Ebooks

PyQt5 ebook
Tkinter ebook
SQLite Python
wxPython ebook
Windows API ebook
Java Swing ebook
Java games ebook
MySQL Java ebook

C# LINQ

last modified December 3, 2021

C# LINQ tutorial shows how to use Language Integrated Query (LINQ) in C#. C# tutorial is a comprehensive tutorial on C# language.

Language-Integrated Query (LINQ) is the name for a set of technologies based on the integration of query capabilities directly into the C# language. LINQ provides a consistent query experience for objects (LINQ to Objects), relational databases (LINQ to SQL), and XML (LINQ to XML).

LINQ extends the language by the addition of query expressions, which are similar to SQL statements. LINQ query expressions can be used to conveniently extract and process data from arrays, enumerable classes, XML documents, relational databases, and third-party data sources.

Query expressions can be used to query and to transform data from any LINQ-enabled data source. Query expressions have deferred execution. They are not executed until we iterate over the query variable, for example, in a foreach statement.

LINQ queries can be written in the query syntax or the method syntax.

C# LINQ query and method syntax

In LINQ, we can use either the query or the method syntax. A few methods, such as Append or Concat, do not have equivalents in the query syntax.

Program.cs

```
var words = new string[] { "falcon", "eagle", "sky", "tree", "water" };

// Query syntax
var res = from word in words
          where word.Contains('a')
          select word;

foreach (var word in res)
{
    Console.WriteLine(word);
}

Console.WriteLine("-----------");

// Method syntax
var res2 = words.Where(word => word.Contains('a'));

foreach (var word in res2)
{
    Console.WriteLine(word);
}
```

The example uses the query and the method syntax to find out all words that contain the 'a' character.

The LINQ extension methods are available in the System.Linq namespace.

```
// Query syntax
var res = from word in words
          where word.Contains('a')
          select word;
```

This is the query syntax; it is similar to SQL code.

```
// Method syntax
var res2 = words.Where(word => word.Contains('a'));
```

This is the method syntax; the methods can be chained.

```
$ dotnet run
falcon
eagle
water
-----------
falcon
eagle
water
```

C# LINQ element access

There are a couple of helper methods for accessing elements.

Program.cs

```
string[] words = { "falcon", "oak", "sky", "cloud", "tree", "tea", "water" };

Console.WriteLine(words.ElementAt(2));
Console.WriteLine(words.First());
Console.WriteLine(words.Last());

Console.WriteLine(words.First(word => word.Length == 3));
Console.WriteLine(words.Last(word => word.Length == 3));
```

In the example, we access elements of an array.

```
Console.WriteLine(words.ElementAt(2));
```

We get the third element from the array with ElementAt.

```
Console.WriteLine(words.First());
Console.WriteLine(words.Last());
```

We retrieve the first and the last element of the array.

```
Console.WriteLine(words.First(word => word.Length == 3));
Console.WriteLine(words.Last(word => word.Length == 3));
```

The First and Last methods also can take a predicate. We get the first an last elements that have three characters.

```
$ dotnet run
sky
falcon
water
oak
tea
```

The Prepend adds a value to the beginning of the sequence and the Append appends a value to the end of the sequence. Note that these methods do not modify the elements of the collection. Instead, they create a copy of the collection with the new elements.

Program.cs

```
int[] vals = {1, 2, 3};

vals.Prepend(0);
vals.Append(4);

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

var vals2 = vals.Prepend(0);
var vals3 = vals2.Append(4);

Console.WriteLine(string.Join(", ", vals3));
```

In the example, we prepend and append values to the array of integers.

```
$ dotnet run
```

```
0, 1, 2, 3, 4
```

C# LINQ select

The select clause or the Select method projects each element of a sequence into a new form. It selects, projects and transforms elements in a collection. The Select is usually called Map in other languages.

Program.cs

```
int[] vals = { 2, 4, 6, 8 };

var powered = vals.Select(e => Math.Pow(e, 2));
Console.WriteLine(string.Join(", ", powered));

string[] words = { "sky", "earth", "oak", "falcon" };
var wordLens = words.Select(e => e.Length);
Console.WriteLine(string.Join(", ", wordLens));
```

In the example, we transform an array of integers into a sequence of its powers and transform an array of words into a sequence of word lengths.

```
$ dotnet run
4, 16, 36, 64
3, 5, 3, 6
```

C# LINQ select into anonymous type

Projections are selection of specific fields from the returned objects. Projections are performed with the select clause. We can project fields into anonymous types.

Program.cs

```
User[] users =
{
  new (1, "John", "London", "2001-04-01"),
  new (2, "Lenny", "New York", "1997-12-11"),
  new (3, "Andrew", "Boston", "1987-02-22"),
  new (4, "Peter", "Prague", "1936-03-24"),
  new (5, "Anna", "Bratislava", "1973-11-18"),
  new (6, "Albert", "Bratislava", "1940-12-11"),
  new (7, "Adam", "Trnava", "1983-12-01"),
  new (8, "Robert", "Bratislava", "1935-05-15"),
  new (9, "Robert", "Prague", "1998-03-14"),
};

var res = from user in users
          where user.City == "Bratislava"
          select new { user.Name, user.City };

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

record User(int id, string Name, string City, string DateOfBirth);
```

In the example, we select users who live in Bratislava.

```
var res = from user in users
          where user.City == "Bratislava"
          select new { user.Name, user.City };
```

With the select new clause, we create an anonymous type with two fields: Name and City.

```
$ dotnet run
{ Name = Anna, City = Bratislava }, { Name = Albert, City = Bratislava },
{ Name = Robert, City = Bratislava }
```

C# LINQ SelectMany

The SelectMany flattens sequences into a single sequence.

Program.cs

```
int[][] vals = {
    new[] {1, 2, 3},
    new[] {4},
    new[] {5, 6, 6, 2, 7, 8},
};

var res = vals.SelectMany(array => array).OrderBy(x => x);

Console.WriteLine(string.Join(", ", res));
```

In the example, we have an array of arrays. With the SelectMany method, we flatten the two-dimensional array into an one-dimensional array values. The values are also ordered.

```
$ dotnet run
1, 2, 2, 3, 4, 5, 6, 6, 7, 8
```

In the next example, we flatten the nested lists into a single list of unique values.

Program.cs

```
var vals = new List<List<int>> {
    new List<int> {1, 2, 3, 3},
    new List<int> {4},
    new List<int> {5, 6, 6, 7, 7}
};

var res = vals.SelectMany(list => list)
              .Distinct()
              .OrderByDescending(e => e);

Console.WriteLine(string.Join(", ", res));
```

The Distinct method is used to get unique values.

```
$ dotnet run
7, 6, 5, 4, 3, 2, 1
```

C# LINQ Concat

The Concat method concatenates two sequences.

Program.cs

```
User[] users1 =
{
    new User("John", "Doe", "gardener"),
    new User("Jane", "Doe", "teacher"),
    new User("Roger", "Roe", "driver")
};

User[] users2 =
{
    new User("Peter", "Smith", "teacher"),
    new User("Lucia", "Black", "accountant"),
    new User("Michael", "Novak", "programmer")
};

var allUsers = users1.Concat(users2);

foreach (var user in allUsers)
{
    Console.WriteLine(user);
}

record User(string FirstName, string LastName, string Occupation);
```

We have two arrays of users. We merge them with Concat.

```
$ dotnet run
User { FirstName = John, LastName = Doe, Occupation = gardener }
User { FirstName = Jane, LastName = Doe, Occupation = teacher }
User { FirstName = Roger, LastName = Roe, Occupation = driver }
User { FirstName = Peter, LastName = Smith, Occupation = teacher }
User { FirstName = Lucia, LastName = Black, Occupation = accountant }
User { FirstName = Michael, LastName = Novak, Occupation = programmer }
```

C# LINQ filter

We can filter data with the where clause. The conditions can be combined with && or || operators.

Program.cs

```
var words = new List<string> { "sky", "rock", "forest", "new",
        "falcon", "jewelry", "eagle", "blue", "gray" };

var query = from word in words
            where word.Length == 4
            select word;

foreach (var word in query)
{
    Console.WriteLine(word);
}
```

In the example, we pick all words that have four letters.

```
$ dotnet run
rock
blue
gray
```

In our list, we have three words that satisfy the condition.

In the next example, we use the || operator to combine conditions.

Program.cs

```
var words = new List<string> { "sky", "rock", "forest", "new",
                "falcon", "jewelry", "small", "eagle", "blue", "gray" };

var res = from word in words
          where word.StartsWith('f') || word.StartsWith('s')
          select word;

foreach (var word in res)
{
    Console.WriteLine(word);
}
```

In the example, we pick all words that either start with 'f', or 's' characters.

```
$ dotnet run
sky
forest
falcon
small
```

In the following example, we apply two conditions with &&.

Program.cs

```
var cars = new List<Car>
{
    new ("Audi", 52642),
    new ("Mercedes", 57127),
    new ("Skoda", 9000),
    new ("Volvo", 29000),
    new ("Bentley", 350000),
    new ("Citroen", 21000),
    new ("Hummer", 41400),
    new ("Volkswagen", 21600),
};

var res = from car in cars
          where car.Price > 30000 && car.Price < 100000
          select new { car.Name, car.Price };

foreach (var car in res)
{
    Console.WriteLine($"{car.Name} {car.Price}");
}

record Car(string Name, int Price);
```

In the example, we filter the list of car objects with the where clause. We include all cars whose price is between 30000 and 100000.

```
$ dotnet run
Audi 52642
Mercedes 57127
Hummer 41400
```

C# LINQ Cartesian product

Cartesian Product is the multiplication of two sets to form the set of all ordered pairs.

Program.cs

```
char[] letters = "abcdefghi".ToCharArray();
char[] digits = "123456789".ToCharArray();

var coords =
    from l in letters
    from d in digits
    select $"{l}{d}";

foreach (var coord in coords)
{
    Console.Write($"{coord} ");

    if (coord.EndsWith("9"))
    {
        Console.WriteLine();
    }
}

Console.WriteLine();
```

In the example, we create a Cartesian product of letters and digits.

```
var coords =
    from l in letters
    from d in digits
    select $"{l}{d}";
```

To accomplish the task, we use two from clauses.

```
$ dotnet run
a1 a2 a3 a4 a5 a6 a7 a8 a9
b1 b2 b3 b4 b5 b6 b7 b8 b9
c1 c2 c3 c4 c5 c6 c7 c8 c9
d1 d2 d3 d4 d5 d6 d7 d8 d9
e1 e2 e3 e4 e5 e6 e7 e8 e9
f1 f2 f3 f4 f5 f6 f7 f8 f9
g1 g2 g3 g4 g5 g6 g7 g8 g9
h1 h2 h3 h4 h5 h6 h7 h8 h9
i1 i2 i3 i4 i5 i6 i7 i8 i9
```

C# LINQ Zip

The Zip method takes elements from two sequences and combines them. The pairs are created from elements from the same position.

Program.cs

```
string[] students = { "Adam", "John", "Lucia", "Tom"  };
int[] scores = { 68, 56, 90, 86 };

var result = students.Zip(scores, (e1, e2) => e1 + "'s score: " + e2);

foreach (var user in result)
{
    Console.WriteLine(user);
}

Console.WriteLine("---------------");

var left = new[] { 1, 2, 3 };
var right = new[] { 10, 20, 30 };

var products = left.Zip(right, (m, n) => m * n);

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

Console.WriteLine("---------------");

int[] codes = Enumerable.Range(1, 5).ToArray();
string[] states =
{
```

```
    "Alabama",
    "Alaska",
    "Arizona",
    "Arkansas",
    "California"
};

var CodesWithStates = codes.Zip(states, (code, state) => code + ": " + state);

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

In the example, we use the Zip method in three cases.

```
string[] students = { "Adam", "John", "Lucia", "Tom"  };
int[] scores = { 68, 56, 90, 86 };

var result = students.Zip(scores, (e1, e2) => e1 + "'s score: " + e2);
```

We have an array of students and a corresponding array of scores. We combine the two arrays into a single sequence with Zip. The e1 comes from the students array and the e2 from the scores array; from the same position.

```
var left = new[] { 1, 2, 3 };
var right = new[] { 10, 20, 30 };

var products = left.Zip(right, (m, n) => m * n);
```

Here we create products of values from two arrays.

```
int[] codes = Enumerable.Range(1, 5).ToArray();
string[] states =
{
    "Alabama",
    "Alaska",
    "Arizona",
    "Arkansas",
    "California"
};

var CodesWithStates = codes.Zip(states, (code, state) => code + ": " + state);
```

Finally, we give a number to each of the states array elements.

```
$ dotnet run
Adam's score: 68
John's score: 56
Lucia's score: 90
Tom's score: 86
---------------
10, 40, 90
---------------
1: Alabama
2: Alaska
3: Arizona
4: Arkansas
5: California
```

C# LINQ built-in aggregate calculations

LINQ allows us to calculate aggregate calculations, such as min, max, or sum.

Program.cs

```
var vals = new List<int> { 6, 2, -3, 4, -5, 9, 7, 8 };

var n1 = vals.Count();
Console.WriteLine($"There are {n1} elements");

var n2 = vals.Count(e => e % 2 == 0);
Console.WriteLine($"There are {n2} even elements");

var sum = vals.Sum();
Console.WriteLine($"The sum of all values is: {sum}");

var s2 = vals.Sum(e => e > 0 ? e : 0);
Console.WriteLine($"The sum of all positive values is: {s2}");

var avg = vals.Average();
Console.WriteLine($"The average of values is: {avg}");

var max = vals.Max();
Console.WriteLine($"The maximum value is: {max}");

var min = vals.Min();
Console.WriteLine($"The minimum value is: {min}");
```

In the example, we use the Count, Sum, Average, Max, and Min methods.

```
$ dotnet run
There are 8 elements
There are 4 even elements
The sum of all values is: 28
The sum of all positive values is: 36
The average of values is: 3.5
The maximum value is: 9
The minimum value is: -5
```

The following example uses query expressions.

Program.cs

```
var vals = new List<int> { 1, -2, 3, -4, 5, 6, 7, -8 };
var s = (from x in vals where x > 0 select x).Sum();

Console.WriteLine($"The sum of positive values is: {s}");

var words = new List<string> { "falcon", "eagle", "hawk", "owl" };
int len = (from x in words select x.Length).Sum();

Console.WriteLine($"There are {len} letters in the list");
```

In the example, we count the number of positive values in the vals list and the number of characters in the words list.

```
$ dotnet run
The sum of positive values is: 22
There are 18 letters in the list
```

C# LINQ custom aggregate calculations

Custom aggregate calculations can be computed with Aggregate. It applies an accumulator function over a sequence.

Program.cs

```
var vals = new List<int> { 6, 2, -3, 4, -5, 9, 7, 8 };

var sum = vals.Aggregate((total, next) => total + next);
Console.WriteLine($"The sum is {sum}");

var product = vals.Aggregate((total, next) => total * next);
Console.WriteLine($"The product is {product}");
```

In the example, we calculate the sum and the product of values in the list with Aggregate.

```
$ dotnet run
The sum is 28
The product is 362880
```

C# LINQ orderby

With the OrderBy method or the orderby clause we can sort the elements of a sequence.

Program.cs

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

var result = from e in vals
             orderby e ascending
             select e;

Console.WriteLine(string.Join(", ", result));
```

```
var result2 = from e in vals
              orderby e descending
              select e;

Console.WriteLine(string.Join(", ", result2));
```

In the example, we sort the integers in ascending and descending order. The ascending keyword is optional.

```
$ dotnet run
0, 1, 2, 3, 4, 5, 6, 7
7, 6, 5, 4, 3, 2, 1, 0
```

In the next example, we sort objects by multiple fields.

Program.cs

```
var users = new List<User>
{
    new ("John", "Doe", 1230),
    new ("Lucy", "Novak", 670),
    new ("Ben", "Walter", 2050),
    new ("Robin", "Brown", 2300),
    new ("Amy", "Doe", 1250),
    new ("Joe", "Draker", 1190),
    new ("Janet", "Doe", 980),
    new ("Albert", "Novak", 1930),
};

Console.WriteLine("sort ascending by last name and salary");

var sortedUsers = users.OrderBy(u => u.LastName).ThenBy(u => u.Salary);

foreach (var user in sortedUsers)
{
    Console.WriteLine(user);
}

Console.WriteLine("----------------------");

Console.WriteLine("sort descending by last name and salary");

var sortedUsers2 = users.OrderByDescending(u => u.LastName)
    .ThenByDescending(u => u.Salary);

foreach (var user in sortedUsers2)
{
    Console.WriteLine(user);
}

record User(string FirstName, string LastName, int Salary);
```

In the example, sort the users first by their last names, then by their salaries.

```
var sortedUsers = users.OrderBy(u => u.LastName).ThenBy(u => u.Salary);
```

We sort the users by their last names and then by their salaries in ascending order.

```
var sortedUsers2 = users.OrderByDescending(u => u.LastName)
    .ThenByDescending(u => u.Salary);
```

Here, we sort the users by their last names and then by their salaries in descending order.

```
$ dotnet run
sort ascending by last name and salary
User { FirstName = Robin, LastName = Brown, Salary = 2300 }
User { FirstName = Janet, LastName = Doe, Salary = 980 }
User { FirstName = John, LastName = Doe, Salary = 1230 }
User { FirstName = Amy, LastName = Doe, Salary = 1250 }
User { FirstName = Joe, LastName = Draker, Salary = 1190 }
User { FirstName = Lucy, LastName = Novak, Salary = 670 }
User { FirstName = Albert, LastName = Novak, Salary = 1930 }
User { FirstName = Ben, LastName = Walter, Salary = 2050 }
----------------------
sort descending by last name and salary
User { FirstName = Ben, LastName = Walter, Salary = 2050 }
User { FirstName = Albert, LastName = Novak, Salary = 1930 }
User { FirstName = Lucy, LastName = Novak, Salary = 670 }
User { FirstName = Joe, LastName = Draker, Salary = 1190 }
User { FirstName = Amy, LastName = Doe, Salary = 1250 }
User { FirstName = John, LastName = Doe, Salary = 1230 }
User { FirstName = Janet, LastName = Doe, Salary = 980 }
User { FirstName = Robin, LastName = Brown, Salary = 2300 }
```

C# LINQ MaxBy & MinBy

With the MaxBy and MinBy methods we can find the maximum and minimum values by the given property. The methods were introduced in .NET 6.

Program.cs

```
var users = new List<User>
{
    new ("John", "Doe", 1230),
    new ("Lucy", "Novak", 670),
    new ("Ben", "Walter", 2050),
    new ("Robin", "Brown", 2300),
    new ("Amy", "Doe", 1250),
    new ("Joe", "Draker", 1190),
    new ("Janet", "Doe", 980),
    new ("Albert", "Novak", 1930),
};

Console.WriteLine("User with max salary:");

var u1 = users.MaxBy(u => u.Salary);
Console.WriteLine(u1);

Console.WriteLine("User with min salary:");
var u2 = users.MinBy(u => u.Salary);
Console.WriteLine(u2);

record User(string FirstName, string LastName, int Salary);
```

In the example, we find the user with the maximum and minimum salary.

```
$ dotnet run
User with max salary:
User { FirstName = Robin, LastName = Brown, Salary = 2300 }
User with min salary:
User { FirstName = Lucy, LastName = Novak, Salary = 670 }
```

C# LINQ Reverse

The Reverse method inverts the order of the elements in a sequence. (Note that this is not the same as sorting in descending order.)

Program.cs

```
int[] vals = { 1, 3, 6, 0, -1, 2, 9, 9, 8 };

var reversed = vals.Reverse();
Console.WriteLine(string.Join(", ", reversed));

var reversed2 = (from val in vals select val).Reverse();
Console.WriteLine(string.Join(", ", reversed2));
```

In the example, we reverse the elements of an array using both method and query syntax.

```
$ dotnet run
8, 9, 9, 2, -1, 0, 6, 3, 1
8, 9, 9, 2, -1, 0, 6, 3, 1
```

C# LINQ group by

We can group data into categories based on a certain key.

Program.cs

```
var cars = new List<Car>
{
    new ("Audi", "red", 52642),
    new ("Mercedes", "blue", 57127),
    new ("Skoda", "black", 9000),
    new ("Volvo", "red", 29000),
    new ("Bentley", "yellow", 350000),
    new ("Citroen", "white", 21000),
    new ("Hummer", "black", 41400),
```

```
    new ("Volkswagen", "white", 21600),
};

var groups = from car in cars
             group car by car.Colour;

foreach (var group in groups)
{
    Console.WriteLine(group.Key);

    foreach (var car in group)
    {
        Console.WriteLine($" {car.Name} {car.Price}");
    }
}

record Car(string Name, string Colour, int Price);
```

In the example, we separate the available cars into groups by their colour.

```
$ dotnet run
red
  Audi 52642
  Volvo 29000
blue
  Mercedes 57127
black
  Skoda 9000
  Hummer 41400
yellow
  Bentley 350000
white
  Citroen 21000
  Volkswagen 21600
```

In the following example, we perform a grouping and aggregation operations.

Program.cs

```
Revenue[] revenues =
{
    new (1, "Q1", 2340),
    new (2, "Q1", 1200),
    new (3, "Q1", 980),
    new (4, "Q2", 340),
    new (5, "Q2", 780),
    new (6, "Q3", 2010),
    new (7, "Q3", 3370),
    new (8, "Q4", 540),
};

var res = from revenue in revenues
          group revenue by revenue.Quarter
          into g
          select new { Quarter = g.Key, Total = g.Sum(e => e.Amount) };

foreach (var line in res)
{
    Console.WriteLine(line);
}

record Revenue(int Id, string Quarter, int Amount);
```

We have revenues for four quarters. We group the revenues by the quarters and sum the amounts.

```
$ dotnet run
{ Quarter = Q1, Total = 4520 }
{ Quarter = Q2, Total = 1120 }
{ Quarter = Q3, Total = 5380 }
{ Quarter = Q4, Total = 540 }
```

We can apply a filter on aggregated data with `where` clause.

Program.cs

```
Revenue[] revenues =
{
    new (1, "Q1", 2340),
    new (2, "Q1", 1200),
    new (3, "Q1", 980),
    new (4, "Q2", 340),
    new (5, "Q2", 780),
    new (6, "Q3", 2010),
    new (7, "Q3", 3370),
    new (8, "Q4", 540),
};

var res = from revenue in revenues
          group revenue by revenue.Quarter
          into g
          where g.Count() == 2
          select new { Quarter = g.Key, Total = g.Sum(c => c.Amount) };


foreach (var line in res)
{
    Console.WriteLine(line);
}

record Revenue(int Id, string Quarter, int Amount);
```

In the example, we pick only those quarters, which have exactly two revenues.

```
$ dotnet run
{ Quarter = Q2, Total = 1120 }
{ Quarter = Q3, Total = 5380 }
```

C# LINQ word frequency

In the next example, we count the frequency of words in a file.

```
$ wget https://raw.githubusercontent.com/janbodnar/data/main/the-king-james-bible.txt
```

We use the King James Bible.

Program.cs

```
using System.Text.RegularExpressions;

var fileName = "/home/janbodnar/Documents/the-king-james-bible.txt";
var text = File.ReadAllText(fileName);

var matches = new Regex("[a-z-A-Z']+").Matches(text);
var words = matches.Select(m => m.Value).ToList();

var res = words
    .GroupBy(m => m)
    .OrderByDescending(g => g.Count())
    .Select(x => new { word = x.Key, Count = x.Count() })
    .Take(10);

foreach (var r in res)
{
    Console.WriteLine($"{r.word}: {r.Count}");
}
```

We count the frequency of the words from the King James Bible.

```
var matches = new Regex("[a-z-A-Z']+").Matches(text);
var words = matches.Select(m => m.Value).ToList();
```

We find all the matches witch `Matches` method. From the match collection, we get all the words into a list.

```
var res = words
    .GroupBy(m => m)
    .OrderByDescending(g => g.Count())
    .Select(x => new { word = x.Key, Count = x.Count() })
    .Take(10);
```

The words are grouped and ordered by frequency in descending order. We take the first top words.

```
$ dotnet run
the 62103
and 38848
of 34478
to 13400
And 12846
```

```
that 12576
in 12331
shall 9760
he 9665
unto 8942
```

C# LINQ join

The join clause joins sequences.

Program.cs

```
string[] basketA = { "coin", "book", "fork", "cord", "needle" };
string[] basketB = { "watches", "coin", "pen", "book", "pencil" };

var res = from item1 in basketA
          join item2 in basketB
          on item1 equals item2
          select item1;

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

We have two arrays in the example. With the join clause, we find all items that are present in both arrays.

```
$ dotnet run
coin
book
```

The coin and book words are included in both arrays.

C# LINQ partitioning

The Skip method skips the specified number of elements from the start of the sequence and returns the remaining elements. The SkipLast method returns the sequence elements with the specified last number of elements omitted. The SkipWhile skips elements in a sequence as long as the specified condition is true and then returns the remaining elements.

The Take method returns the specified number of contiguous elements from the start of a sequence. The TakeLast method omits all but the specified number of the last elements. The TakeWhile method returns elements from a sequence as long as a specified condition is true, and then skips the remaining elements.

Note: The SkipWhile and TakeWhile methods stop at the first non-matching element.

Program.cs

```
int[] vals = { 1, 2, 7, 8, 5, 6, 3, 4, 9, 10 };

var res1 = vals.Skip(3);
Console.WriteLine(string.Join(", ", res1));

var res2 = vals.SkipLast(3);
Console.WriteLine(string.Join(", ", res2));

var res3 = vals.SkipWhile(e => e < 5);
Console.WriteLine(string.Join(", ", res3));

Console.WriteLine("----------");

var res4 = vals.Take(3);
Console.WriteLine(string.Join(", ", res4));

var res5 = vals.TakeLast(3);
Console.WriteLine(string.Join(", ", res5));

var res6 = vals.TakeWhile(e => e < 5);
Console.WriteLine(string.Join(", ", res6));
```

The example uses all the six partition methods.

```
$ dotnet run
8, 5, 6, 3, 4, 9, 10
1, 2, 7, 8, 5, 6, 3
7, 8, 5, 6, 3, 4, 9, 10
----------
1, 2, 7
4, 9, 10
1, 2
```

C# LINQ convertions

We can transform the returned enumerable into a list, array, or dictionary.

Program.cs

```
User[] users =
{
  new (1, "John", "London", "2001-04-01"),
  new (2, "Lenny", "New York", "1997-12-11"),
  new (3, "Andrew", "Boston", "1987-02-22"),
  new (4, "Peter", "Prague", "1936-03-24"),
  new (5, "Anna", "Bratislava", "1973-11-18"),
  new (6, "Albert", "Bratislava", "1940-12-11"),
  new (7, "Adam", "Trnava", "1983-12-01"),
  new (8, "Robert", "Bratislava", "1935-05-15"),
  new (9, "Robert", "Prague", "1998-03-14"),
};

string[] cities = (from user in users
                   select user.City).Distinct().ToArray();

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

Console.WriteLine("------------");

List<User> inBratislava = (from user in users
                           where user.City == "Bratislava"
                           select user).ToList();

foreach (var user in inBratislava)
{
    Console.WriteLine(user);
}

Console.WriteLine("------------");

Dictionary<int, string> userIds =
        (from user in users
         select user).ToDictionary(user => user.id, user => user.Name);

foreach (var kvp in userIds)
{
    Console.WriteLine($"{kvp.Key}: {kvp.Value}");
}

record User(int id, string Name, string City, string DateOfBirth);
```

We execute three queries on our data source; the resulting enumerable is transformed into a list, array, and dictionary.

```
string[] cities = (from user in users
                   select user.City).Distinct().ToArray();
```

In this query, we select all cities from the data source. We apply the Distinct method and finally call the ToArray method.

```
List<User> inBratislava = (from user in users
                           where user.City == "Bratislava"
                           select user).ToList();
```

Here we get a list of users who live in Bratislava; we call the ToList method.

```
Dictionary<int, string> userIds =
        (from user in users
         select user).ToDictionary(user => user.id, user => user.Name);
```

In this query, we turn the user names and their ids into a dictionary.

```
$ dotnet run
London, New York, Boston, Prague, Bratislava, Trnava
-----------
User { id = 5, Name = Anna, City = Bratislava, DateOfBirth = 1973-11-18 }
User { id = 6, Name = Albert, City = Bratislava, DateOfBirth = 1940-12-11 }
User { id = 8, Name = Robert, City = Bratislava, DateOfBirth = 1935-05-15 }
-----------
1: John
2: Lenny
3: Andrew
4: Peter
5: Anna
6: Albert
7: Adam
8: Robert
9: Robert
```

C# LINQ generating sequences

The Range, Repeat, and Empty methods can be used to generate sequences.

Program.cs

```
var res = Enumerable.Range(1, 10).Select(e => Math.Pow(e, 3));
Console.WriteLine(string.Join(", ", res));

Console.WriteLine("------------------");

int[] vals = { 8, 4, 3, 2, 5, 11, 15, 10, 3, 5, 6 };

var lines = vals.Select(e => Enumerable.Repeat("*", e)).ToArray();

foreach (var line in lines)
{
    Console.WriteLine(string.Join("", line));
}

Console.WriteLine("------------------");

int[] nums = { 1, 3, 2, 3, 3, 3, 4, 4, 10, 10 };

var powered = nums.Aggregate(Enumerable.Empty<double>(), (total, next) =>
        total.Append(Math.Pow(next, 2)));

foreach (var val in powered)
{
    Console.WriteLine(val);
}
```

In the example, we use the Range, Repeat, and Empty methods to generate sequences.

```
var res = Enumerable.Range(1, 10).Select(e => Math.Pow(e, 3));
```

With Range, we generate integers 1 through 10 and them cube them.

```
int[] vals = { 8, 4, 3, 2, 5, 11, 15, 10, 3, 5, 6 };

var lines = vals.Select(e => Enumerable.Repeat("*", e)).ToArray();
```

With the help of the Repeat method, we generate a horizontal bar for each of the values from the vals array.

```
var powered = nums.Aggregate(Enumerable.Empty<double>(), (total, next) =>
    total.Append(Math.Pow(next, 2)));
```

We use the Empty method to create an empty sequence for the Aggregate method.

```
$ dotnet run
1, 8, 27, 64, 125, 216, 343, 512, 729, 1000
------------------
********
****
***
**
*****
***********
***************
**********
****
*****
******
------------------
1
9
4
9
9
9
16
16
100
100
```

C# LINQ quantifiers

With quantifiers, we check for certain conditions.

Program.cs

```
var vals = new List<int> { -1, -3, 0, 1, -3, 2, 9, -4 };

bool positive = vals.Any(x => x > 0);

if (positive)
{
    Console.WriteLine("There is a positive value");
}

bool allPositive = vals.All(x => x > 0);

if (allPositive)
{
    Console.WriteLine("All values are positive");
}

bool hasSix = vals.Contains(6);

if (hasSix)
{
    Console.WriteLine("6 value is in the array");
}
```

With the Any method, we check whether any element in the list is a positive value. With the All method, we check whether all elements in the list are positive. Finally, with the Contains method we determine whether the list contains value six.

C# LINQ set operations

LINQ has methods performing set operations, including Union, Intersect, Except, and Distinct.

Program.cs

```
var vals1 = "abcde".ToCharArray();
var vals2 = "defgh".ToCharArray();

var data = vals1.Union(vals2);
Console.WriteLine("{" + string.Join(" ", data) + "}");

var data2 = vals1.Intersect(vals2);
Console.WriteLine("{" + string.Join(" ", data2) + "}");

var data3 = vals1.Except(vals2);
Console.WriteLine("{" + string.Join(" ", data3) + "}");

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

var data4 = nums.Distinct();
Console.WriteLine("{" + string.Join(" ", data4) + "}");
```

In the example, we do set operations on array elements.

```
var vals1 = "abcde".ToCharArray();
var vals2 = "defgh".ToCharArray();
```

With the help of the ToCharArray, we create two arrays of characters.

```
var data = vals1.Union(vals2);
```

The Union produces the set union of the two arrays.

```
var data2 = vals1.Intersect(vals2);
```

The Intersect produces the set intersection of the two arrays.

```
var data3 = vals1.Except(vals2);
```

The Except produces the set difference of the two arrays.

```
var data4 = nums.Distinct();
```

The Distinct returns distinct elements from the array. In other words, it creates a set from the array.

```
$ dotnet run
{a b c d e f g h}
{d e}
{a b c}
{1 2 3 4 5 6 7 8}
```

C# LINQ XML

LINQ can be used to process XML.

Program.cs

```
using System.Xml.Linq;

string myXML = @"
<Users>
    <User>
        <Name>Jack</Name>
        <Sex>male</Sex>
    </User>
    <User>
        <Name>Paul</Name>
        <Sex>male</Sex>
    </User>
    <User>
        <Name>Frank</Name>
        <Sex>male</Sex>
    </User>
    <User>
        <Name>Martina</Name>
        <Sex>female</Sex>
    </User>
    <User>
        <Name>Lucia</Name>
        <Sex>female</Sex>
    </User>
</Users>";

var xdoc = new XDocument();
xdoc = XDocument.Parse(myXML);

var females = from u in xdoc.Root.Descendants()
              where (string)u.Element("Sex") == "female"
              select u.Element("Name");

foreach (var e in females)
{
    Console.WriteLine("{0}", e);
}
```

We parse the XML data and choose all female names.

```
$ dotnet run
<Name>Martina</Name>
<Name>Lucia</Name>
```

In this tutorial, we have worked with LINQ in C#.

Read C# tutorial or list all C# tutorials.