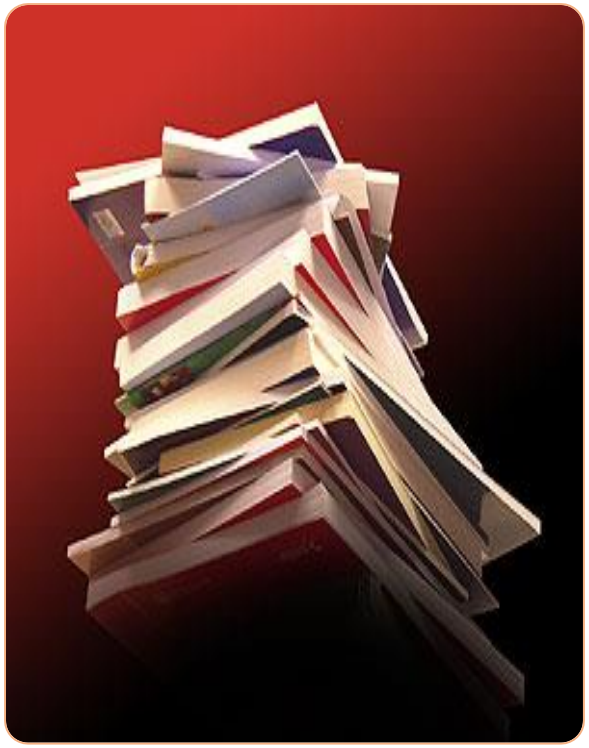
LINQ in C#

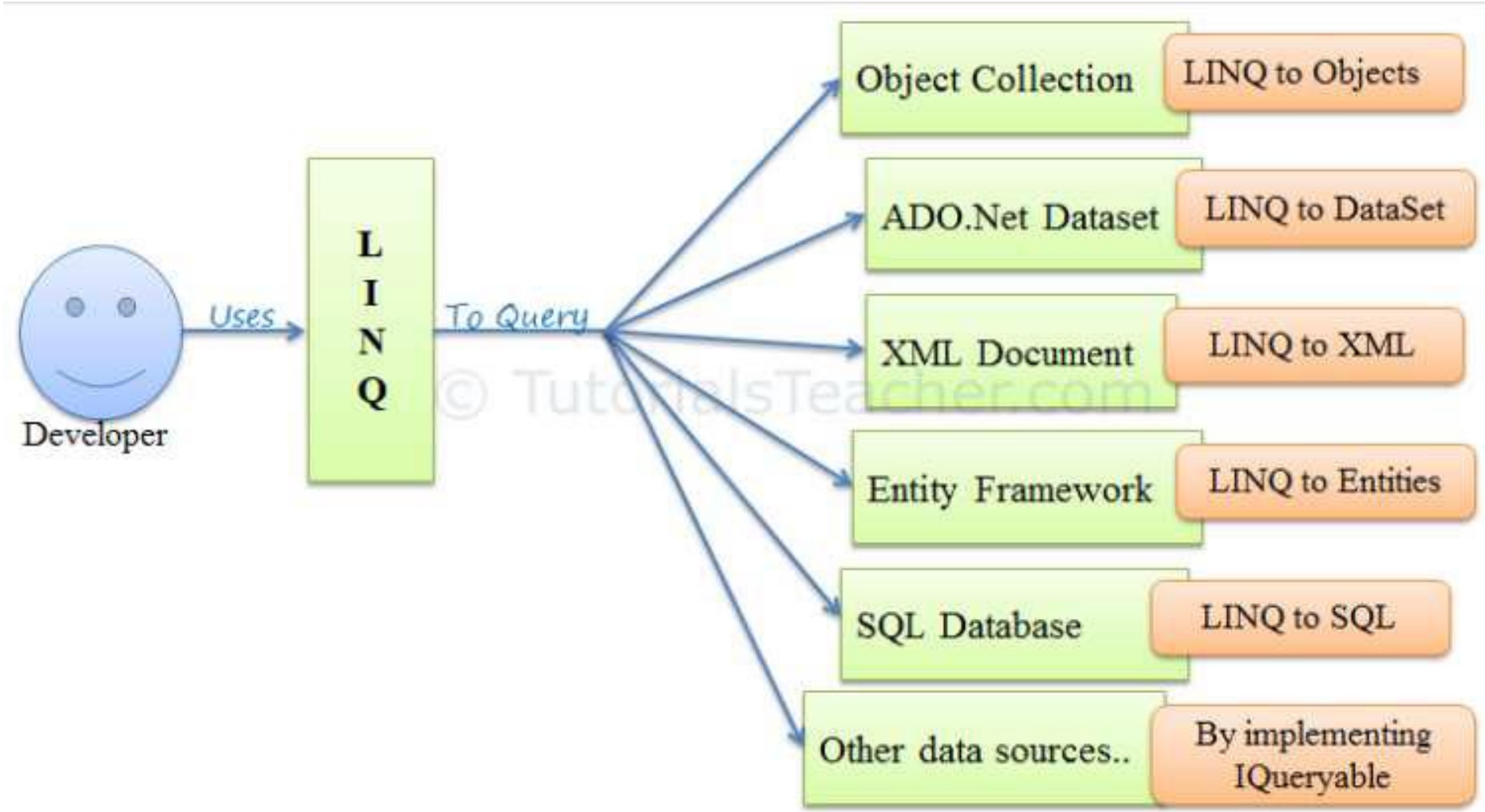
Inhoud

1. Wat is LINQ?
2. Voordelen van LINQ
3. LINQ Operators
   - Filtering Operators
   - Grouping Operators
   - Concatenation
   - Sorting Operators
   - Join Operators
   - Equality
   - Projection Operations
   - Aggregation
   - Quantifier Operations
   - Partition Operations
   - Generation Operations
   - Set Operations
   - Conversions
   - Element Operators



Wat is LINQ?

- Language Integrated Query (LINQ)
- LINQ is a query syntax dat kan gebruikt worden om gegevens te lezen en bewaren van **verschillende types van data sources:**
  - Object Collection
  - SQL server database
  - XML
  - web service
  - …


Developer
Uses
L
I
N
Q
To Query
Object Collection
LINQ to Objects
ADO.Net Dataset
LINQ to DataSet
XML Document
LINQ to XML
Entity Framework
LINQ to Entities
SQL Database
LINQ to SQL
Other data sources..
By implementing IQueryable

LINQ Usage

Voordelen van LINQ

- Syntax zeer compact (Query syntax vs Method syntax)
- Gemakkelijk te debuggen
- Extensible: uitbreidbaar, mogelijk LINQ op nieuw soorten datasources te gebruiken.
- Gemakkelijk om verschillende datasources te combineren (joining) in één enkele LINQ query
- Gemakkelijk om transformatie toe te passen (bv. transformatie van SQL data naar XML data.)

LINQ Operatoren

1. Toepassingen van Linq
2. Restriction/Filtering operators
3. Projection operators
4. Aggregate operators
5. Conversions
6. Element Operators
7. Generators
8. Grouping Operators
9. Join Operators
10. Sorting/ordering Operators
11. Partition Operations
12. Quantifier Operations
13. Sequence operations
14. Set Operations
15. Query Excecution (deferred vs immediate)

Toepassingen van Linq

- **Linq to objects**

  Linq queries op collections/arrays van objects

- **Linq to XML**

  Queries on XML data en XML documents

- **Linq to DataSet**

  Toepassen van Linq queries op ADO.NET DataSet objecten

- **Linq to Entities**

  Linq queries voor ADO.Net Entity Framework API

- **Parallel Linq (PLINQ)**

  Parallelle verwerking van data die teruggegeven door een Linq query

Linq to Objects

Linq to Objects

- Linq queries op collections/arrays van objects
- Assembly: System.Core.dll
- Gebruik namespace: **using System.Linq**

Voorbeeld:

```
public class Product {
        public int ProductID { get; set; }
        public string ProductName { get; set; }
        public string Category { get; set; }
        public decimal UnitPrice { get; set; }
        public int UnitsInStock { get; set; }
         public override string ToString() => $"ProductID={ProductID} ProductName={ProductName} Category={Category}
UnitPrice={UnitPrice:C2} UnitsInStock={UnitsInStock}";    }

public static class Products{

public static List<Product> ProductList { get;  } =  new List<Product> {

new Product { ProductID = 1, ProductName = "Chai",Category = "Beverages", UnitPrice = 18.0000M, UnitsInStock = 39 },

new Product { ProductID = 2, ProductName = "Chang",Category = "Beverages", UnitPrice = 19.0000M, UnitsInStock = 17 },

new Product { ProductID=3, ProductName ="Aniseed Syrup",Category="Condiments",UnitPrice=10.0000M, UnitsInStock=13 },

new Product { ProductID = 4, ProductName="Chef Anton's Cajun Seasoning",Category= "Condiments", UnitPrice = 22.0000M,
UnitsInStock = 53 },

new Product { ProductID = 5, ProductName = "Chef Anton's Gumbo Mix", Category = "Condiments", UnitPrice = 21.3500M,
UnitsInStock = 0 }};
```

Linq to Objects

Voorbeeld (vervolg):

```
var categories =
from p in products
group p by p.Category into g
select (Category: g.Key, MostExpensivePrice: g.Max(p => p.UnitPrice));


foreach (var c in categories)
{
    Console.WriteLine($"Category: {c.Category} Most expensive product:
    {c.MostExpensivePrice}");
}
```

Linq to XML

Linq to XML

- Queries on XML data en XML documents
- Assembly: System.Xml.Linq.dll
- Gebruik namespace: **using System.Xml.Linq;**

Voorbeeld:

```
public static class Customers{

       public static List<Customer> CustomerList { get; } =

           (from e in XDocument.Parse(InputValues.CustomersXml).Root.Elements("customer")

            select new Customer{

                CustomerID = (string)e.Element("id"),  CompanyName = (string)e.Element("name"),

                Address = (string)e.Element("address"),

                City = (string)e.Element("city"), Region = (string)e.Element("region"),

                PostalCode = (string)e.Element("postalcode"),

                Country = (string)e.Element("country"), Phone = (string)e.Element("phone"),

                Orders = (

                   from o in e.Elements("orders").Elements("order")

                   select new Order{

                       OrderID = (int)o.Element("id"), OrderDate = (DateTime)o.Element("orderdate"),

                       Total = (decimal)o.Element("total")

                   }).ToArray()

           }).ToList();    }
```

Voorbeeld (vervolg):

```
List<Product> products = GetProductList();


Product product12 =
   (from p in products
   where p.ProductID == 12
   select p).First();

   Console.WriteLine(product12);
```

Linq - Restriction/Filtering operators

Where clause

Voorbeeld:

```
List<Product> products = GetProductList();

var expensiveInStockProducts =
from prod in products
where prod.UnitsInStock > 0 && prod.UnitPrice > 3.00M
select prod;

Console.WriteLine("In-stock products that cost more than 3.00:");
foreach (var product in expensiveInStockProducts)
{
    Console.WriteLine($"{product.ProductName} is in stock and
    costs more than 3.00.");
}
```

Linq – Projection operators

Select clause

Voorbeeld 1:

```
int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };

var numsPlusOne = from n in numbers
                  select n + 1;

Console.WriteLine("Numbers + 1:");
            foreach (var i in numsPlusOne)
{
                Console.WriteLine(i);
}
```

Linq – Projection operators (vervolg)

Select clause

Voorbeeld 2:

```
List<Product> products = GetProductList();

var productNames = from p in products
                   select p.ProductName;

Console.WriteLine("Product Names:");
foreach (var productName in productNames)
{
     Console.WriteLine(productName);
}
```

Linq – Projection operators (vervolg 2)

Select clause

Voorbeeld 3:

```
string[] words = { "aPPLE", "BlUeBeRrY", "cHeRry" };

var upperLowerWords = from w in words
select new{ Upper= w.ToUpper(),Lower = w.ToLower() };

foreach (var ul in upperLowerWords)
{
   Console.WriteLine($"Uppercase: {ul.Upper},
   Lowercase: {ul.Lower}");
}
```

Linq - Aggregate operators

1. Count
2. Sum
3. Min
4. Max
5. Average
6. Aggregate

Linq - Aggregate operators – 1. Count

Count Operator

Voorbeelden:

```
int[] factorsOf300 = { 2, 2, 3, 5, 5 };
int uniqueFactors =
factorsOf300.Distinct().Count();
Console.WriteLine($"There are {uniqueFactors}
unique factors of 300.");

int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
int oddNumbers = numbers.Count(n => n % 2 == 1);
Console.WriteLine("There are {0} odd numbers in
the list.", oddNumbers);
```

Linq - Aggregate operators – 2. Sum

Sum Operator

Voorbeelden:

```
int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
double numSum = numbers.Sum();
Console.WriteLine($"The sum of the numbers is
{numSum}");

string[] words = { "cherry", "apple", "blueberry" };
double totalChars = words.Sum(w => w.Length);
Console.WriteLine($"There are a total of {totalChars}
characters in these words.");
```

Linq - Aggregate operators – 3. Min

Min Operator

Voorbeelden:

```
int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
int minNum = numbers.Min();
Console.WriteLine($"The minimum number is {minNum}");



string[] words = { "cherry", "apple", "blueberry" };
int shortestWord = words.Min(w => w.Length);
Console.WriteLine($"The shortest word is
{shortestWord} characters long.");
```

Linq - Aggregate operators – 3. Max

Max Operator

Voorbeelden:

```
int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
int maxNum = numbers.Max();

string[] words = { "cherry", "apple", "blueberry" };
int longestLength = words.Max(w => w.Length);
Console.WriteLine($"The longest word is
{longestLength} characters long.");
```

Linq - Aggregate operators – 3. Average

Average Operator

Voorbeelden:

```
int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
double averageNum = numbers.Average();
Console.WriteLine($"The average number is
{averageNum}.");

List<Product> products = GetProductList();
var categories = from p in products
                 group p by p.Category into g
select (Category: g.Key, AveragePrice: g.Average(p =>
p.UnitPrice));
foreach (var c in categories){
   Console.WriteLine($"Category: {c.Category},
   Average price: {c.AveragePrice}");}
```

Linq - Aggregate operators – 4. Aggregate

Aggregate Operator

Voorbeelden:

```
double[] doubles = { 1.7, 2.3, 1.9, 4.1, 2.9 };
double product = doubles.Aggregate((runningProduct,
nextFactor) => runningProduct * nextFactor);
Console.WriteLine($"Total product of all numbers:
{product}");
double startBalance = 100.0;
int[] attemptedWithdrawals = { 20, 10, 40, 50, 10,
70, 30 };
double endBalance =
attemptedWithdrawals.Aggregate(startBalance,
(balance, nextWithdrawal) => ((nextWithdrawal <=
balance) ?(balance - nextWithdrawal) : balance));
Console.WriteLine($"Ending balance: {endBalance}");
```

Linq - Conversions

1. ToArray
2. ToList
3. ToDictionary
4. OfType<T>

Linq – conversions – 1. ToArray

ToArray

Voorbeeld:

```
double[] doubles = { 1.7, 2.3, 1.9, 4.1, 2.9 };
var sortedDoubles = from d in doubles
                    orderby d descending
                    select d;
var doublesArray = sortedDoubles.ToArray();
```

Linq – conversions – 2. ToList

ToList

Voorbeeld:

```
 string[] words = { "cherry", "apple", "blueberry" };

var sortedWords = from w in words
                  orderby w
                  select w;
var wordList = sortedWords.ToList();
```

Linq – conversions – 3. ToDictionary

ToDictionary

Voorbeeld:

```
var scoreRecords = new[] {
new {Name = "Alice", Score = 50},
new {Name = "Bob"  , Score = 40},
new {Name = "Cathy", Score = 45}};

var scoreRecordsDict =
scoreRecords.ToDictionary(sr => sr.Name);

Console.WriteLine("Bob's score: {0}",
scoreRecordsDict["Bob"]);
```

Linq – conversions – 4. OfType<T>

OfType<T>

Voorbeeld:

```
object[] numbers = { null, 1.0, "two", 3, "four", 5,
"six", 7.0 };
var doubles = numbers.OfType<double>();
Console.WriteLine("Numbers stored as doubles:");
foreach (var d in doubles)
{
     Console.WriteLine(d);
}
```

Linq – Element Operations

1. First
2. FirstOrDefault
3. ElementAt

Linq - Element operations – 1. First

First

Voorbeelden:

```
List<Product> products = GetProductList();
Product product12 = (from p in products
                     where p.ProductID == 12
                     select p).First();
Console.WriteLine(product12);

string[] strings = { "zero", "one", "two", "three",
"four", "five", "six", "seven", "eight", "nine" };
string startsWithO = strings.First(s => s[0] =='o');
Console.WriteLine($"A string starting with 'o':
{startsWithO}");
```

Linq - Element operations – 2. FirstOrDefault

FirstOrDefault

Voorbeelden:

```
int[] numbers = { };
int firstNumOrDefault = numbers.FirstOrDefault();
Console.WriteLine(firstNumOrDefault);

List<Product> products = GetProductList();
Product product789 = products.FirstOrDefault(p =>
p.ProductID == 789);
Console.WriteLine($"Product 789 exists:{product789 !=
null}");
```

Linq - Element operations – 3. ElementAt

ElementAt

Voorbeeld:

```
int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
int fourthLowNum = (from n in numbers
                    where n > 5
                    select n)
                    .ElementAt(1);
// second number is index 1 because sequences use 0-
based indexing

Console.WriteLine($"Second number > 5:
{fourthLowNum}");
```

Linq – Generators

1. **Enumerable.Range**
2. **Enumerable.Repeat**

Linq – Generators – 1. Enumerable.Range

Enumerable.Range

Voorbeeld:

```
var numbers = from n in Enumerable.Range(100, 50)
select (Number: n, OddEven: n % 2 == 1 ? "odd" :
"even");

foreach (var n in numbers)
{
   Console.WriteLine("The number {0} is {1}.",
   n.Number, n.OddEven);
}
```

Linq – Generators – 2. Enumerable.Repeat

Enumerable.Repeat

Voorbeeld:

```
var numbers = Enumerable.Repeat(7, 10);

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

Linq – Grouping operators

1. GroupBy
2. GroupBy x by y into z

Linq - grouping operators – 1. groupby

GroupBy

Voorbeeld:

```
class Pet {
      public string Name { get; set; }
      public double Age { get; set; }
}
List<Pet> petsList = new List<Pet>{
new Pet { Name="Barley", Age=8.3 },
new Pet { Name="Boots", Age=4.9 },
new Pet { Name="Whiskers", Age=1.5 },
new Pet { Name="Daisy", Age=4.3 } };
var query = petsList.GroupBy( pet => Math.Floor(pet.Age),
pet => pet.Age, (baseAge, ages) => new { Key = baseAge,
Count = ages.Count(), Min = ages.Min(), Max = ages.Max()
});
```

Linq - grouping operators – 2. GroupBy x by y into z

Voorbeeld:

```
int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
var numberGroups = from n in numbers
   group n by n % 5 into g
   select (Remainder: g.Key, Numbers: g);

foreach (var g in numberGroups){
     Console.WriteLine($"Numbers with a
remainder of {g.Remainder} when divided by 5:");
     foreach (var n in g.Numbers){
          Console.WriteLine(n);
      }
}
```

Linq - joins operators

Where clause

Voorbeeld:

```
List<Product> products = GetProductList();

var expensiveInStockProducts =
from prod in products
where prod.UnitsInStock > 0 && prod.UnitPrice > 3.00M
select prod;

Console.WriteLine("In-stock products that cost more than 3.00:");
foreach (var product in expensiveInStockProducts)
{
    Console.WriteLine($"{product.ProductName} is in stock and
    costs more than 3.00.");
}
```

Linq – Join operator

1. **Inner join**
2. **Group join**

Linq – Join operator – inner join

Join

Voorbeeld:

```
 string[] categories = {
                    "Beverages",
                    "Condiments",
                    "Vegetables",
                    "Dairy Products",
                    "Seafood"};

List<Product> products = GetProductList();
var q = from c in categories
      join p in products on c equals p.Category
       select (Category: c, p.ProductName);
```

Linq – Join operator – group-join

Voorbeeld:

```
 string[] categories = {"Beverages","Condiments",
                        "Vegetables","Dairy Products",
                        "Seafood"};
List<Product> products = GetProductList();
var q = from c in categories
      join p in products on
      c equals p.Category into ps
     select (Category: c, Products: ps);

foreach (var v in q){
      Console.WriteLine(v.Category + ":");
      foreach (var p in v.Products){
       Console.WriteLine("    " + p.ProductName);} }
```

Linq – Ordering/sorting operators

1. Orderby …
2. Orderby …descending
3. Orderby…ThenBy
4. Reverse

Linq – sorting operators- 1.Orderby

Voorbeeld:

```
string[] words = { "cherry", "apple", "blueberry" };

var sortedWords = from word in words
                  orderby word.Length
                  select word;

List<Product> products = GetProductList();
var sortedProducts = from prod in products
                     orderby prod.ProductName
                     select prod;
```

Linq – sorting operators-
2. Orderby …descending

Voorbeeld:

```
double[] doubles = { 1.7, 2.3, 1.9, 4.1, 2.9 };

var sortedDoubles = from d in doubles
                    orderby d descending
                    select d;

List<Product> products = GetProductList();
var sortedProducts = from prod in products
           orderby prod.UnitsInStock descending
           select prod;
```

Linq - sorting operators-
3. Orderby …ThenBy

Voorbeelden:

```
string[] digits = { "zero", "one", "two", "three", "four",
"five", "six", "seven", "eight", "nine" };
var sortedDigits = from digit in digits
                   orderby digit.Length, digit
                   select digit;
 // Custom comparer for use with ordering operators
public class CaseInsensitiveComparer : IComparer<string>{
 public int Compare(string x, string y) =>
  string.Compare(x, y, StringComparison.OrdinalIgnoreCase);}

string[] words = { "aPPLE", "AbAcUs", "bRaNcH", "BlUeBeRrY",
"ClOvEr", "cHeRry" };

var sortedWords = words
                .OrderBy(a => a.Length)
                .ThenBy(a => a, new CaseInsensitiveComparer());
```

Linq – sorting operators- 4. Reverse

Voorbeeld:

```
 string[] digits = { "zero", "one", "two", "three",
"four", "five", "six", "seven", "eight", "nine" };

var reversedIDigits = (
                  from digit in digits
                  where digit[1] == 'i'
                  select digit)
                  .Reverse();
 Console.WriteLine("A backwards list of the digits
with a second character of 'i':");
foreach (var d in reversedIDigits){
      Console.WriteLine(d);
}
```

Linq – Partition operators

1. Take
2. Skip
3. TakeWhile
4. SkipWhile

Linq - partition operators- 1.Take

Voorbeeld:

```
 int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
var first3Numbers = numbers.Take(3);
Console.WriteLine("First 3 numbers:");
foreach (var n in first3Numbers){
     Console.WriteLine(n);
}

List<Customer> customers = GetCustomerList();
var first3WAOrders = ( from cust in customers
      from order in cust.Orders
      where cust.Region == "WA"
     select (cust.CustomerID, order.OrderID,
     order.OrderDate)).Take(3);
```

Linq - partition operators- 2. Skip

Voorbeeld:

```
int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
var allButFirst4Numbers = numbers.Skip(4);
Console.WriteLine("All but first 4 numbers:");
foreach (var n in allButFirst4Numbers){
    Console.WriteLine(n);
}
List<Customer> customers = GetCustomerList();
var waOrders = from cust in customers
               from order in cust.Orders
               where cust.Region == "WA"
               select (cust.CustomerID, order.OrderID,
               order.OrderDate);
var allButFirst2Orders = waOrders.Skip(2);
 Console.WriteLine("All but first 2 orders in WA:");
foreach (var order in allButFirst2Orders) {
                  Console.WriteLine(order); }
```

Linq - partition operators- 3. TakeWhile

Voorbeeld:

```
 int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
 Console.WriteLine("First numbers less than 6:");
 var firstNumbersLessThan6 = numbers.TakeWhile(n => n < 6);


int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
 Console.WriteLine("First numbers not less than their
position:");
var firstSmallNumbers =
numbers.TakeWhile((n, index) => n >= index);
```

Linq - partition operators- 4. SkipWhile

Voorbeeld:

```
int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
// In the lambda expression, 'n' is the input parameter that
identifies each element in the collection in succession. It is
is inferred to be of type int because numbers is an int array.
Console.WriteLine("All elements starting from first element
divisible by 3:");
var allButFirst3Numbers = numbers.SkipWhile(n => n % 3 != 0);

int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
Console.WriteLine("All elements starting from first element less
than its position:");
var laterNumbers = numbers.SkipWhile((n, index) => n >= index);
```

Linq – Quantifiers

1. Any
2. All

Linq – Quantifier – 1. Any

Voorbeeld:

```
 string[] words = { "believe", "relief", "receipt",
"field" };
bool iAfterE = words.Any(w => w.Contains("ei"));

List<Product> products = GetProductList();
var productGroups = from p in products
          group p by p.Category into g
          where g.Any(p => p.UnitsInStock == 0)
          select (Category: g.Key, Products: g);

foreach(var group in productGroups){
    Console.WriteLine(group.Category);
    foreach(var product in group.Products){
        Console.WriteLine($"\t{product}");
    }}
```

Linq – Quantifier – 2. All

Voorbeeld:

```
int[] numbers = { 1, 11, 3, 19, 41, 65, 19 };
bool onlyOdd = numbers.All(n => n % 2 == 1);
Console.WriteLine($"The list contains only odd numbers:
{onlyOdd}");

List<Product> products = GetProductList();
var productGroups = from p in products
                                group p by p.Category into g
                                where g.All(p => p.UnitsInStock > 0)
                                select (Category: g.Key, Products: g);

foreach (var group in productGroups){
        Console.WriteLine(group.Category);
        foreach (var product in group.Products){
        Console.WriteLine($"\t{product}");
        }
}
```

Linq – Sequence operations

1. SequenceEqual
2. Concat
3. Zip

Linq – Sequence operations
1 . SequenceEqual

Voorbeeld:

```
var wordsA = new string[]
{ "cherry", "apple", "blueberry" };
var wordsB = new string[]
{ "cherry", "apple", "blueberry" };

bool match = wordsA.SequenceEqual(wordsB);

Console.WriteLine($"The sequences match: {match}");
```

Linq – Sequence operations 2 . Concat

Voorbeelden:

```
int[] numbersA = { 0, 2, 4, 5, 6, 8, 9 };
int[] numbersB = { 1, 3, 5, 7, 8 };
var allNumbers = numbersA.Concat(numbersB);
Console.WriteLine("All numbers from both arrays:");
foreach (var n in allNumbers){
       Console.WriteLine(n);}

List<Customer> customers = GetCustomerList();
List<Product> products = GetProductList();
var customerNames = from c in customers
                    select c.CompanyName;
var productNames = from p in products
                    select p.ProductName;
var allNames = customerNames.Concat(productNames);
Console.WriteLine("Customer and product names:");
foreach (var n in allNames) {
       Console.WriteLine(n);}
```

Linq – Sequence operations 3 . Zip

Voorbeelden:

```
int[] vectorA = { 0, 2, 4, 5, 6 };
int[] vectorB = { 1, 3, 5, 7, 8 };

int dotProduct = vectorA.Zip(vectorB, (a, b) => a * b).Sum();

Console.WriteLine($"Dot product: {dotProduct}");
```

Linq – Set operations

1. Distinct
2. Union
3. Intersect
4. Except

Linq – Set operations 1. Distinct

Voorbeelden:

```
 int[] factorsOf300 = { 2, 2, 3, 5, 5 };

var uniqueFactors = factorsOf300.Distinct();



List<Product> products = GetProductList();
var categoryNames =
      (from p in products
      select p.Category).Distinct();
```

Linq – Set operations 2. Union

Voorbeelden:

```
int[] numbersA = { 0, 2, 4, 5, 6, 8, 9 };
int[] numbersB = { 1, 3, 5, 7, 8 };

Console.WriteLine("Unique numbers from both
arrays:");
var uniqueNumbers = numbersA.Union(numbersB);
```

Linq – Set operations 3. Intersect

Voorbeelden:

```
int[] numbersA = { 0, 2, 4, 5, 6, 8, 9 };
int[] numbersB = { 1, 3, 5, 7, 8 };

Console.WriteLine("Common numbers shared by both
arrays:");
var commonNumbers = numbersA.Intersect(numbersB);
```

Linq – Set operations 4. Except

Voorbeelden:

```
int[] numbersA = { 0, 2, 4, 5, 6, 8, 9 };
int[] numbersB = { 1, 3, 5, 7, 8 };

Console.WriteLine("Numbers in first array but not
second array:");


IEnumerable<int> aOnlyNumbers =
numbersA.Except(numbersB);
```

Linq – Query Execution

1. **Deferred**
2. **Immediate (Eager)**

Linq – Query execution - Deferred

Voorbeeld:

```
int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
int i = 0;
var q = from n in numbers
            select ++i;
// Note, the local variable 'i' is not
incremented until each element is evaluated (as a
side-effect):
foreach (var v in q)
{
     Console.WriteLine($"v = {v}, i = {i}");
}
```

Linq – Query execution - Immediate

Voorbeeld:

```
// Methods like ToList(), ToArray() cause the
query to be executed immediately, caching the
results.
int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
int i = 0;
var q = (from n in numbers
         select ++i).ToList();
// The local variable i has already been fully
// incremented before we iterate the results:
foreach (var v in q){
    Console.WriteLine($"v = {v}, i = {i}");
}
```

Lambda Expressions en LINQ

Questions?

Referenties

- Telerik Software Academy
  - https://www.telerikacademy.com/

https://docs.microsoft.com/en-us/dotnet/api/system.linq?view=netcore-3.0