                select cust;  
foreach (var item in custQuery)  
{  
    Console.WriteLine(item);  
}
```

Diagram illustrating the relationships between LINQ types in the provided code:

- 1. `Customers` (implicit `var`) is the source of the query.
- 2. `cust` (implicit `var`) is the element being queried from `Customers`.
- 3. `custQuery` (implicit `var`) is the result of the query, which is then iterated over in the `foreach` loop.

Similar cu exemplul 2, dar folosim variabile cu tip implicit - var

# Executia amanata

- Interogările LINQ nu sunt evaluate decat in momentul in care se itereaza secventa
- Beneficiu – putem utiliza o interogare LINQ de mai multe ori pentru acelasi container si putem fi siguri ca vom obtine rezultate actualizate

```
static void QueryOverInts()
{
    int[] numbers = { 10, 20, 30, 40, 1, 2, 3, 8 };

    var subset = from i in numbers where i < 10 select i;
    // LINQ se evalueaza aici!
    foreach (var i in subset)
        Console.WriteLine("{0} < 10", i);
    Console.WriteLine();
    // Modificam datele
    numbers[0] = 4;
    // Se reevalueaza!
    foreach (var j in subset)
        Console.WriteLine("{0} < 10", j);
    Console.WriteLine();
}
```



# Executia Imediata

- Cand e nevoie sa evaluam o expresie LINQ fara a itera colectia putem utiliza metode extinse ale clasei Enumerable `ToArray<T>()`, `ToDictionary<TSource,TKey>()`, and `ToList<T>()`.
- Aceste metode vor face ca interogarea LINQ sa se execute in momentul apelarii metodei pentru a obtine datele

```
static void ImmediateExecution()
```

```
{
```

```
int[] numbers = { 10, 20, 30, 40, 1, 2, 3, 8 };
```

```
// obtinem datele IMEDIAT ca int[].
```

```
int[] subsetAsIntArray = (from i in numbers where i < 10 select i).ToArray<int>();
```

```
// obtinem datele IMEDIAT ca List<int>.
```

```
List<int> subsetAsListOfInts = (from i in numbers where i < 10 select i).ToList<int>();
```

```
}
```



# Interogari cu agregari

```
var subset = from i in numbers where i < 10  
select i;
```

```
int NumCount = subset.Count();
```

Interogari care realizeaza agregari se executa fara o instructiune foreach pentru ca interogarea in sine foloseste un foreach pentru a returna rezultatul

# Query Syntax vs. Method Syntax

```
int[] numbers = { 5, 10, 8,  
3, 6, 12};
```

//Query syntax:

```
IEnumerable<int> numQuery1 = from  
num in numbers
```

```
where num % 2 == 0
```

```
orderby num select num;
```

//Method syntax:

```
IEnumerable<int> numQuery2 =  
numbers.Where(num => num % 2 ==  
0).OrderBy(n => n);
```

```
foreach (int i in numQuery1)  
{ Console.Write(i + " "); }
```

```
Console.WriteLine(System.Environment.NewLine);
```

```
foreach (int i in numQuery2)  
{ Console.Write(i + " "); }
```