Linq to objects Voorbeelden

Overzicht Linq Operators

[1 Test classen en Collecties 1](#_Toc30925705)

[2 Restrictions 2](#_Toc30925706)

[3 Projections 3](#_Toc30925707)

[4 Aggregate Operators 7](#_Toc30925708)

[5 Conversion 12](#_Toc30925709)

[6 Elementoperations 13](#_Toc30925710)

[7 Generators 14](#_Toc30925711)

[8 Groupings 15](#_Toc30925712)

[9 Joinoperations 17](#_Toc30925713)

[10 Orderings 19](#_Toc30925714)

[11 Partitions 22](#_Toc30925715)

[12 Quantifiers 24](#_Toc30925716)

[13 QueryExecution 26](#_Toc30925717)

[14 Sequence operations 27](#_Toc30925718)

[15 Set operations 29](#_Toc30925719)

[16 Referenties 31](#_Toc30925720)

# Test classen en Collecties



1.Download “Oefeningen-LINQ-START-Classes.zip” van LES5 van moodle

2.Maak een nieuw project en copieer de classen Products, Customers, Orders en InputValues in dit project

3. Maak hierop de volgende oefeningen :

1. Geef de verschillende categoryNamen (geen dubbels) van alle produkten

2. Geef het eerste productNaam en ID uit de category "Seafood"

3. Geef het aantal produkten voor elke categoryNaam

4. Geef voor elke category de produkten die de laagste prijs hebben in deze category

var categories = from p in productsList

group p by p.Category into g

let minPrice = g.Min(p => p.UnitPrice)

select new {Category= g.Key,

CheapestProducts= g.Where(p => p.UnitPrice == minPrice)};

foreach (var c in categories)

{

Console.WriteLine($"Category: {c.Category}");

foreach(var p in c.CheapestProducts)

{

Console.WriteLine($"\tProduct: {p}");

}

}

5. Geef de gemiddelde prijs van de produkten per category

6. Geef de lijst van alle productNamen en hun categoryNaam

<https://www.tutorialsteacher.com/linq/linq-tutorials>

**Online LINQ Test (20 vragen)**

7. Geef de eerste 5 produkten uit de lijst waarvan het aantal unitsInStock =0

8. Sla de produkten over zolang de prijs < 20.00M en geef dan al de rest van de producten terug

9. Geef het eerste element uit de lijst van products waarvan de category = “Beverages”

10. Bestaan er producten (bool) in de lijst van products waarin de tekst “Coffee” in hun productnaam voorkomt?

11. (bool)Hebben alle producten in de lijst van products unitsInStock >0 ?

12. Geef voor elke CustomerID het totaal aantal orders en de som van Totalen van zijn orders

13. Geef de lijst customers die orders hebben geplaatst vóór 01/01/1998

14. Geef de naam van customers die in Mexico wonen en de gemiddelde van Totalen van zijn orders.

15. Geef het eerste order van de customer waarvan customernaam de tekst “snabbköp” bevat

16. Geef de CustomerID, customerName, OrderID van de customers met de orders waarvan het order Total > 500.00M

17. Geef de lijst van Customers in omgekeerde volgorde

18. Geef het totaal aantal orders van alle customers die in Mexico wonen

# Restriction operators

using System;

using System.Collections.Generic;

using System.Linq;

namespace LinqSamples

{

public class Restrictions

{

public List<Product> GetProductList() => Products.ProductList;

public List<Customer> GetCustomerList() => Customers.CustomerList;

public int LowNumbers()

{

#region where-syntax

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

var lowNums = from num in numbers

where num < 5

select num;

Console.WriteLine("Numbers < 5:");

foreach (var x in lowNums)

{

Console.WriteLine(x);

}

#endregion

return 0;

}

public int ProductsOutOfStock()

{

#region where-property

List<Product> products = GetProductList();

var soldOutProducts = from prod in products

where prod.UnitsInStock == 0

select prod;

Console.WriteLine("Sold out products:");

foreach (var product in soldOutProducts)

{

Console.WriteLine($"{product.ProductName} is sold out!");

}

#endregion

return 0;

}

public int ExpensiveProductsInStock()

{

#region where-multiple-properties

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.");

}

#endregion

return 0;

}

public int DisplayCustomerOrders()

{

#region where-drilldown

List<Customer> customers = GetCustomerList();

var waCustomers = from cust in customers

where cust.Region == "WA"

select cust;

Console.WriteLine("Customers from Washington and their orders:");

foreach (var customer in waCustomers)

{

Console.WriteLine($"Customer {customer.CustomerID}: {customer.CompanyName}");

foreach (var order in customer.Orders)

{

Console.WriteLine($" Order {order.OrderID}: {order.OrderDate}");

}

}

#endregion

return 0;

}

public int IndexedWhere()

{

#region where-indexed

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

var shortDigits = digits.Where((digit, index) => digit.Length < index);

Console.WriteLine("Short digits:");

foreach (var d in shortDigits)

{

Console.WriteLine($"The word {d} is shorter than its value.");

}

#endregion

return 0;

}

}

}

# Projections

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

namespace LinqSamples

{

public class Projections

{

public List<Product> GetProductList() => Products.ProductList;

public List<Customer> GetCustomerList() => Customers.CustomerList;

public int SelectSyntax()

{

#region select-syntax

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);

}

return 0;

#endregion

}

public int SelectProperty()

{

#region select-property

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);

}

#endregion

return 0;

}

public int TransformWithSelect()

{

#region select-transform

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

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

var textNums = from n in numbers

select strings[n];

Console.WriteLine("Number strings:");

foreach (var s in textNums)

{

Console.WriteLine(s);

}

#endregion

return 0;

}

public int SelectByCaseAnonymous()

{

#region select-case-anonymous

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}");

}

#endregion

return 0;

}

public int SelectByCaseTuple()

{

#region select-case-tuple

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

var upperLowerWords = from w in words

select (Upper : w.ToUpper(), Lower : w.ToLower());

foreach (var ul in upperLowerWords)

{

Console.WriteLine($"Uppercase: {ul.Upper}, Lowercase: {ul.Lower}");

}

#endregion

return 0;

}

public int SelectAnonymousConstructions()

{

#region select-new-type

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

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

var digitOddEvens = from n in numbers

select new { Digit = strings[n], Even = (n % 2 == 0) };

foreach (var d in digitOddEvens)

{

Console.WriteLine($"The digit {d.Digit} is {(d.Even ? "even" : "odd")}.");

}

#endregion

return 0;

}

public int SelectTupleConstructions()

{

#region select-new-type-tuple

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

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

var digitOddEvens = from n in numbers

select (Digit : strings[n], Even : (n % 2 == 0));

foreach (var d in digitOddEvens)

{

Console.WriteLine($"The digit {d.Digit} is {(d.Even ? "even" : "odd")}.");

}

#endregion

return 0;

}

public int SelectPropertySubset()

{

#region select-subset-properties

List<Product> products = GetProductList();

var productInfos = from p in products

select (p.ProductName, p.Category, Price : p.UnitPrice);

Console.WriteLine("Product Info:");

foreach (var productInfo in productInfos)

{

Console.WriteLine($"{productInfo.ProductName} is in the category {productInfo.Category} and costs {productInfo.Price} per unit.");

}

#endregion

return 0;

}

public int SelectWithIndex()

{

#region select-with-index

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

var numsInPlace = numbers.Select((num, index) => (Num : num, InPlace : (num == index)));

Console.WriteLine("Number: In-place?");

foreach (var n in numsInPlace)

{

Console.WriteLine($"{n.Num}: {n.InPlace}");

}

#endregion

return 0;

}

public int SelectWithWhere()

{

#region select-with-where

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

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

var lowNums = from n in numbers

where n < 5

select digits[n];

Console.WriteLine("Numbers < 5:");

foreach (var num in lowNums)

{

Console.WriteLine(num);

}

#endregion

return 0;

}

public int SelectFromMultipleSequences()

{

#region select-many-syntax

int[] numbersA = { 0, 2, 4, 5, 6, 8, 9 };

int[] numbersB = { 1, 3, 5, 7, 8 };

var pairs = from a in numbersA

from b in numbersB

where a < b

select (a, b);

Console.WriteLine("Pairs where a < b:");

foreach (var pair in pairs)

{

Console.WriteLine($"{pair.a} is less than {pair.b}");

}

#endregion

return 0;

}

public int SelectFromChildSequence()

{

#region select-many-drilldown

List<Customer> customers = GetCustomerList();

var orders = from c in customers

from o in c.Orders

where o.Total < 500.00M

select (c.CustomerID, o.OrderID, o.Total);

foreach(var order in orders)

{

Console.WriteLine($"Customer: {order.CustomerID}, Order: {order.OrderID}, Total value: {order.Total}");

}

#endregion

return 1;

}

public int SelectManyWithWhere()

{

#region select-many-filter

List<Customer> customers = GetCustomerList();

var orders = from c in customers

from o in c.Orders

where o.OrderDate >= new DateTime(1998, 1, 1)

select (c.CustomerID, o.OrderID, o.OrderDate);

foreach (var order in orders)

{

Console.WriteLine($"Customer: {order.CustomerID}, Order: {order.OrderID}, Total date: {order.OrderDate.ToShortDateString()}");

}

#endregion

return 0;

}

public int SelectManyWhereAssignment()

{

#region select-many-assignment

List<Customer> customers = GetCustomerList();

var orders = from c in customers

from o in c.Orders

where o.Total >= 2000.0M

select (c.CustomerID, o.OrderID, o.Total);

foreach (var order in orders)

{

Console.WriteLine($"Customer: {order.CustomerID}, Order: {order.OrderID}, Total value: {order.Total}");

}

#endregion

return 0;

}

public int SelectMultipleWhereClauses()

{

#region multiple-where-clauses

List<Customer> customers = GetCustomerList();

DateTime cutoffDate = new DateTime(1997, 1, 1);

var orders = from c in customers

where c.Region == "WA"

from o in c.Orders

where o.OrderDate >= cutoffDate

select (c.CustomerID, o.OrderID);

foreach (var order in orders)

{

Console.WriteLine($"Customer: {order.CustomerID}, Order: {order.OrderID}");

}

#endregion

return 0;

}

public int IndexedSelectMany()

{

#region indexed-select-many

List<Customer> customers = GetCustomerList();

var customerOrders =

customers.SelectMany(

(cust, custIndex) =>

cust.Orders.Select(o => "Customer #" + (custIndex + 1) +

" has an order with OrderID " + o.OrderID));

foreach (var order in customerOrders)

{

Console.WriteLine(order);

}

#endregion

return 0;

}

}

}

# Aggregate Operators

using System;

using System.Collections.Generic;

using System.Linq;

namespace LinqSamples

{

public class AggregateOperators

{

public List<Product> GetProductList() => Products.ProductList;

public List<Customer> GetCustomerList() => Customers.CustomerList;

public int CountSyntax()

{

#region count-syntax

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

int uniqueFactors = factorsOf300.Distinct().Count();

Console.WriteLine($"There are {uniqueFactors} unique factors of 300.");

#endregion

return 0;

}

public int CountConditional()

{

#region count-conditional

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);

#endregion

return 0;

}

public int NestedCount()

{

#region nested-count

List<Customer> customers = GetCustomerList();

var orderCounts = from c in customers

select (c.CustomerID, OrderCount: c.Orders.Count());

foreach(var customer in orderCounts)

{

Console.WriteLine($"ID: {customer.CustomerID}, count:

{customer.OrderCount}");

}

#endregion

return 0;

}

public int GroupedCount()

{

#region grouped-count

List<Product> products = GetProductList();

var categoryCounts = from p in products

group p by p.Category into g

select (Category: g.Key, ProductCount: g.Count());

foreach(var c in categoryCounts)

{

Console.WriteLine($"Category: {c.Category}: Product count: {c.ProductCount}");

}

#endregion

return 0;

}

public int SumSyntax()

{

#region sum-syntax

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}");

#endregion

return 0;

}

public int SumProjection()

{

#region sum-of-projection

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

double totalChars = words.Sum(w => w.Length);

Console.WriteLine($"There are a total of {totalChars} characters in these words.");

#endregion

return 0;

}

public int SumGrouped()

{

#region grouped-sum

List<Product> products = GetProductList();

var categories = from p in products

group p by p.Category into g

select (Category: g.Key, TotalUnitsInStock: g.Sum(p => p.UnitsInStock));

foreach(var pair in categories)

{

Console.WriteLine($"Category: {pair.Category}, Units in stock: {pair.TotalUnitsInStock}");

}

#endregion

return 0;

}

public int MinSyntax()

{

#region min-syntax

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

int minNum = numbers.Min();

Console.WriteLine($"The minimum number is {minNum}");

#endregion

return 0;

}

public int MinProjection()

{

#region min-projection

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

int shortestWord = words.Min(w => w.Length);

Console.WriteLine($"The shortest word is {shortestWord} characters long.");

#endregion

return 0;

}

public int MinGrouped()

{

#region min-grouped

List<Product> products = GetProductList();

var categories = from p in products

group p by p.Category into g

select (Category: g.Key, CheapestPrice: g.Min(p => p.UnitPrice));

foreach(var c in categories)

{

Console.WriteLine($"Category: {c.Category}, Lowest price: {c.CheapestPrice}");

}

#endregion

return 0;

}

public int MinEachGroup()

{

#region min-each-group

List<Product> products = GetProductList();

var categories = from p in products

group p by p.Category into g

let minPrice = g.Min(p => p.UnitPrice)

select (Category: g.Key, CheapestProducts: g.Where(p => p.UnitPrice == minPrice));

foreach (var c in categories)

{

Console.WriteLine($"Category: {c.Category}");

foreach(var p in c.CheapestProducts)

{

Console.WriteLine($"\tProduct: {p}");

}

}

#endregion

return 0;

}

public int MaxSyntax()

{

#region max-syntax

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

int maxNum = numbers.Max();

Console.WriteLine($"The maximum number is {maxNum}");

#endregion

return 0;

}

public int MaxProjection()

{

#region max-projection

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

int longestLength = words.Max(w => w.Length);

Console.WriteLine($"The longest word is {longestLength} characters long.");

#endregion

return 0;

}

public int MaxGrouped()

{

#region max-grouped

List<Product> products = GetProductList();

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}");

}

#endregion

return 0;

}

public int MaxEachGroup()

{

#region max-each-group

List<Product> products = GetProductList();

var categories = from p in products

group p by p.Category into g

let maxPrice = g.Max(p => p.UnitPrice)

select (Category: g.Key, MostExpensiveProducts: g.Where(p => p.UnitPrice == maxPrice));

foreach (var c in categories)

{

Console.WriteLine($"Category: {c.Category}");

foreach (var p in c.MostExpensiveProducts)

{

Console.WriteLine($"\t{p}");

}

}

#endregion

return 0;

}

public int AverageSyntax()

{

#region average-syntax

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

double averageNum = numbers.Average();

Console.WriteLine($"The average number is {averageNum}.");

#endregion

return 0;

}

public int AverageProjection()

{

#region average-projection

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

double averageLength = words.Average(w => w.Length);

Console.WriteLine($"The average word length is {averageLength} characters.");

#endregion

return 0;

}

public int AverageGrouped()

{

#region average-grouped

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}");

}

#endregion

return 0;

}

public int AggregateSyntax()

{

#region aggregate-syntax

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}");

#endregion

return 0;

}

public int SeededAggregate()

{

#region aggregate-seeded

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}");

#endregion

return 0;

}

}

}

# Conversion

using System;

using System.Linq;

namespace LinqSamples

{

public class Conversions

{

public int ConvertToArray()

{

#region convert-to-array

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();

Console.WriteLine("Every other double from highest to lowest:");

for (int d = 0; d < doublesArray.Length; d += 2)

{

Console.WriteLine(doublesArray[d]);

}

#endregion

return 0;

}

public int ConvertToList()

{

#region convert-to-list

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

var sortedWords = from w in words

orderby w

select w;

var wordList = sortedWords.ToList();

Console.WriteLine("The sorted word list:");

foreach (var w in wordList)

{

Console.WriteLine(w);

}

#endregion

return 0;

}

public int ConvertToDictionary()

{

#region convert-to-dictionary

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"]);

#endregion

return 0;

}

public int ConvertSelectedItems()

{

#region convert-to-type

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);

}

#endregion

return 0;

}

}

}

# Element operations

using System;

using System.Collections.Generic;

using System.Linq;

namespace LinqSamples

{

public class ElementOperations

{

public List<Product> GetProductList() => Products.ProductList;

public List<Customer> GetCustomerList() => Customers.CustomerList;

public int FirstElement()

{

#region first-element

List<Product> products = GetProductList();

Product product12 = (from p in products

where p.ProductID == 12

select p)

.First();

Console.WriteLine(product12);

#endregion

return 0;

}

public int FirstMatchingElement()

{

#region first-matching-element

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}");

#endregion

return 0;

}

public int MaybeFirstElement()

{

#region first-or-default

int[] numbers = { };

int firstNumOrDefault = numbers.FirstOrDefault();

Console.WriteLine(firstNumOrDefault);

#endregion

return 0;

}

public int MaybeFirstMatchingElement()

{

#region first-matching-or-default

List<Product> products = GetProductList();

Product product789 = products.FirstOrDefault(p => p.ProductID == 789);

Console.WriteLine($"Product 789 exists: {product789 != null}");

#endregion

return 0;

}

public int ElementAtPosition()

{

#region element-at

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}");

#endregion

return 0;

}

}

}

# Generators

using System;

using System.Linq;

namespace LinqSamples

{

public class Generators

{

public int RangeOfIntegers()

{

#region generate-range

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);

}

#endregion

return 0;

}

public int RepeatNumber()

{

#region generate-repeat

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

foreach (var n in numbers)

{

Console.WriteLine(n);

}

#endregion

return 0;

}

}

}

# Groupings

using System;

using System.Collections.Generic;

using System.Linq;

namespace LinqSamples

{

public class Groupings

{

public List<Product> GetProductList() => Products.ProductList;

public List<Customer> GetCustomerList() => Customers.CustomerList;

public int GroupingSyntax()

{

#region groupby-syntax

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);

}

}

#endregion

return 0;

}

public int GroupByProperty()

{

#region groupby-property

string[] words = { "blueberry", "chimpanzee", "abacus", "banana", "apple", "cheese" };

var wordGroups = from w in words

group w by w[0] into g

select (FirstLetter: g.Key, Words: g);

foreach (var g in wordGroups)

{

Console.WriteLine("Words that start with the letter '{0}':", g.FirstLetter);

foreach (var w in g.Words)

{

Console.WriteLine(w);

}

}

#endregion

return 0;

}

public int GroupByCategory()

{

#region groupby-category

List<Product> products = GetProductList();

var orderGroups = from p in products

group p by p.Category into g

select (Category: g.Key, Products: g);

foreach(var orderGroup in orderGroups)

{

Console.WriteLine($"Products in {orderGroup.Category} category:");

foreach(var product in orderGroup.Products)

{

Console.WriteLine($"\t{product}");

}

}

#endregion

return 0;

}

public int NestedGrouBy()

{

#region nested-groupby

List<Customer> customers = GetCustomerList();

var customerOrderGroups = from c in customers

select

(

c.CompanyName,

YearGroups: from o in c.Orders

group o by o.OrderDate.Year into yg

select

(

Year: yg.Key,

MonthGroups: from o in yg

group o by o.OrderDate.Month into mg

select (Month: mg.Key, Orders: mg)

)

);

foreach( var ordersByCustomer in customerOrderGroups)

{

Console.WriteLine($"Customer Name: {ordersByCustomer.CompanyName}");

foreach(var ordersByYear in ordersByCustomer.YearGroups)

{

Console.WriteLine($"\tYear: {ordersByYear.Year}");

foreach(var ordersByMonth in ordersByYear.MonthGroups)

{

Console.WriteLine($"\t\tMonth: {ordersByMonth.Month}");

foreach(var order in ordersByMonth.Orders)

{

Console.WriteLine($"\t\t\tOrder: {order}");

}

}

}

}

#endregion

return 0;

}

public int GroupByCustomComparer()

{

#region groupby-custom-comparer

string[] anagrams = { "from ", " salt", " earn ", " last ", " near ", " form " };

var orderGroups = anagrams.GroupBy(w => w.Trim(), new AnagramEqualityComparer());

foreach(var set in orderGroups)

{

// The key would be the first item in the set

foreach(var word in set)

{

Console.WriteLine(word);

}

Console.WriteLine("...");

}

#endregion

return 0;

}

public int NestedGroupByCustom()

{

#region nested-groupby-custom

string[] anagrams = { "from ", " salt", " earn ", " last ", " near ", " form " };

var orderGroups = anagrams.GroupBy(

w => w.Trim(),

a => a.ToUpper(),

new AnagramEqualityComparer()

);

foreach (var set in orderGroups)

{

Console.WriteLine(set.Key);

foreach (var word in set)

{

Console.WriteLine($"\t{word}");

}

}

#endregion

return 0;

}

}

#region anagram-comparer

public class AnagramEqualityComparer : IEqualityComparer<string>

{

public bool Equals(string x, string y) => getCanonicalString(x) == getCanonicalString(y);

public int GetHashCode(string obj) => getCanonicalString(obj).GetHashCode();

private string getCanonicalString(string word)

{

char[] wordChars = word.ToCharArray();

Array.Sort<char>(wordChars);

return new string(wordChars);

}

}

#endregion

}

# Join operations

using System;

using System.Collections.Generic;

using System.Linq;

namespace LinqSamples

{

public class JoinOperations

{

public List<Product> GetProductList() => Products.ProductList;

public List<Customer> GetCustomerList() => Customers.CustomerList;

public int CrossJoinQuery()

{

#region cross-join

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);

foreach (var v in q)

{

Console.WriteLine(v.ProductName + ": " + v.Category);

}

#endregion

return 0;

}

public int GroupJoinQquery()

{

#region group-join

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);

}

}

#endregion

return 0;

}

public int CrossGroupJoin()

{

#region cross-group-join

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

from p in ps

select (Category: c, p.ProductName);

foreach (var v in q)

{

Console.WriteLine(v.ProductName + ": " + v.Category);

}

#endregion

return 0;

}

public int LeftOuterJoin()

{

#region left-outer-join

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

from p in ps.DefaultIfEmpty()

select (Category: c, ProductName: p == null ? "(No products)" : p.ProductName);

foreach (var v in q)

{

Console.WriteLine($"{v.ProductName}: {v.Category}");

}

#endregion

return 0;

}

}

}

# Orderings

using System;

using System.Linq;

using System.Collections.Generic;

namespace LinqSamples

{

public class Orderings

{

public List<Product> GetProductList() => Products.ProductList;

public List<Customer> GetCustomerList() => Customers.CustomerList;

public int OrderbySyntax()

{

#region orderby-syntax

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

var sortedWords = from word in words

orderby word

select word;

Console.WriteLine("The sorted list of words:");

foreach (var w in sortedWords)

{

Console.WriteLine(w);

}

#endregion

return 0;

}

public int OrderbyProperty()

{

#region orderby-property

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

var sortedWords = from word in words

orderby word.Length

select word;

Console.WriteLine("The sorted list of words (by length):");

foreach (var w in sortedWords)

{

Console.WriteLine(w);

}

#endregion

return 0;

}

public int OrderByProducts()

{

#region orderby-user-types

List<Product> products = GetProductList();

var sortedProducts = from prod in products

orderby prod.ProductName

select prod;

foreach (var product in sortedProducts)

{

Console.WriteLine(product);

}

#endregion

return 0;

}

#region custom-comparer

// 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);

}

#endregion

public int OrderByWithCustomComparer()

{

#region orderby-custom-comparer

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

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

foreach(var word in sortedWords)

{

Console.WriteLine(word);

}

#endregion

return 0;

}

public int OrderByDescendingSyntax()

{

#region orderbydescending-syntax

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

var sortedDoubles = from d in doubles

orderby d descending

select d;

Console.WriteLine("The doubles from highest to lowest:");

foreach (var d in sortedDoubles)

{

Console.WriteLine(d);

}

#endregion

return 0;

}

public int OrderProductsDescending()

{

#region orderby-descending-type

List<Product> products = GetProductList();

var sortedProducts = from prod in products

orderby prod.UnitsInStock descending

select prod;

foreach(var product in sortedProducts)

{

Console.WriteLine(product);

}

#endregion

return 0;

}

public int DescendingCustomComparer()

{

#region desc-custom-comparer

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

var sortedWords = words.OrderByDescending(a => a, new CaseInsensitiveComparer());

foreach(var word in sortedWords)

{

Console.WriteLine(word);

}

#endregion

return 0;

}

public int ThenBySyntax()

{

#region thenby-syntax

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

var sortedDigits = from digit in digits

orderby digit.Length, digit

select digit;

Console.WriteLine("Sorted digits:");

foreach (var d in sortedDigits)

{

Console.WriteLine(d);

}

#endregion

return 0;

}

public int ThenByCustom()

{

#region thenby-custom

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

var sortedWords = words

.OrderBy(a => a.Length)

.ThenBy(a => a, new CaseInsensitiveComparer());

foreach(var word in sortedWords)

{

Console.WriteLine(word);

}

#endregion

return 0;

}

public int ThenByDifferentOrdering()

{

#region thenby-ordering

List<Product> products = GetProductList();

var sortedProducts = from prod in products

orderby prod.Category, prod.UnitPrice descending

select prod;

foreach(var product in sortedProducts)

{

Console.WriteLine(product);

}

#endregion

return 0;

}

public int CustomThenByDescending()

{

#region thenby-custom-descending

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

var sortedWords = words

.OrderBy(a => a.Length)

.ThenByDescending(a => a, new CaseInsensitiveComparer());

foreach (var word in sortedWords)

{

Console.WriteLine(word);

}

#endregion

return 0;

}

public int OrderingReversal()

{

#region reverse

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);

}

#endregion

return 0;

}

}

}

# Partitions

using System;

using System.Collections.Generic;

using System.Linq;

namespace LinqSamples

{

public class Partitions

{

public List<Product> GetProductList() => Products.ProductList;

public List<Customer> GetCustomerList() => Customers.CustomerList;

public int TakeSyntax()

{

#region take-syntax

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);

}

#endregion

return 0;

}

public int NestedTake()

{

#region nested-take

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);

Console.WriteLine("First 3 orders in WA:");

foreach (var order in first3WAOrders)

{

Console.WriteLine(order);

}

#endregion

return 0;

}

public int SkipSyntax()

{

#region skip-syntax

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);

}

#endregion

return 0;

}

public int NestedSkip()

{

#region nested-skip

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);

}

#endregion

return 1;

}

public int TakeWhileSyntax()

{

#region takewhile-syntax

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

var firstNumbersLessThan6 = numbers.TakeWhile(n => n < 6);

Console.WriteLine("First numbers less than 6:");

foreach (var num in firstNumbersLessThan6)

{

Console.WriteLine(num);

}

#endregion

return 0;

}

public int IndexedTakeWhile()

{

#region indexed-takewhile

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

var firstSmallNumbers = numbers.TakeWhile((n, index) => n >= index);

Console.WriteLine("First numbers not less than their position:");

foreach (var n in firstSmallNumbers)

{

Console.WriteLine(n);

}

#endregion

return 0;

}

public int SkipWhileSyntax()

{

#region skipwhile-syntax

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.

var allButFirst3Numbers = numbers.SkipWhile(n => n % 3 != 0);

Console.WriteLine("All elements starting from first element divisible by 3:");

foreach (var n in allButFirst3Numbers)

{

Console.WriteLine(n);

}

#endregion

return 0;

}

public int IndexedSkipWhile()

{

#region indexed-skipwhile

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

var laterNumbers = numbers.SkipWhile((n, index) => n >= index);

Console.WriteLine("All elements starting from first element less than its position:");

foreach (var n in laterNumbers)

{

Console.WriteLine(n);

}

#endregion

return 0;

}

}

}

# Quantifiers

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

namespace LinqSamples

{

public class Quantifiers

{

public List<Product> GetProductList() => Products.ProductList;

public List<Customer> GetCustomerList() => Customers.CustomerList;

public int AnyMatchingElements()

{

#region any-matches

string[] words = { "believe", "relief", "receipt", "field" };

bool iAfterE = words.Any(w => w.Contains("ei"));

Console.WriteLine($"There is a word that contains in the list that contains 'ei': {iAfterE}");

#endregion

return 0;

}

public int GroupedAnyMatchedElements()

{

#region any-grouped

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}");

}

}

#endregion

return 0;

}

public int AllMatchedElements()

{

#region all-match

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}");

#endregion

return 0;

}

public int GroupedAllMatchedElements()

{

#region all-grouped

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}");

}

}

#endregion

return 0;

}

}

}

# QueryExecution

using System;

using System.Linq;

namespace LinqSamples

{

public class QueryExecution

{

public int DeferredExecution()

{

#region deferred-execution

// Sequence operators form first-class queries that

// are not executed until you enumerate over them.

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}");

}

#endregion

return 0;

}

public int EagerExecution()

{

#region eager-execution

// Methods like ToList() 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}");

}

#endregion

return 0;

}

public int ReuseQueryDefinition()

{

#region reuse-query

// Deferred execution lets us define a query once

// and then reuse it later after data changes.

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

var lowNumbers = from n in numbers

where n <= 3

select n;

Console.WriteLine("First run numbers <= 3:");

foreach (int n in lowNumbers)

{

Console.WriteLine(n);

}

for (int i = 0; i < 10; i++)

{

numbers[i] = -numbers[i];

}

// During this second run, the same query object,

// lowNumbers, will be iterating over the new state

// of numbers[], producing different results:

Console.WriteLine("Second run numbers <= 3:");

foreach (int n in lowNumbers)

{

Console.WriteLine(n);

}

#endregion

return 0;

}

}

}

# Sequence operations

using System;

using System.Collections.Generic;

using System.Linq;

namespace LinqSamples

{

public class SequenceOperations

{

public List<Product> GetProductList() => Products.ProductList;

public List<Customer> GetCustomerList() => Customers.CustomerList;

public int EqualSequence()

{

#region equal-sequence

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}");

#endregion

return 0;

}

// Combine in Markdown.s

public int Linq97()

{

#region not-equal-sequence

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

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

bool match = wordsA.SequenceEqual(wordsB);

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

#endregion

return 0;

}

public int ConcatSeries()

{

#region concat-series

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);

}

#endregion

return 0;

}

public int ConcatProjection()

{

#region concat-projections

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);

}

#endregion

return 0;

}

public int DotProduct()

{

#region dot-product

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}");

#endregion

return 0;

}

}

}

# Set operations

using System;

using System.Collections.Generic;

using System.Linq;

namespace LinqSamples

{

public class SetOperations

{

public List<Product> GetProductList() => Products.ProductList;

public List<Customer> GetCustomerList() => Customers.CustomerList;

public int DistinctSyntax()

{

#region distinct-syntax

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

var uniqueFactors = factorsOf300.Distinct();

Console.WriteLine("Prime factors of 300:");

foreach (var f in uniqueFactors)

{

Console.WriteLine(f);

}

#endregion

return 0;

}

public int DistinctPropertyValues()

{

#region distinct-property-values

List<Product> products = GetProductList();

var categoryNames = (from p in products

select p.Category)

.Distinct();

Console.WriteLine("Category names:");

foreach (var n in categoryNames)

{

Console.WriteLine(n);

}

#endregion

return 0;

}

public int UnionSyntax()

{

#region union-syntax

int[] numbersA = { 0, 2, 4, 5, 6, 8, 9 };

int[] numbersB = { 1, 3, 5, 7, 8 };

var uniqueNumbers = numbersA.Union(numbersB);

Console.WriteLine("Unique numbers from both arrays:");

foreach (var n in uniqueNumbers)

{

Console.WriteLine(n);

}

#endregion

return 0;

}

public int UnionOfQueryResults()

{

#region union-query-results

List<Product> products = GetProductList();

List<Customer> customers = GetCustomerList();

var productFirstChars = from p in products

select p.ProductName[0];

var customerFirstChars = from c in customers

select c.CompanyName[0];

var uniqueFirstChars = productFirstChars.Union(customerFirstChars);

Console.WriteLine("Unique first letters from Product names and Customer names:");

foreach (var ch in uniqueFirstChars)

{

Console.WriteLine(ch);

}

#endregion

return 0;

}

public int IntersectSynxtax()

{

#region intersect-syntax

int[] numbersA = { 0, 2, 4, 5, 6, 8, 9 };

int[] numbersB = { 1, 3, 5, 7, 8 };

var commonNumbers = numbersA.Intersect(numbersB);

Console.WriteLine("Common numbers shared by both arrays:");

foreach (var n in commonNumbers)

{

Console.WriteLine(n);

}

#endregion

return 0;

}

public int IntersectQueryResults()

{

#region intersect-different-queries

List<Product> products = GetProductList();

List<Customer> customers = GetCustomerList();

var productFirstChars = from p in products

select p.ProductName[0];

var customerFirstChars = from c in customers

select c.CompanyName[0];

var commonFirstChars = productFirstChars.Intersect(customerFirstChars);

Console.WriteLine("Common first letters from Product names and Customer names:");

foreach (var ch in commonFirstChars)

{

Console.WriteLine(ch);

}

#endregion

return 0;

}

public int DifferenceOfSets()

{

#region except-syntax

int[] numbersA = { 0, 2, 4, 5, 6, 8, 9 };

int[] numbersB = { 1, 3, 5, 7, 8 };

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

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

foreach (var n in aOnlyNumbers)

{

Console.WriteLine(n);

}

#endregion

return 0;

}

public int DifferenceOfQueries()

{

#region difference-of-queries

List<Product> products = GetProductList();

List<Customer> customers = GetCustomerList();

var productFirstChars = from p in products

select p.ProductName[0];

var customerFirstChars = from c in customers

select c.CompanyName[0];

var productOnlyFirstChars = productFirstChars.Except(customerFirstChars);

Console.WriteLine("First letters from Product names, but not from Customer names:");

foreach (var ch in productOnlyFirstChars)

{

Console.WriteLine(ch);

}

#endregion

return 1;

}

}

}

# Referenties

<https://www.tutorialsteacher.com/linq/linq-tutorials>

<https://docs.microsoft.com/en-us/dotnet/api/system.linq?view=netframework-4.8>

https://www.codingame.com/playgrounds/213/using-c-linq---a-practical-overview