.NET and C#

LINQ

Agenda

- Predefinerede delegates til at understøtte lambdas:
  - Action
  - Predicate
  - Func
- Language Integrated Query (LinQ)
  - Motivation
  - Eksempler
  - To slags syntax
  - Converting IEnumerable results
  - Deferred execution
  - Projecting to new type
  - Aggregation operations
  - SQL examples : join, where, orderby...

Prædefinerede delegates

- Der er nogle prædefinerede delegates i Frameworket, som kan være nyttige at bruge:
  - Action (til at udføre noget kode)
  - Predicate (til at filtrere objekter)
  - Func (bliver ikke gennemgået her)

Predicate

- Predicate delegate er defineret som:

```
public delegate bool Predicate<in T>( T obj )
```

- Dvs. den skal altid returnere en bool, men kan arbejde på alle typer objecter
- Eksempel : List<T>.FindAll(Predicate<T>)

(se næste slides for kode eksempler)

Predicate – eksempel med findAll

```
var data = new List<int> { 1, -2, 3, 0, 2, -1 };

var predicate = new Predicate<int>(isPositive);
var filtered = data.FindAll(predicate);

Console.WriteLine(string.Join(",", filtered));

bool isPositive(int val)
{
    return val > 0;
}

// Hvad udskriver denne kode?
```

Predicate – eksempel med findAll (2)

Som vi så sidste gang, kan det også skrives mere elegant med Lambda udtryk:

var filtered = data.FindAll(`i => i > 0`);

Her er input, i, af typen int, fordi vores liste jo er ints og så skal vi returnere en bool, fordi det er returtype på predicatet.

Vi kan også arbejde på en liste af Personer eller hvilket som helst andet objekt.
F.eks. Kunne en filtrering af Personer på alder se sådan ud – igen skal vi returnere en bool:

var pensionists = persons.FindAll(`p => p.age >= 67`);

Action – udfører en void metode

- The Action delegate er erklæret som:

```
public delegate void Action<in T>( T obj )
```

```
public static void Main() {
 // create a three element array of integers
   int[] intArray = new int[] {2, 3, 4};
    // set a delegate for the ShowSquares method
    Action<int> action = new Action<int>(ShowSquares);  //tager en int som parameter
      Array.ForEach(intArray, action);
}
private static void ShowSquares(int val) { Console.WriteLine("{0:d} squared = {1:d}",
    val, val*val); }
```

Action med mere end 1 parameter

- Frameworket definerer:
  - Action(T)
  - Action(T1, T2)
  - Action(T1, T2, T3)
  - ...
  - Action(T1, T2 ,T3 , … , T9)
- i.e.:

```
public delegate void Action<in T1, in T2, in T3>( T1
   arg1, T2 arg2, T3 arg3 )
```

Find-metoder

- Find
- FindAll
- FindIndex
- FindLast
- FindLastIndex
- Tager alle et Predicate-argument
- De er IKKE en del af LinQ
- Vi tager dem med alligevel som sammenligning

Lille opgave

- Lav opgave 5.1

LINQ

SQL “agtig” syntax til data strukturer

LinQ

- LinQ udtales "link"
- *Quote from Wikipedia:*

  Language Integrated Query, LinQ, is a Microsoft .NET Framework component that adds native data querying capabilities to .NET languages using a syntax reminiscent of SQL
- LinQ kan også ændre data.

LinQ – kan bruges på alle slags data

- **LinQ to Objects: arrays and collections**
- LinQ to XML: XML documents
- LinQ to DataSet: ADO.NET datasets
- LinQ to SQL: relational databases
- **LinQ to Entities: ADO.NET Entity Framework (also rel. databases)**
- Parallel LinQ: parallel execution of a LinQ query

Motivation for at bruge LinQ.

- At gå fra SQL, loops og til LinQ skulle gerne gøre det ensartet og nemmere at søge/filtrere og ændre data uanset om det er fra en database eller data i f.eks. en List.
- *Citat fra stackoverflow.com:*

  "LinQ is one of the greatest improvements to .NET since generics and it saves me tons of time, and lines of code."

Eksempel – LinQ på en collection

```
public class Person
{
  public string Name { get; set; }
  public int Rating { get; set; }
  public bool Approved { get; set; }
}

List<Person> persons = new List<Person>{
new Person { Name = "Michael", Rating = 8, Approved = true },
new Person { Name = "Susan", Rating = 9, Approved = true },
new Person { Name = "Ben", Rating = 5, Approved = false },
new Person { Name = "Camilla", Rating = 7, Approved = true } };
```

To slags syntax (nr 1 her) er mulig – de gør det samme

- ***1. Query methods:***

```
var s = persons.Where(p => p.Approved).
            OrderBy(p => p.Rating).
            Select(p => p.Name);
```

- s har så typen IEnumerable<string>.
- s kan så f.eks. løbes igennem med en foreach løkke.
- (brug af lambda og method chaining med .)

To slags syntax kan bruges (nr 2 her)

- ***2. Query expression:***

```
var s = from p in persons
where p.Approved
orderby p.Rating
select p.Name;
```

- s vil indeholde præcis de samme data som på sidste slide.

(mere SQL agtig syntax)

Konvertering af IEnumerable resultater

- Resultatet fra en LinQ query er en IEnumerable
- IEnumerable kan gennemløbes med en iterator – ligesom i Java.
- IEnumerable kan også f.eks. konverteres til en liste eller et array, hvis man vil:

```
List<String> list = s.ToList<string>();
string[] array = s.ToArray<string>();
```

LinQ - Deferred execution

```
int[] nums={20, 1, 2, 30, 3, 40, 8, 50};
var small_numbers =from i in nums where i<10 select i;
// vil så udskrive 1 2 3 8:
foreach(int n in small_numbers) Console.Write(n+" ");
Console.WriteLine();

nums[0]=7;

// vil så udskrive 7 1 2 3 8:
foreach(int n in small_numbers) Console.Write(n+" ");
Console.WriteLine();
```

Deferred execution

- En query defineret som:

```
var small_numbers =
   from i in nums
   where i<10
    select i;
```

- Ovenstående definerer query, **men udfører den ikke**.
- query bliver udført, når vi skal bruge resultatet – f.eks. når vi laver en foreach eller bruger ToList<T>() på resultatet.

Flere query expression eksempler

```
string[] colors = { "Light Red", "Green", "Yellow", "Dark
 Red", "Red", "Purple"};

var EReds = from c in colors
            where c.Contains("Red")
            select c;
string[] reds = EReds.ToArray();

for (int i = 0; i < EReds.Length; i++)
  Console.WriteLine(EReds[i]);

// query expression og ToArray kan kombineres
reds = (from c in colors
        where c.Contains("Red") select c).ToArray();
```

Lille opgave

Lav opgave 5.2

Anonyme objekter og projection til ny type i LinQ

Anonyme objekter

```
class Program {
    static void Main(string[] args) {
        Console.WriteLine("Hello world");

        int a = 4;
        string s = "text";

        var obj = new { number = a, name = s };

        Console.WriteLine("obj = " + obj);
        Console.WriteLine("obj.number = " + obj.number);
        Console.WriteLine($"obj : name={obj.name}, number={obj.number}");


        Console.ReadLine();
    }
}
```

LINQ projecting to new type – anonymt objekt

```
var s = from p in persons
        where p.Approved
        orderby p.Rating
        select new { p.Name, p.Rating };

foreach (var o in s)
  Console.WriteLine(o);
```

LinQ eksempler og operatorer

Aggregation operations – ”statistik”

```
var min = (from p in persons
           select p.Rating).Min();

var max = (from p in persons
           select p.Rating).Max();

var sum = (from p in persons
           select p.Rating).Sum();

var avg = (from p in persons
           select p.Rating).Average();
```

Count

```
var n = (from c in colors
         where c.Contains("Red")
         select c).Count();


var m = (from c in colors
         where c.Length>5
         select c).Count();
```

Distinct

```
var dr = (from p in persons
          select p.Rating).Distinct();
```

- Finder alle *distinct* (dvs. unikke) Ratings i persons

Orderby (default er ascending)

```
var s = from p in persons
where p.Approved
orderby p.Rating
select p.Name;
```

```
var s = from p in persons
where p.Approved
orderby p.Rating descending
select p.Name;
```

join

- Finder alle stores som ligger i samme postnummer som adressen (nyttigt til en webshop)

```
var res =
  from store in stores
  join address in addresses
  on store.Zip equals address.Zip
  select new { StoreName = store.Name, StreetName =
               address.StreetName, StreetNumber =
               address.StreetNumber
             };
```

Join – med method syntax

```
var res3 =
  stores.Join(addresses, s => s.Zip, a => a.Zip,
              (store, address) =>
              new
              {
                 StoreName = store.Name,
                 StreetName = address.StreetName,
                 StreenNumber = address.StreetNumber
              });
```

join med where

```
var res2 =
 from store2 in stores
 from address2 in addresses
 where store2.Zip == address2.Zip
 select new { StoreName = store2.Name,
              StreetName = address2.StreetName,
              StreetNumber = address2.StreetNumber
             };
```

Group by

```
var queryLastNames = from student in students
                     group student by student.LastName into newGroup
                     orderby newGroup.Key
                     select newGroup;

foreach (var nameGroup in queryLastNames) {
    Console.WriteLine($"Key: {nameGroup.Key}");
    foreach (var student in nameGroup) {
        Console.WriteLine($"\t{student.LastName}, {student.FirstName}");
    }
}
```

Bemærk, man får også adgang til de grupperede data, ikke som SQL

https://stackoverflow.com/questions/7285714/linq-with-groupby-and-count
https://docs.microsoft.com/en-us/dotnet/csharp/linq/group-query-results

let

- Med let kan du definere en ny variabel

```
from p in persons
let name=p.FirstName+" "+p.Lastname
select new { Name=name, Phone=p.Phone};
```

Let eksempel 2

```
var query =
  from c in db.Customers
  let totalSpend = c.Purchases.Sum(p => p.Price)
  where totalSpend > 1000
  from p in c.Purchases
  select
    new { p.Description,
          p.Price,
          c.Address.State };
```

Fordele ved LinQ

- Nogle udvalgte fordele:
  - Man kan spare mange linjer kode i når man bruger LinQ til at selecte og sortere i data.
  - Samme syntax til data – uanset om det er i lister eller fra en database f.eks.
  - Syntax checking på compile time, hvorimod standard SQL syntax først bliver tjekket på runtime normalt, da det er i ” ” som en string
  - Intellisence virker i Visual Studio når man bygger query expressions.

Query method syntax eksempler

(En anden måde at gøre det samme på)

Query method syntax

- Query methods syntax er et alternativ til query expressions:

```
var s = persons.Where(p => p.Approved)
              .OrderBy(p => p.Rating)
              .Select(p => p.Name);
```

- Metoderne er extension methods til collections

```
var res = persons.Select(p => p).
                  Where(p => p.Rating > 5).
                  OrderBy(p => p.Rating).
                  ThenBy(p => p.Name);
```

- Hvad er resultatet?
- OrderByDescending, ThenByDescending, hvis det ikke skal være ascending

GroupBy eksempel

```
// { Name = "Michael", Rating = 8, Approved = true },
// { Name = "Susan", Rating = 9, Approved = true },
// { Name = "Ben", Rating = 5, Approved = false },
// { Name = "Jane", Rating = 9, Approved = true },
// { Name = "Bob", Rating = 8, Approved = true },
// { Name = "Camilla", Rating = 7, Approved = true }

var res = persons.GroupBy(p => p.Rating, p => p);

foreach(var group in res)
{
  Console.WriteLine("Rating: "+group.Key);

  foreach(Person item in group)
    Console.WriteLine("  - "+item);
}
```

Hvilken syntax skal man bruge?

- Det bliver diskuteret på mange udvikler blogs på Internettet☺
- Det vigtigste er at **være konsekvent og bruge den samme måde i hele koden** synes jeg – ellers bliver det forvirrende, hvis man mixer de to måder
- Umiddelbart er query expression syntax den nemmeste, den minder om SQL, men i det lange løb tror jeg, den forvirrer mig.

OPGAVER