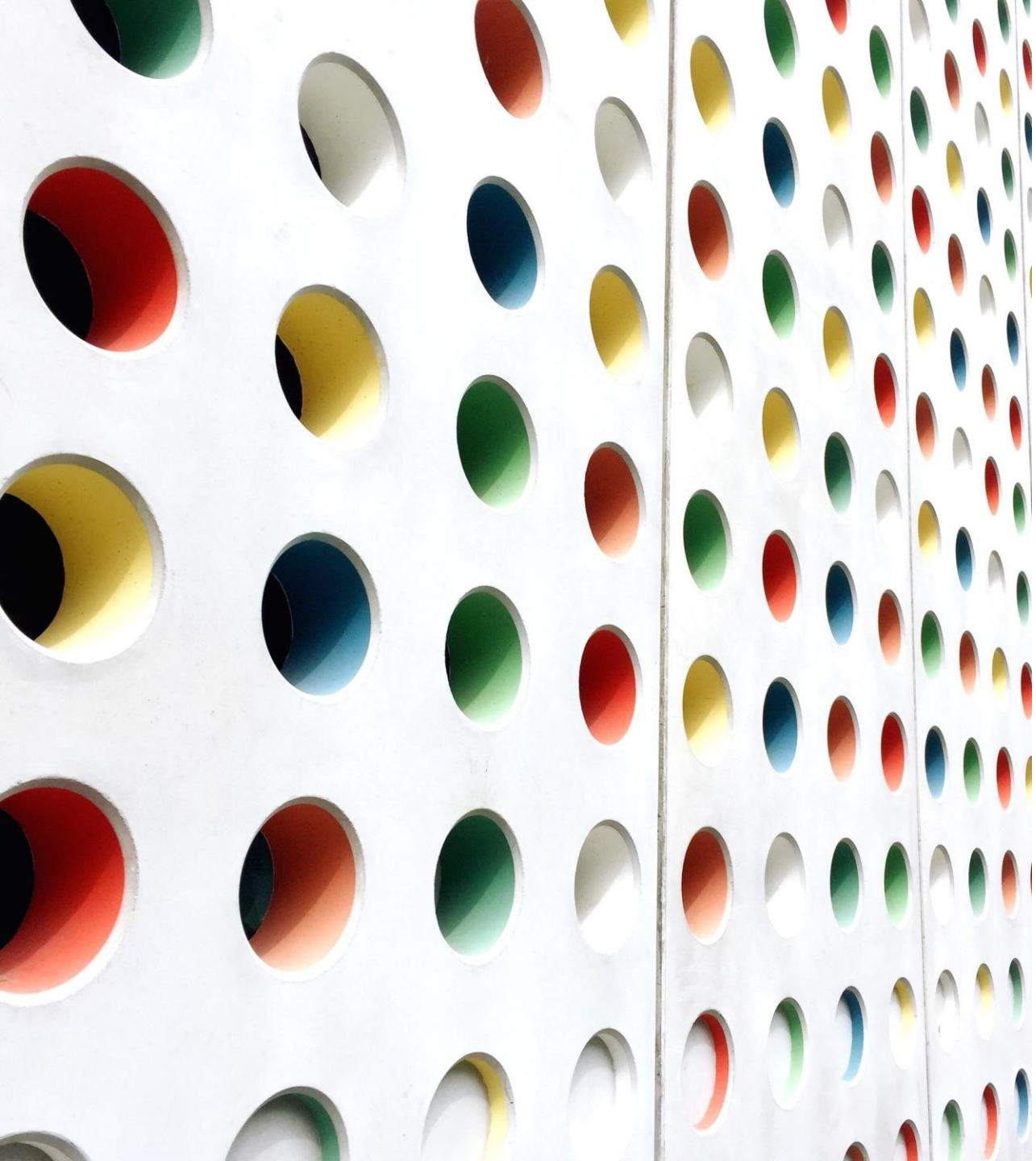


LINQ

ARCTECH INFO PRIVATE LIMITED

What is LINQ

Language Integrated Query

- Pronounced as "LINK"

Created by Erik Meijer at Microsoft

- Originally released with .NET Framework 3.5 in 2007.

A set of classes and methods that allows you to access data from various sources like

In memory data	Relational Databases	Active Directory Entries	XML Documents
Array & enumerable classes	SQL Server, Oracle, etc.	Database which stores network resource info	https://www.w3schools.com/xml/plant_catalog.xml

LINQ vs SQL

LINQ is a Microsoft .NET Component which adds native data querying capabilities to .NET languages

LINQ syntax uses existing .NET languages like C#, Vb.NET etc.

LINQ is maintained by Microsoft and your syntax will remain the same, irrespective of the data source it is connected to

LINQ can be used to query a wide range of data sources.

Syntax has standard operators / methods like select, from, where, orderby, remove

SQL is an ANSI standard language to query data in RDBMS, like Microsoft SQL Server, Oracle RDBMS, Oracle MySQL

SQL has its own syntax which you have to learn.

Though ANSI/ISO publishes the SQL language specification, many implementations will have some variations in syntax.

SQL is primarily used for querying data from RDBMS

Syntax has standard operators like SELECT, FROM, WHERE, ORDER BY, DELETE, Update

LINQ with In-Memory Data

LINQ primarily operate on data like

- Collection of objects in C# using LINQ syntax (array, IEnumerable, ICollection, IList, etc.)
- Tables in a RDBMS using LINQ syntax with Entity Framework

Example of In Memory Data query without LINQ vs with LINQ

```
private static List<int> WithoutLinq()
{
    int[] arr = { 1, 2, 3, 5, 6, 7, 8, 9, 10 };

    var list = new List<int>();

    foreach (var item in arr)
    {
        if (item % 2 == 0)
            list.Add(item);
    }

    return list;
}
```

```
public static IEnumerable<int> WithLinq()
{
    int[] arr = { 1, 2, 3, 5, 6, 7, 8, 9, 10 };

    var list = arr.Where(item => item % 2 == 0)
                  .Select(item => item);

    return list;
}
```

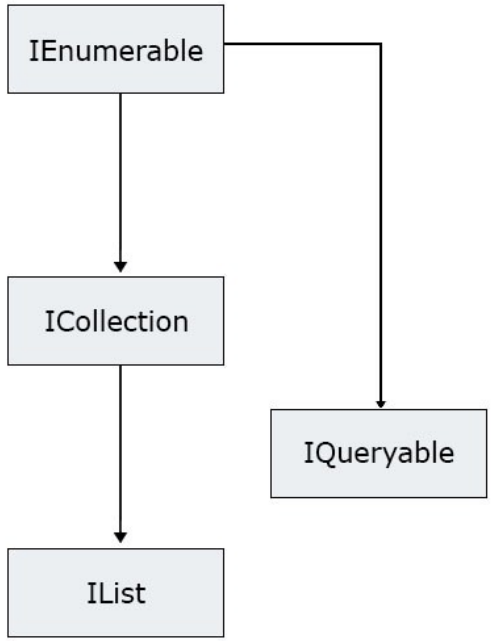
In Memory Data Collection in C#

In .NET, we have collections like IEnumerable, ICollection, Ilist, IQueryable

However, they all have specific characteristics that differentiate them and makes them adaptable to certain scenarios.

When you work with various libraries, you will encounter multiple methods that return one of these.

Which ones should you use?



INHERITANCE

IEnumerable

An **IEnumerable** is a list or a container which can hold some items.

You can iterate through each element in the **IEnumerable**.

You can not edit the items like adding, deleting, updating, etc. instead you just use a container to contain a list of items.

It is the most basic type of list container.

With in-memory data, LINQ returns an **IEnumerable**.

List<T>

List<T> : IList

The List<T> is a collection of strongly typed objects

The objects can be accessed by index

It has methods for sorting, searching, and modifying list.

Generics in C#

If you have multiple methods or classes with similar logic but works on multiple datatype, use generics

Without generics, you to create multiple methods and/or classes

With generics you can create a single methods and/or class and specify the generic parameter

- public void Show<T1, T2>(T1 param1, T2 param2) {}
- public class DataStore<T>
  {
  public void Add(T obj) {}
  }

Instantiate Generic classes / methods as follows

- var ds = new DataStore<Student>()
  ds.Add(new Student{ … });
- Show<int, int>(10, 20); or Show<int, string>(10, "Hello"); or Show(10, 20); [Not angled brackets are not needed]

You can restrict the type of generic parameter by using where clause

- public class DataStore<T> where T: class

LINQ

Query expressions are written in a declarative query syntax.

By using query syntax, you can perform filtering, ordering, and grouping operations on data sources with a minimum of code.

You use the same basic query expression patterns to query and transform data.

```
// Specify the data source.
int[] scores = { 97, 92, 81, 60 };

// Define the query expression.
IEnumerable<int> scoreQuery =
                from score in scores
                where score > 80
                select score;

// Execute the query.
foreach (int i in scoreQuery)
{
    Console.Write(i + " ");
}
// Output: 97 92 81
```

LINQ benefits

LINQ binds the gap between relational and object-oriented approaches

LINQ speeds up development time as

- C# developers do not have to learn SQL
- It catches errors at compile time
- includes IntelliSense & Debugging support.

LINQ expressions are Strongly Typed.

See an example of IntelliSense when selecting all orders in last 2 days

```
var ordersQuery =
  from order in _dbContext.Orders
  where order.OrderDate >= DateTime.Now.AddDays(-2)
  select order;

var orders = ordersQuery.AsEnumerable();

foreach (var order in orders)
{
    // order
}
```

LINQ expressions overview 1/2

Query expressions can be used to query and to transform data from any LINQ-enabled data source.

Query expressions are easy to grasp because they use many familiar C# language constructs

The variables in a query expression are all strongly typed

A query is not executed until you iterate over the query variable

Any query that can be expressed by using query syntax can also be expressed by using method syntax.

```
IEnumerable<int> scoreQuery =
                 from score in scores
                 where score > 80
                 select score;
```

```
IEnumerable<int> scoreQuery =
        scores.Where(score => score > 80);
```

LINQ expressions overview 2/2

As a rule, when you write LINQ queries, we recommend that

- you use query syntax whenever possible and
- method syntax whenever necessary.

There is no semantic or performance difference between the two different forms.

Query expressions are often more readable than equivalent expressions written in method syntax.

Some query operations, such as Count or Max, have no equivalent query expression clause and must therefore be expressed as a method call.

Method syntax can be combined with query syntax in various ways

Lamda Expression 1/2

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

//Query syntax:
IEnumerable<int> numQuery1 =
    from num in numbers
    where num % 2 == 0
    orderby num
    select num;

//Method syntax:
IEnumerable<int> numQuery2 = numbers.
    Where(num => num % 2 == 0).
    OrderBy(num => num);
```

Notice that the conditional expression (n % 2 == 0) is passed as an in-line argument to the Where method

`Where(num => num % 2 == 0)`

This inline expression is called a lambda expression

It is a convenient way to write code that would otherwise be very cumbersome

```
int[] numbers = { 5, 10, 8, 3, 6, 12};
numbers.Where(num => num % 2 == 0)
```

Lamda Expression 2/2

In C# => is the lambda operator and is read as "goes to”

The code inside the Where method is executed for every element in the array

The num on the left of the operator is the input variable which corresponds to num in the query expression and represents an element of the array

To get started using LINQ, you do not have to use lambdas extensively.

However, certain queries can only be expressed in method syntax and some of those require lambda expressions.

After you become more familiar with lambdas, you will find that they are a powerful and flexible tool