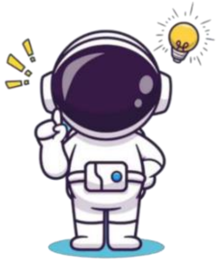
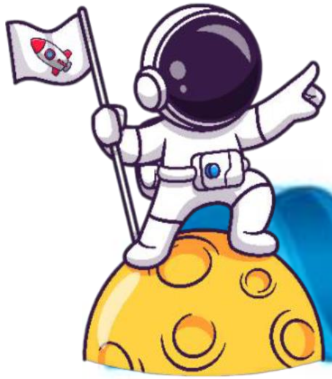




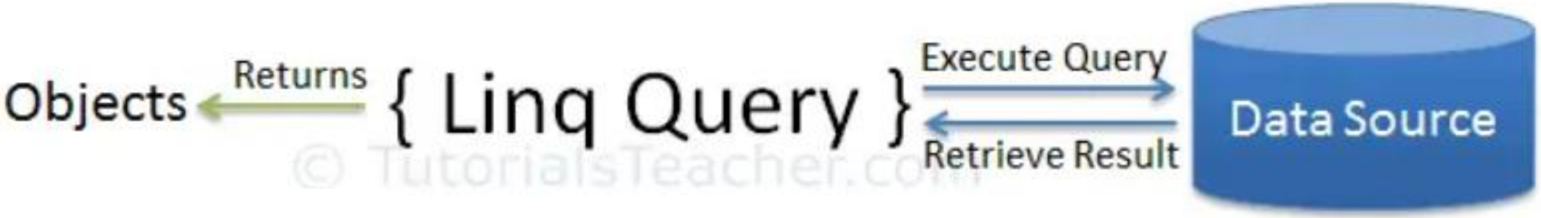
LINQ

Microsoft®
.NET

What is LINQ?

Lenguaje integrado de consultas(LINQ) es un potente conjunto de tecnologías basadas en la integración de las capacidades de consulta directamente en el lenguaje C#. Las consultas LINQ son la construcción de lenguaje de primera clase en C# NET, al igual que las clases, los métodos y los eventos. El LINQ proporciona una experiencia de consulta consistente para consultar objetos (LINQ a objetos), bases de datos relacionales (LINQ a SQL) y XML (LINQ a XML).

LINQ queries return results as objects. It enables you to uses object-oriented approach on the result set and not to worry about transforming different formats of results into objects.



Advantages of LINQ

Familiar language: Developers don't have to learn a new query language for each type of data source or data format.

Less coding: It reduces the amount of code to be written as compared with a more traditional approach.

Readable code: LINQ makes the code more readable so other developers can easily understand and maintain it.

Standardized way of querying multiple data sources: The same LINQ syntax can be used to query multiple data sources.

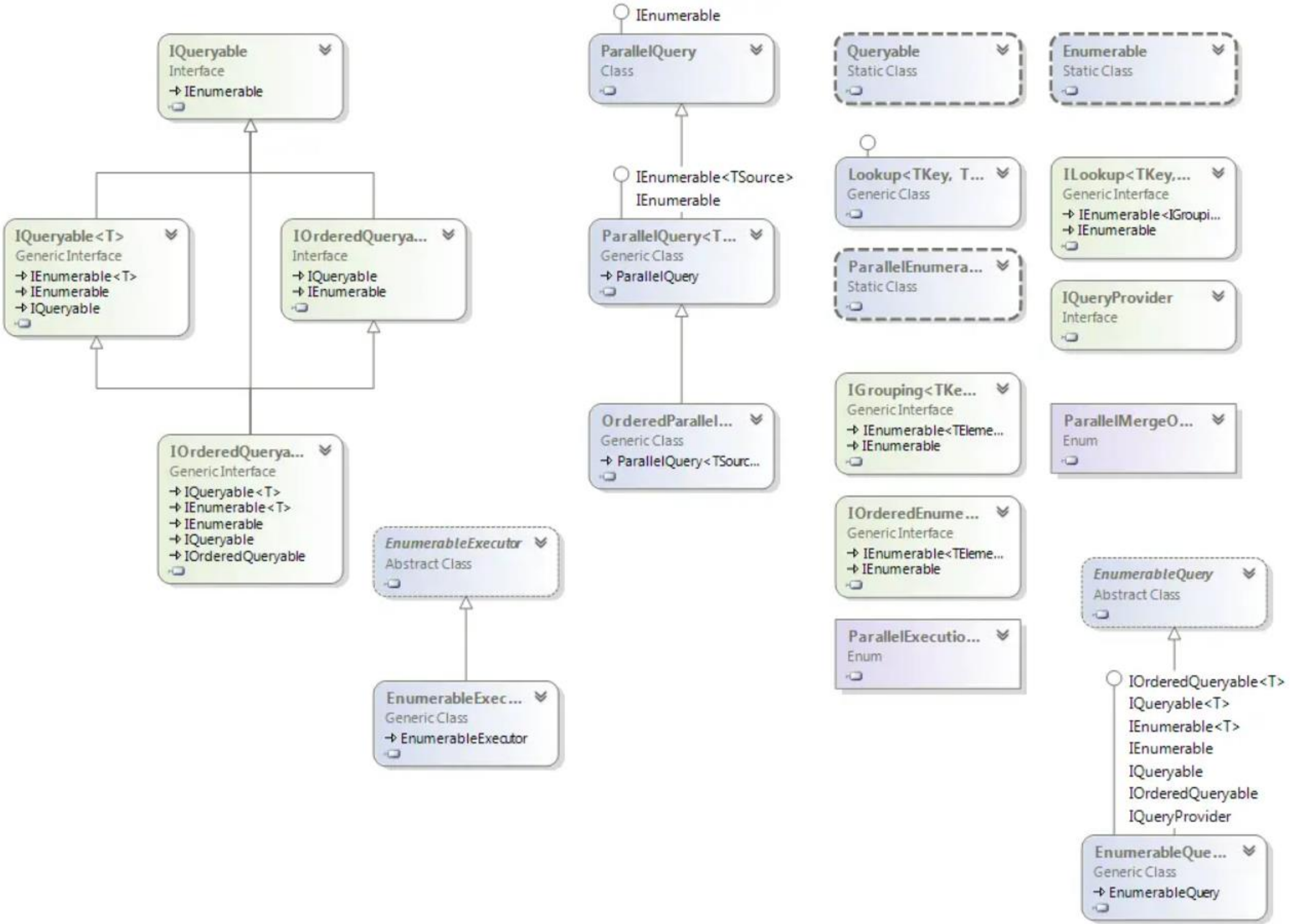
Compile time safety of queries: It provides type checking of objects at compile time.

IntelliSense Support: LINQ provides IntelliSense for generic collections.

Shaping data: You can retrieve data in different shapes.

LINQ API in .NET

We can write LINQ queries for the classes that implement IEnumerable<T> or IQueryable<T> interface. The *System.Linq* namespace includes the following classes and interfaces require for LINQ queries.



IQueryable
Interface
→ IEnumerable

IQueryable<T>
Generic Interface
→ IEnumerable<T>
→ IEnumerable
→ IQueryable

IOrderedQuerya...
Interface
→ IQueryable
→ IEnumerable

IOrderedQuerya...
Generic Interface
→ IQueryable<T>
→ IEnumerable<T>
→ IEnumerable
→ IQueryable
→ IOrderedQueryable

EnumerableExecutor
Abstract Class

EnumerableExec...
Generic Class
→ EnumerableExecutor

IEnumerable

ParallelQuery
Class

IEnumerable<TSource>
IEnumerable

ParallelQuery<T...
Generic Class
→ ParallelQuery

OrderedParallel...
Generic Class
→ ParallelQuery<TSourc...

Queryable
Static Class

Lookup<TKey, T...
Generic Class

ParallelEnumera...
Static Class

IGrouping<TKe...
Generic Interface
→ IEnumerable<TEleme...
→ IEnumerable

IOrderedEnume...
Generic Interface
→ IEnumerable<TEleme...
→ IEnumerable

ParallelExecutio...
Enum

Enumerable
Static Class

ILookup<TKey,...
Generic Interface
→ IEnumerable<IGroupi...
→ IEnumerable

IQueryProvider
Interface

ParallelMergeO...
Enum

EnumerableQuery
Abstract Class

IOrderedQueryable<T>
IQueryable<T>
IEnumerable<T>
IEnumerable
IQueryable
IOrderedQueryable
IQueryProvider

EnumerableQue...
Generic Class
→ EnumerableQuery

LINQ
Microsoft®
.NET

LINQ API

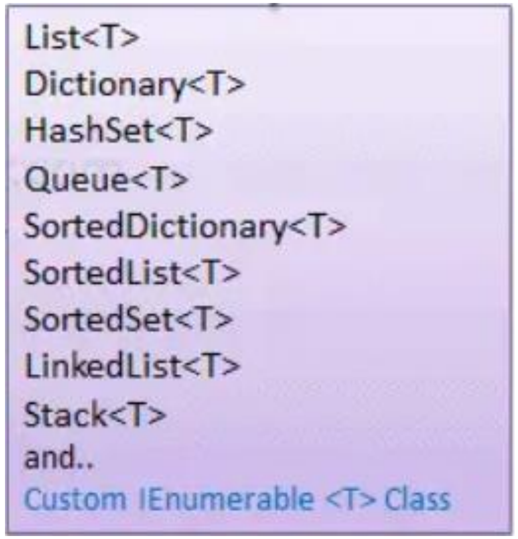
LINQ queries uses extension methods for classes that implement `IEnumerable` or `IQueryable` interface. The `Enumerable` and `Queryable` are two static classes that contain extension methods to write LINQ queries.

TIPS

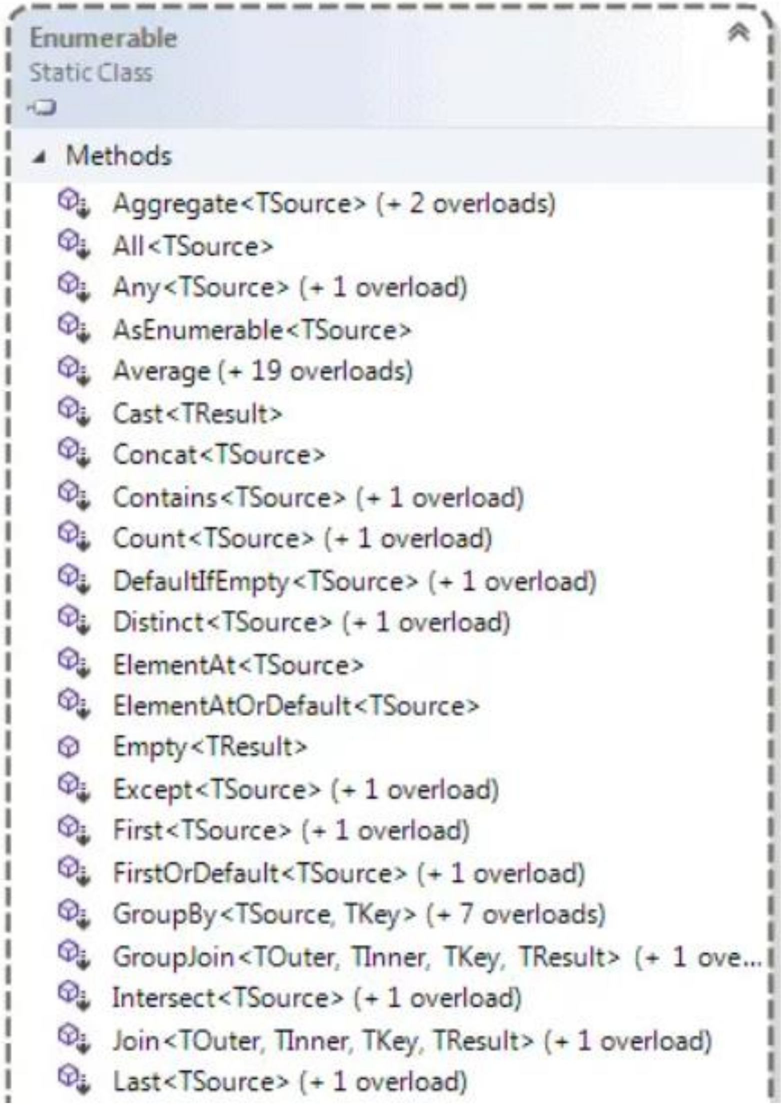
System.Linq namespace is included by default when you add a new class in Visual Studio.

Enumerable

The Enumerable class includes extension methods for the classes that implement `IEnumerable<T>` interface, for example all the built-in collection classes implement `IEnumerable<T>` interface and so we can write LINQ queries to retrieve data from the built-in collections.

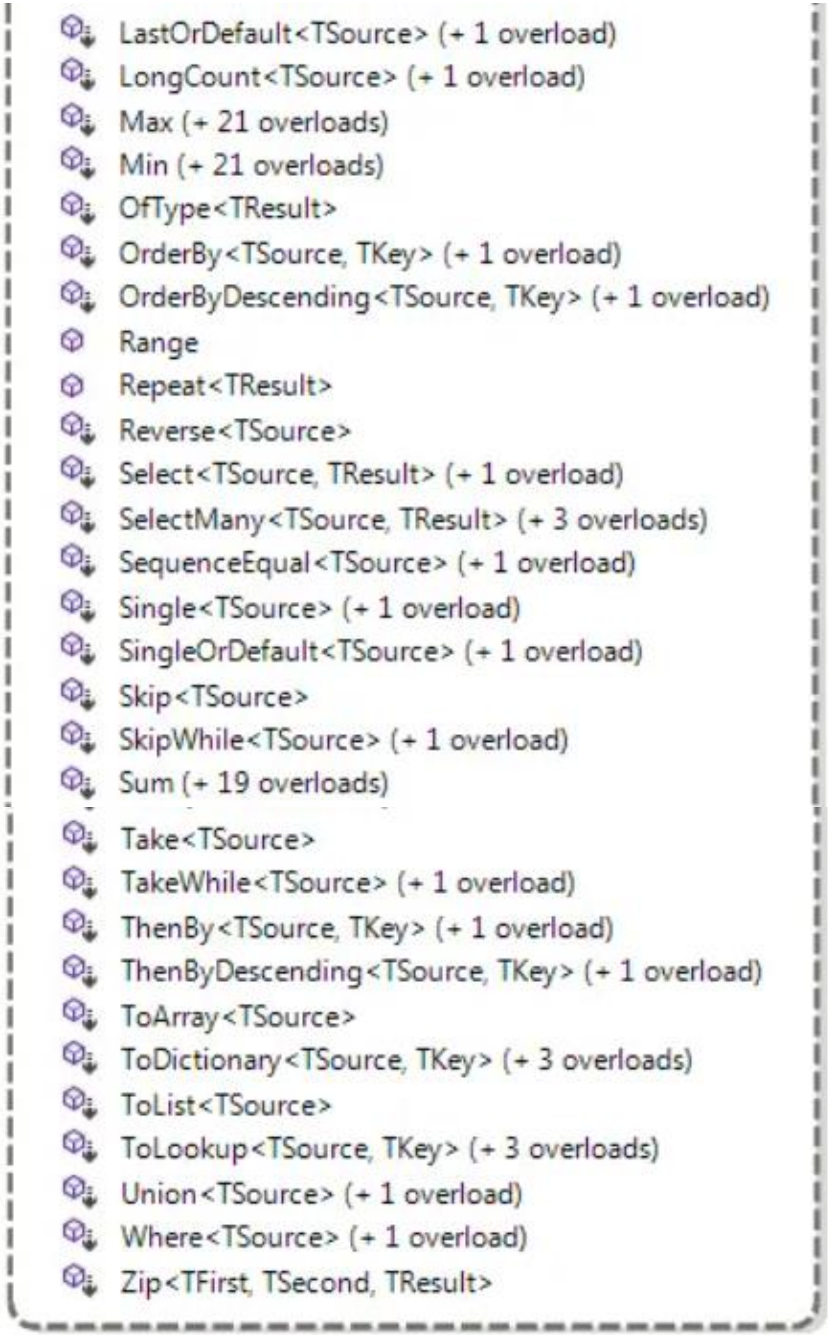


List<T>
Dictionary<T>
HashSet<T>
Queue<T>
SortedDictionary<T>
SortedList<T>
SortedSet<T>
LinkedList<T>
Stack<T>
and..
Custom IEnumerable <T> Class



Enumerable
Static Class

- Methods
  - Aggregate<TSource> (+ 2 overloads)
  - All<TSource>
  - Any<TSource> (+ 1 overload)
  - AsEnumerable<TSource>
  - Average (+ 19 overloads)
  - Cast<TResult>
  - Concat<TSource>
  - Contains<TSource> (+ 1 overload)
  - Count<TSource> (+ 1 overload)
  - DefaultIfEmpty<TSource> (+ 1 overload)
  - Distinct<TSource> (+ 1 overload)
  - ElementAt<TSource>
  - ElementAtOrDefault<TSource>
  - Empty<TResult>
  - Except<TSource> (+ 1 overload)
  - First<TSource> (+ 1 overload)
  - FirstOrDefault<TSource> (+ 1 overload)
  - GroupBy<TSource, TKey> (+ 7 overloads)
  - GroupJoin<TOuter, TInner, TKey, TResult> (+ 1 ove...
  - Intersect<TSource> (+ 1 overload)
  - Join<TOuter, TInner, TKey, TResult> (+ 1 overload)
  - Last<TSource> (+ 1 overload)

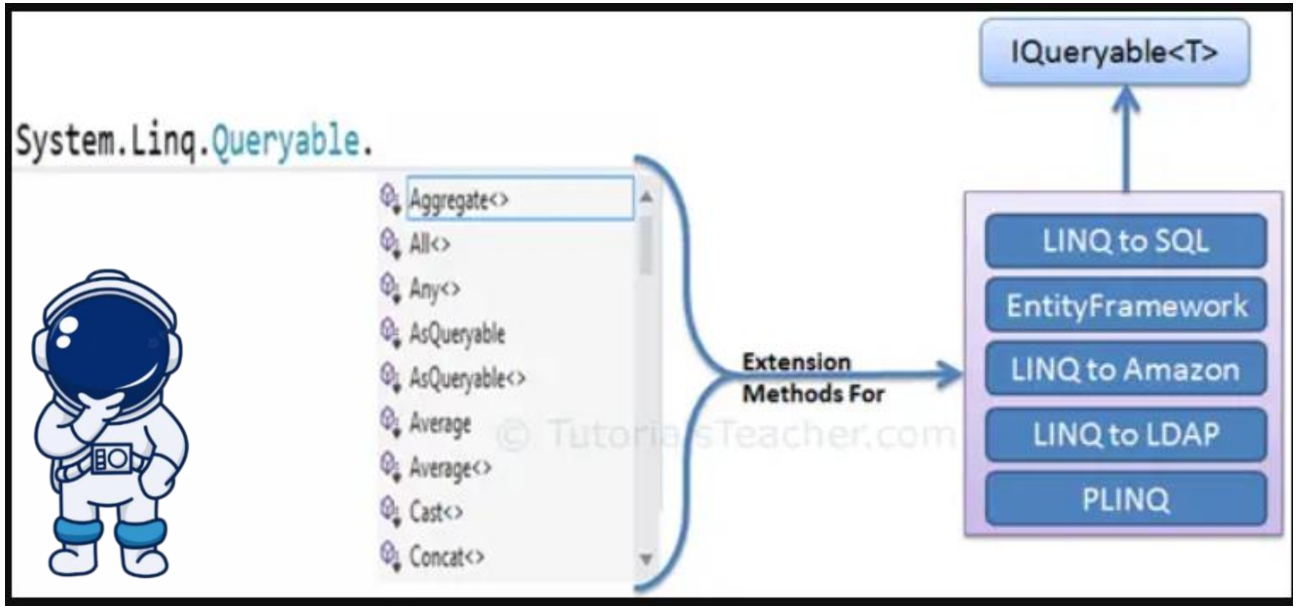


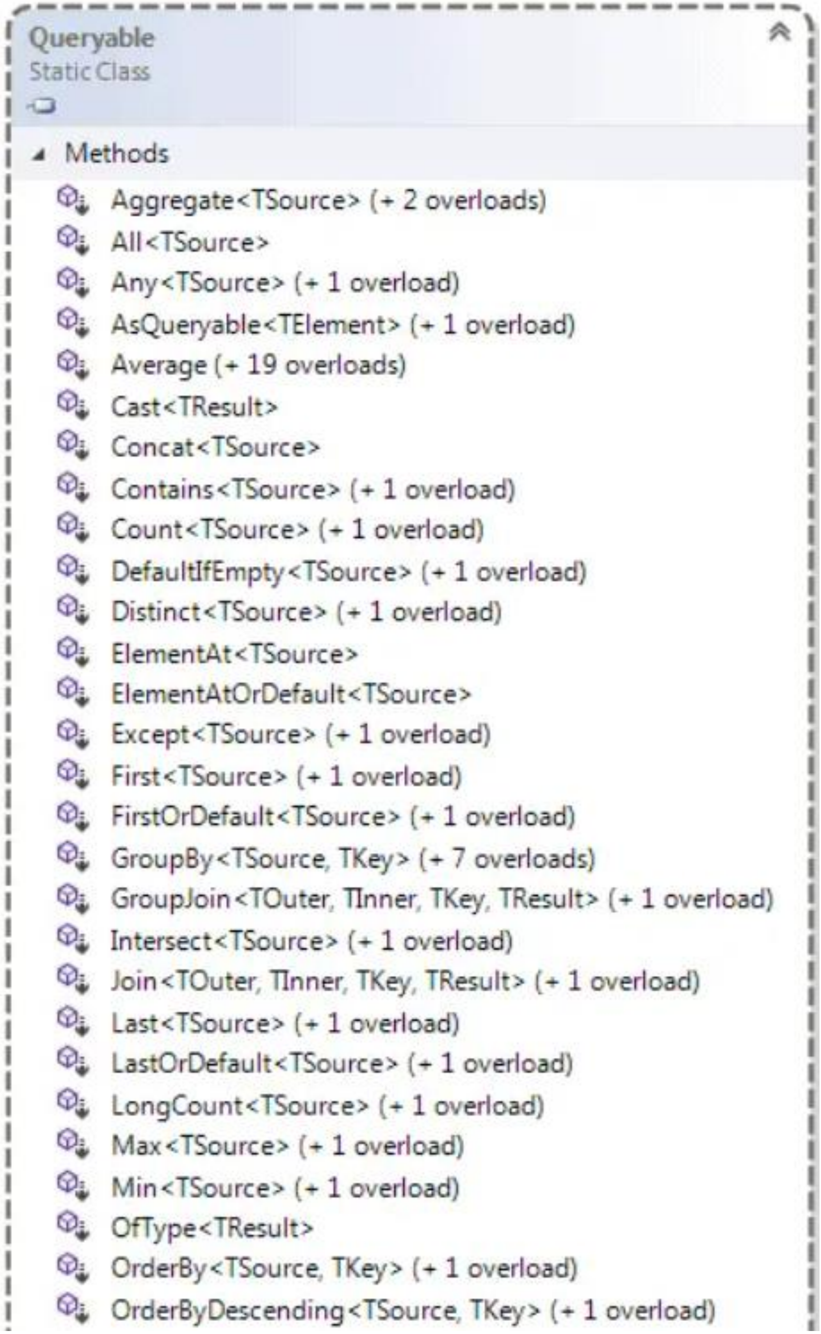
  - LastOrDefault<TSource> (+ 1 overload)
  - LongCount<TSource> (+ 1 overload)
  - Max (+ 21 overloads)
  - Min (+ 21 overloads)
  - OfType<TResult>
  - OrderBy<TSource, TKey> (+ 1 overload)
  - OrderByDescending<TSource, TKey> (+ 1 overload)
  - Range
  - Repeat<TResult>
  - Reverse<TSource>
  - Select<TSource, TResult> (+ 1 overload)
  - SelectMany<TSource, TResult> (+ 3 overloads)
  - SequenceEqual<TSource> (+ 1 overload)
  - Single<TSource> (+ 1 overload)
  - SingleOrDefault<TSource> (+ 1 overload)
  - Skip<TSource>
  - SkipWhile<TSource> (+ 1 overload)
  - Sum (+ 19 overloads)
  - Take<TSource>
  - TakeWhile<TSource> (+ 1 overload)
  - ThenBy<TSource, TKey> (+ 1 overload)
  - ThenByDescending<TSource, TKey> (+ 1 overload)
  - ToArray<TSource>
  - ToDictionary<TSource, TKey> (+ 3 overloads)
  - ToList<TSource>
  - ToLookup<TSource, TKey> (+ 3 overloads)
  - Union<TSource> (+ 1 overload)
  - Where<TSource> (+ 1 overload)
  - Zip<TFirst, TSecond, TResult>

Fuente : https://www.tutorialsteacher.com/linq/linq-api

Queryable

The Queryable class includes extension methods for classes that implement IQueryable<t> interface. The `IQueryable<T>` interface is used to provide querying capabilities against a specific data source where the type of the data is known. For example, Entity Framework api implements `IQueryable<T>` interface to support LINQ queries with underlaying databases such as MS SQL Server.

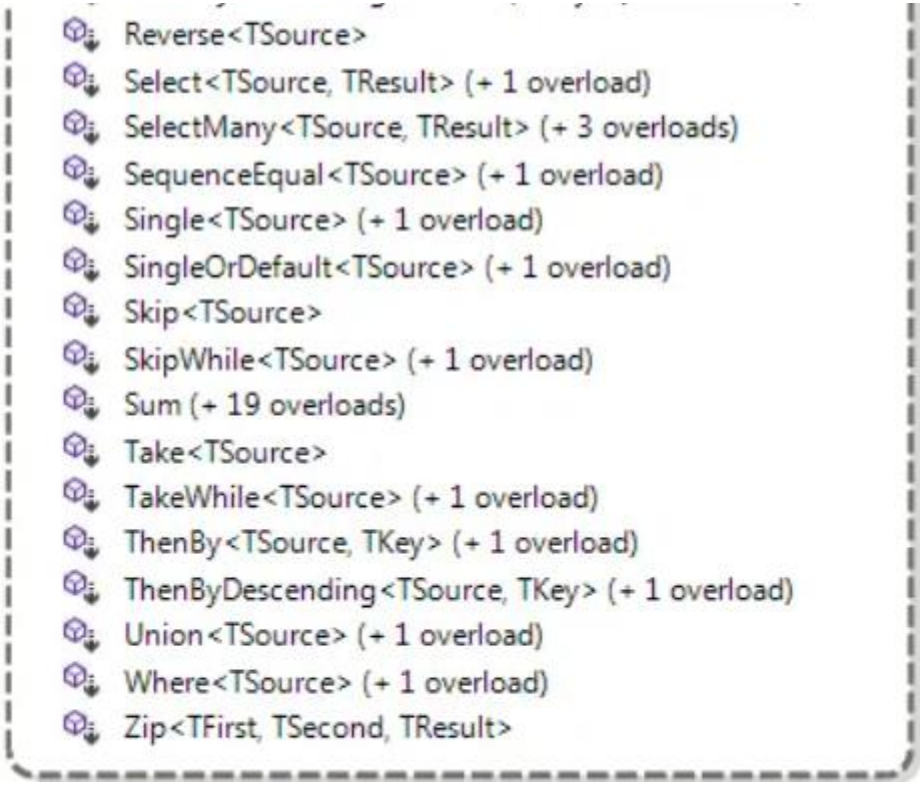




Queryable
Static Class

Methods

- Aggregate<TSource> (+ 2 overloads)
- All<TSource>
- Any<TSource> (+ 1 overload)
- AsQueryable<TElement> (+ 1 overload)
- Average (+ 19 overloads)
- Cast<TResult>
- Concat<TSource>
- Contains<TSource> (+ 1 overload)
- Count<TSource> (+ 1 overload)
- DefaultIfEmpty<TSource> (+ 1 overload)
- Distinct<TSource> (+ 1 overload)
- ElementAt<TSource>
- ElementAtOrDefault<TSource>
- Except<TSource> (+ 1 overload)
- First<TSource> (+ 1 overload)
- FirstOrDefault<TSource> (+ 1 overload)
- GroupBy<TSource, TKey> (+ 7 overloads)
- GroupJoin<TOuter, TInner, TKey, TResult> (+ 1 overload)
- Intersect<TSource> (+ 1 overload)
- Join<TOuter, TInner, TKey, TResult> (+ 1 overload)
- Last<TSource> (+ 1 overload)
- LastOrDefault<TSource> (+ 1 overload)
- LongCount<TSource> (+ 1 overload)
- Max<TSource> (+ 1 overload)
- Min<TSource> (+ 1 overload)
- OfType<TResult>
- OrderBy<TSource, TKey> (+ 1 overload)
- OrderByDescending<TSource, TKey> (+ 1 overload)



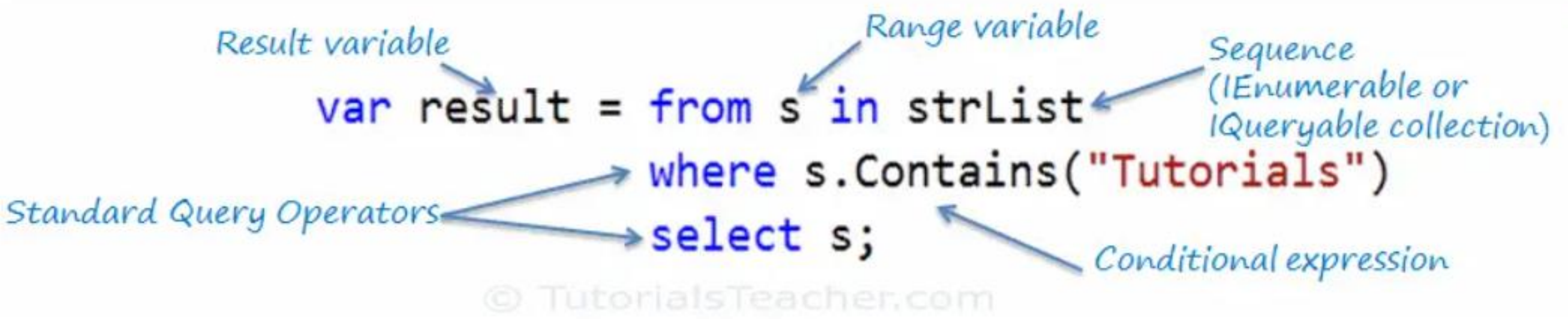
- Reverse<TSource>
- Select<TSource, TResult> (+ 1 overload)
- SelectMany<TSource, TResult> (+ 3 overloads)
- SequenceEqual<TSource> (+ 1 overload)
- Single<TSource> (+ 1 overload)
- SingleOrDefault<TSource> (+ 1 overload)
- Skip<TSource>
- SkipWhile<TSource> (+ 1 overload)
- Sum (+ 19 overloads)
- Take<TSource>
- TakeWhile<TSource> (+ 1 overload)
- ThenBy<TSource, TKey> (+ 1 overload)
- ThenByDescending<TSource, TKey> (+ 1 overload)
- Union<TSource> (+ 1 overload)
- Where<TSource> (+ 1 overload)
- Zip<TFirst, TSecond, TResult>

LINQ Query Syntax

There are two basic ways to write a LINQ query to IEnumerable collection or IQueryable data sources.

1. Query Syntax or Query Expression Syntax
2. Method Syntax or Method Extension Syntax or Fluent

```
from <range variable> in <IEnumerable<T> or IQueryable<T> Collection>

<Standard Query Operators> <lambda expression>

<select or groupBy operator> <result formation>
```



```
var result = from s in strList
             where s.Contains("Tutorials")
             select s;
```

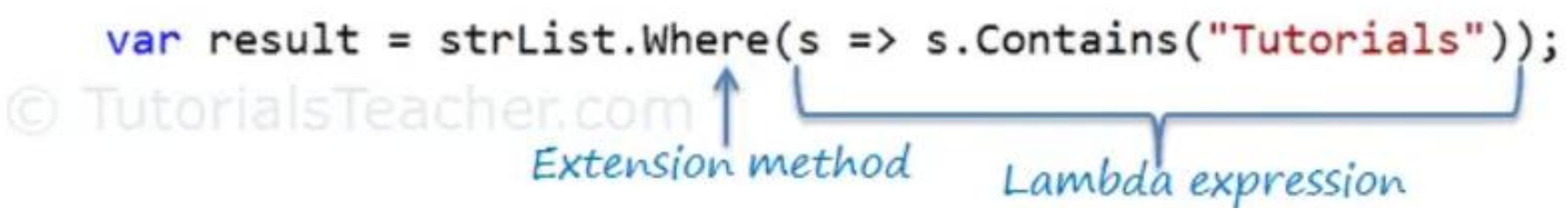
LINQ Query Syntax

Fuente : https://www.tutorialsteacher.com/linq/linq-query-syntax

LINQ Method Syntax

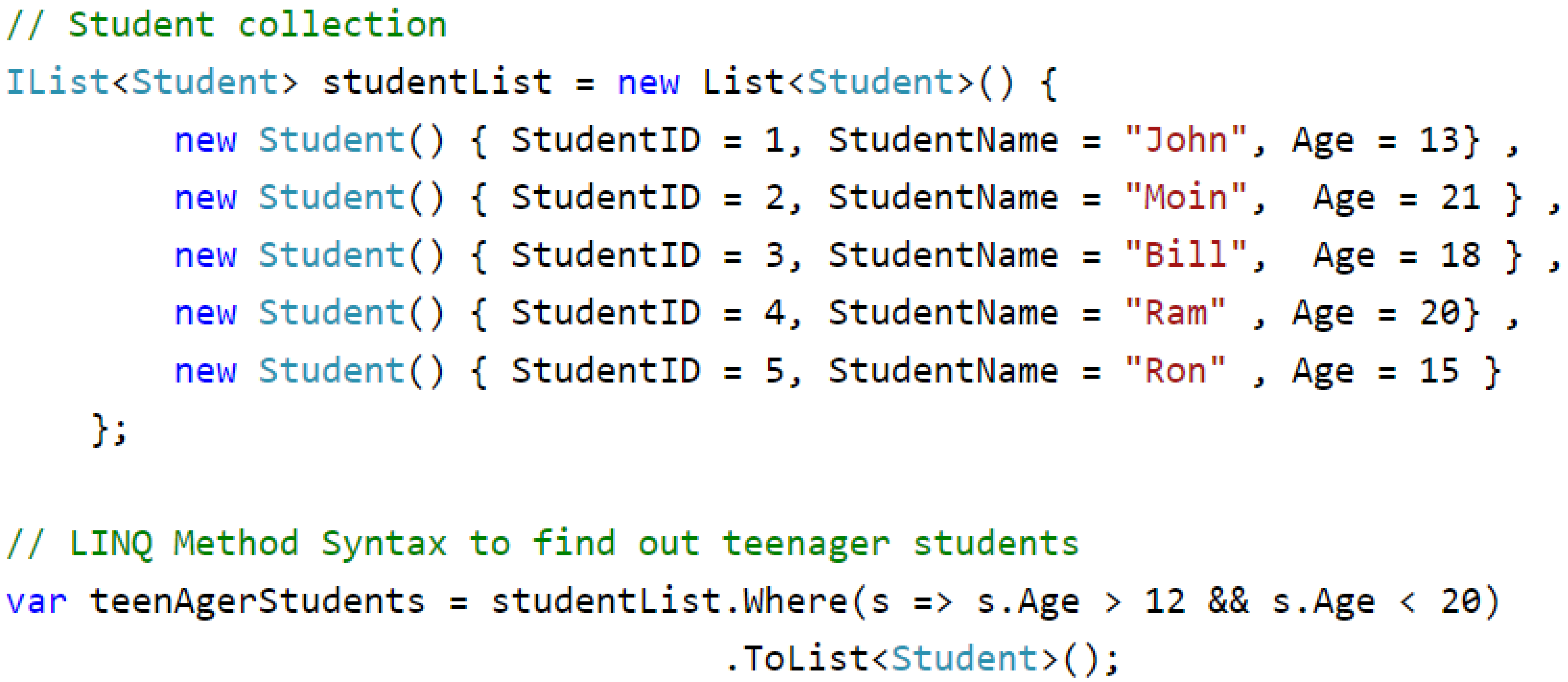
Method syntax (also known as fluent syntax) uses extension methods included in the Enumerable or Queryable static class, similar to how you would call the extension method of any class.

```
// string collection
IList<string> stringList = new List<string>() {
    "C# Tutorials",
    "VB.NET Tutorials",
    "Learn C++",
    "MVC Tutorials" ,
    "Java"
};

// LINQ Method Syntax
var result = stringList.Where(s => s.Contains("Tutorials"));
```



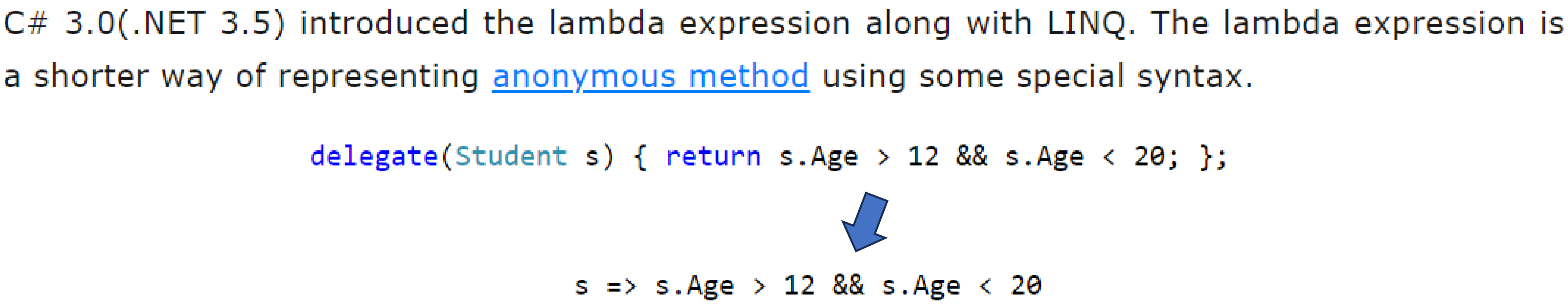
```
var result = strList.Where(s => s.Contains("Tutorials"));
```

Fuente : https://www.tutorialsteacher.com/linq/linq-method-syntax



```
// Student collection
IList<Student> studentList = new List<Student>() {
        new Student() { StudentID = 1, StudentName = "John", Age = 13} ,
        new Student() { StudentID = 2, StudentName = "Moin",  Age = 21 } ,
        new Student() { StudentID = 3, StudentName = "Bill",  Age = 18 } ,
        new Student() { StudentID = 4, StudentName = "Ram" , Age = 20} ,
        new Student() { StudentID = 5, StudentName = "Ron" , Age = 15 }
    };

// LINQ Method Syntax to find out teenager students
var teenAgerStudents = studentList.Where(s => s.Age > 12 && s.Age < 20)
                                  .ToList<Student>();
```

Anatomy of the Lambda Expression



C# 3.0(.NET 3.5) introduced the lambda expression along with LINQ. The lambda expression is a shorter way of representing anonymous method using some special syntax.

```
delegate(Student s) { return s.Age > 12 && s.Age < 20; };
```

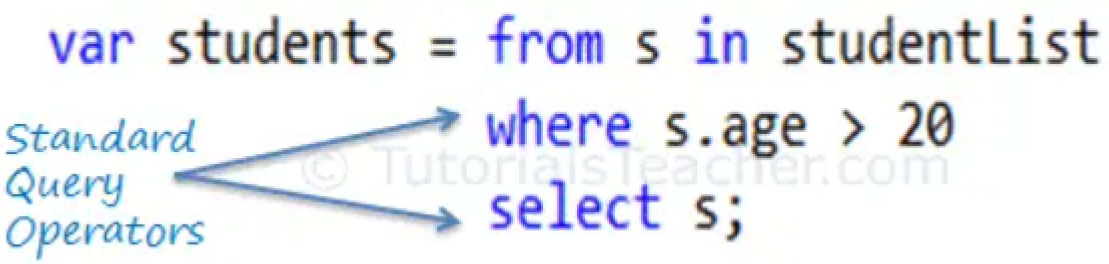
```
s => s.Age > 12 && s.Age < 20
```

https://www.tutorialsteacher.com/linq/linq-lambda-expression

Standard Query Operators

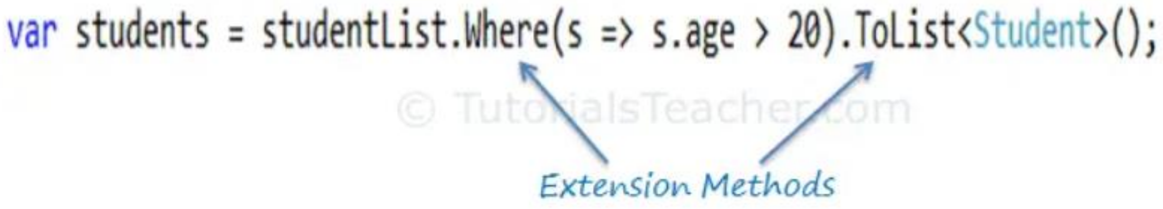
Standard Query Operators in LINQ are actually extension methods for the `IEnumerable<T> and IQueryable<T>` types. They are defined in the `System.Linq.Enumerable` and `System.Linq.Queryable` classes. There are over 50 standard query operators available in LINQ that provide different functionalities like filtering, sorting, grouping, aggregation, concatenation, etc.

Standard Query Operators in Query Syntax

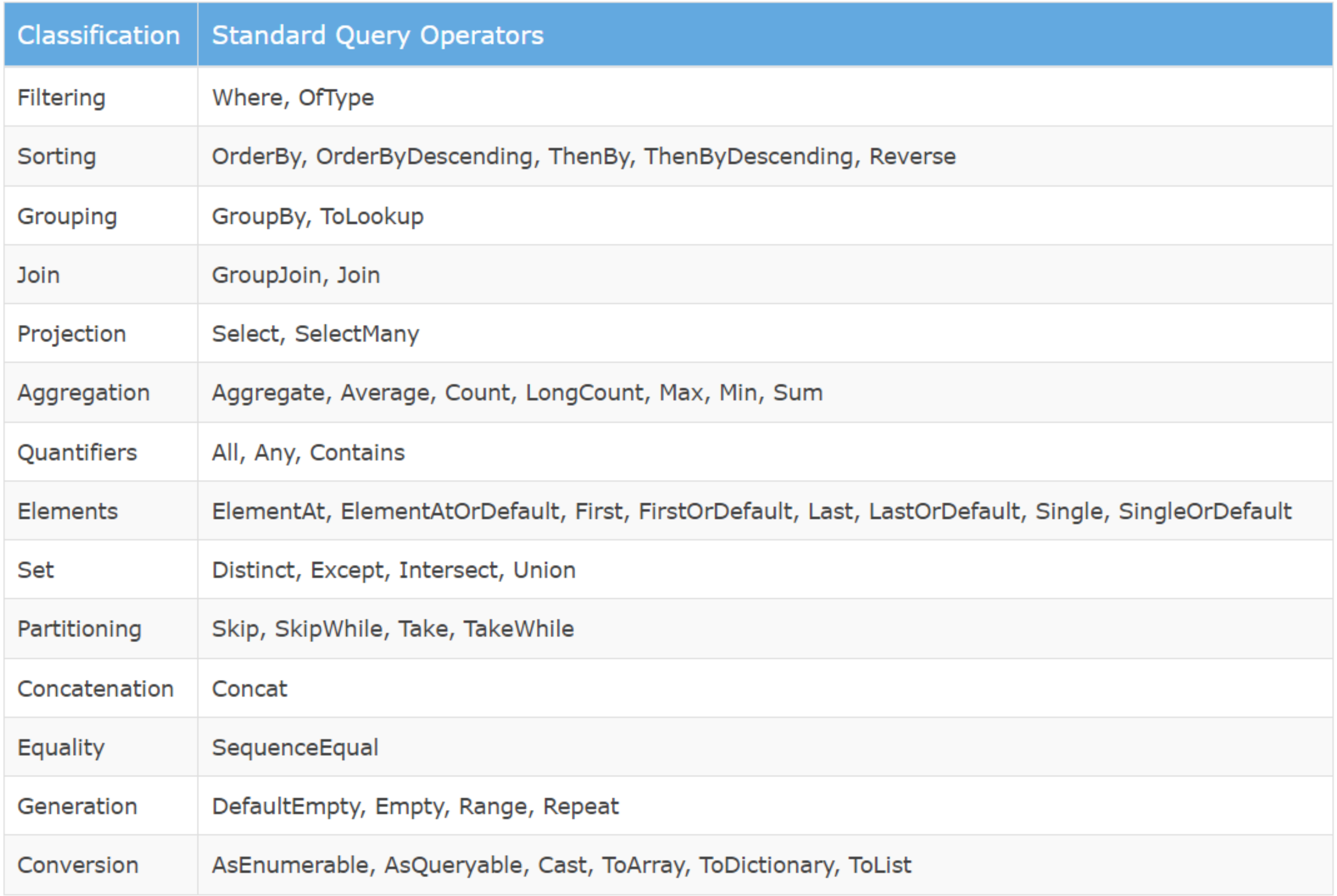


```
var students = from s in studentList
               where s.age > 20
               select s;
```

Standard Query Operators in Method Syntax

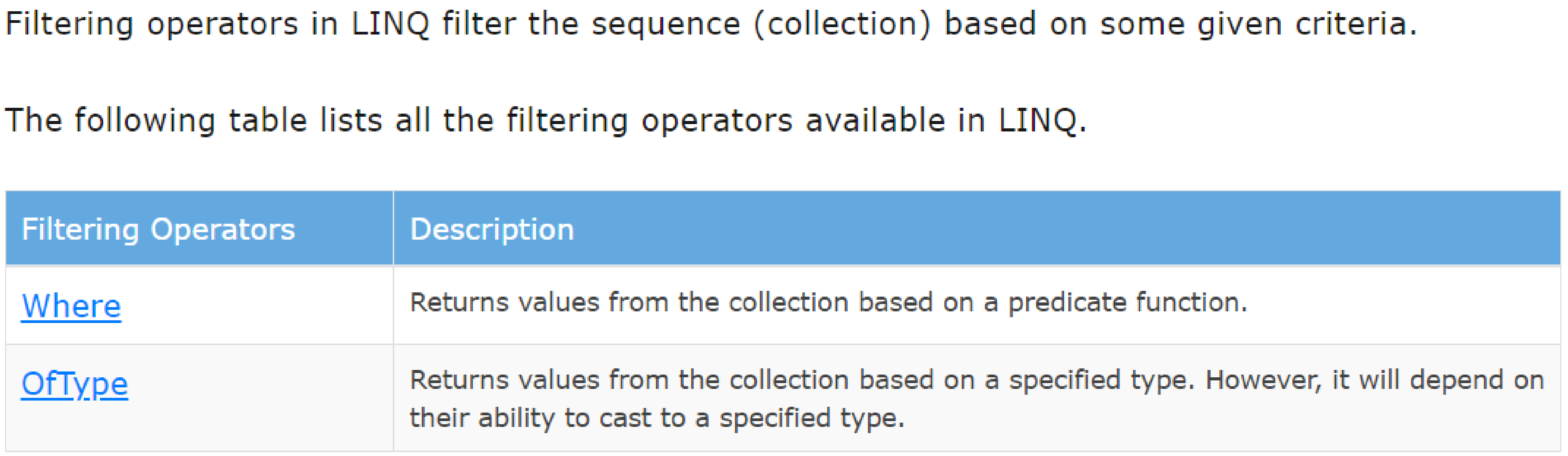


```
var students = studentList.Where(s => s.age > 20).ToList<Student>();
```



Classification	Standard Query Operators
Filtering	Where, OfType
Sorting	OrderBy, OrderByDescending, ThenBy, ThenByDescending, Reverse
Grouping	GroupBy, ToLookup
Join	GroupJoin, Join
Projection	Select, SelectMany
Aggregation	Aggregate, Average, Count, LongCount, Max, Min, Sum
Quantifiers	All, Any, Contains
Elements	ElementAt, ElementAtOrDefault, First, FirstOrDefault, Last, LastOrDefault, Single, SingleOrDefault
Set	Distinct, Except, Intersect, Union
Partitioning	Skip, SkipWhile, Take, TakeWhile
Concatenation	Concat
Equality	SequenceEqual
Generation	DefaultEmpty, Empty, Range, Repeat
Conversion	AsEnumerable, AsQueryable, Cast, ToArray, ToDictionary, ToList

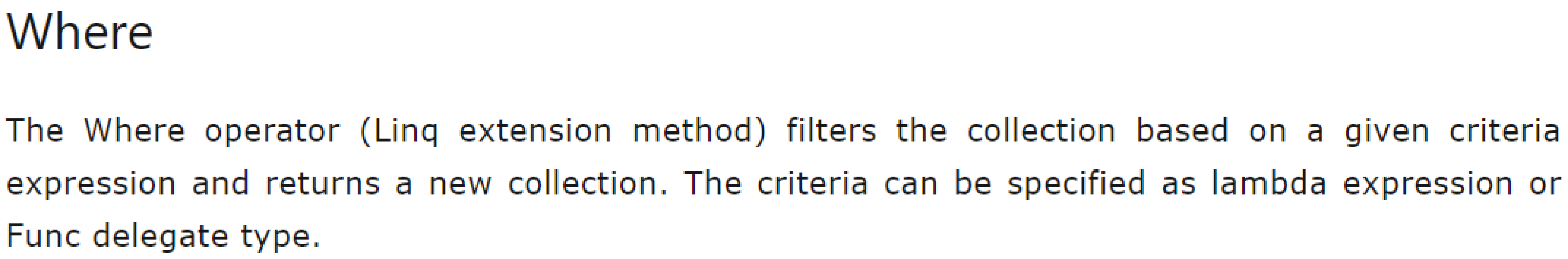
Filtering Operator - Where



Filtering operators in LINQ filter the sequence (collection) based on some given criteria.

The following table lists all the filtering operators available in LINQ.

Filtering Operators	Description
Where	Returns values from the collection based on a predicate function.
OfType	Returns values from the collection based on a specified type. However, it will depend on their ability to cast to a specified type.

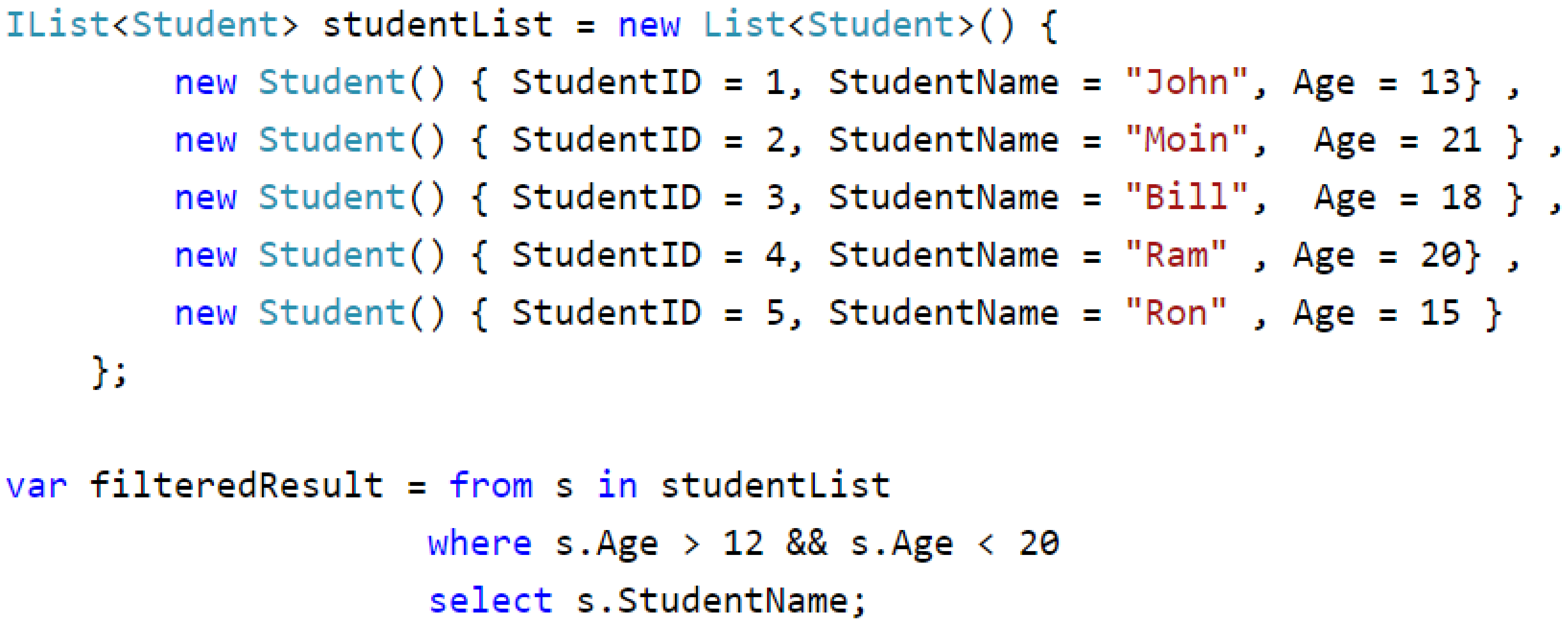


Where

The Where operator (Linq extension method) filters the collection based on a given criteria expression and returns a new collection. The criteria can be specified as lambda expression or Func delegate type.

https://www.tutorialsteacher.com/linq/linq-filtering-operators-where

```
public static IEnumerable<TSource> Where<TSource>(this IEnumerable<TSource> source,
                                                  Func<TSource, bool> predicate);

public static IEnumerable<TSource> Where<TSource>(this IEnumerable<TSource> source,
                                                  Func<TSource, int, bool> predicate);
```



```
IList<Student> studentList = new List<Student>() {
        new Student() { StudentID = 1, StudentName = "John", Age = 13} ,
        new Student() { StudentID = 2, StudentName = "Moin",  Age = 21 } ,
        new Student() { StudentID = 3, StudentName = "Bill",  Age = 18 } ,
        new Student() { StudentID = 4, StudentName = "Ram" , Age = 20} ,
        new Student() { StudentID = 5, StudentName = "Ron" , Age = 15 }
    };

var filteredResult = from s in studentList
                    where s.Age > 12 && s.Age < 20
                    select s.StudentName;
```

Where extension method in Method Syntax

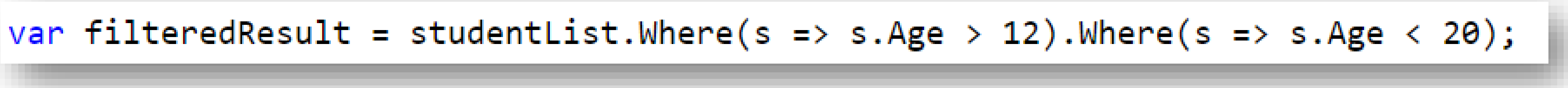
Unlike the query syntax, you need to pass whole lambda expression as a predicate function instead of just body expression in LINQ method syntax.

```
var filteredResult = studentList.Where(s => s.Age > 12 && s.Age < 20);
```

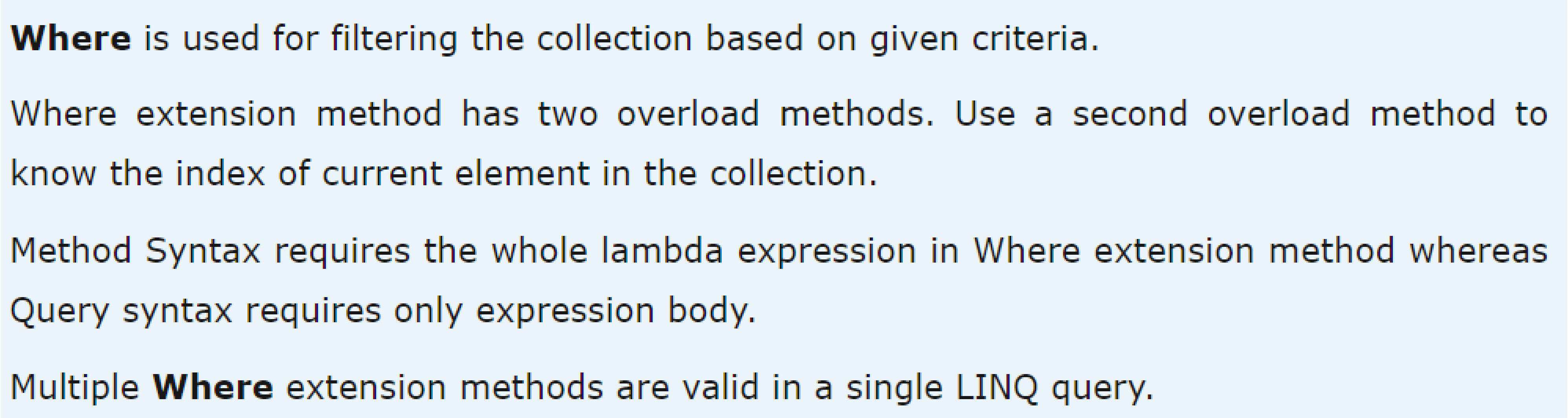
```
IList<Student> studentList = new List<Student>() {
        new Student() { StudentID = 1, StudentName = "John", Age = 18 } ,
        new Student() { StudentID = 2, StudentName = "Steve",  Age = 15 } ,
        new Student() { StudentID = 3, StudentName = "Bill",  Age = 25 } ,
        new Student() { StudentID = 4, StudentName = "Ram" , Age = 20 } ,
        new Student() { StudentID = 5, StudentName = "Ron" , Age = 19 }
    };

var filteredResult = studentList.Where((s, i) => {
            if(i % 2 ==  0) // if it is even element
                return true;

        return false;
    });

foreach (var std in filteredResult)
        Console.WriteLine(std.StudentName);
```

Multiple Where clause

You can call the Where() extension method more than one time in a single LINQ query.



```
var filteredResult = studentList.Where(s => s.Age > 12).Where(s => s.Age < 20);
```



Where is used for filtering the collection based on given criteria.

Where extension method has two overload methods. Use a second overload method to know the index of current element in the collection.

Method Syntax requires the whole lambda expression in Where extension method whereas Query syntax requires only expression body.

Multiple **Where** extension methods are valid in a single LINQ query.

Fuente : https://www.tutorialsteacher.com/linq/linq-filtering-operators-where

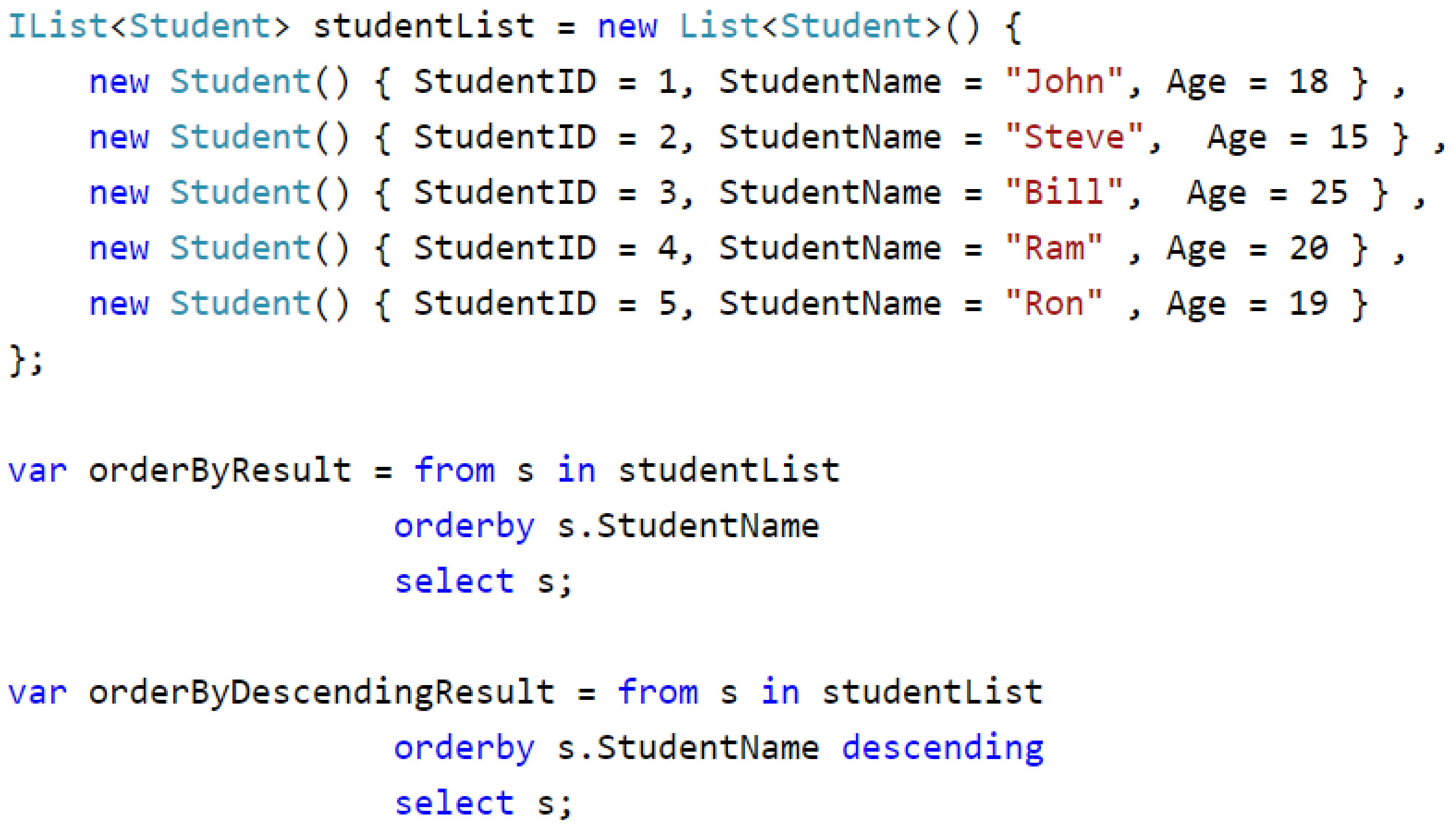
Sorting Operators: OrderBy & OrderByDescending

A sorting operator arranges the elements of the collection in ascending or descending order. LINQ includes following sorting operators.

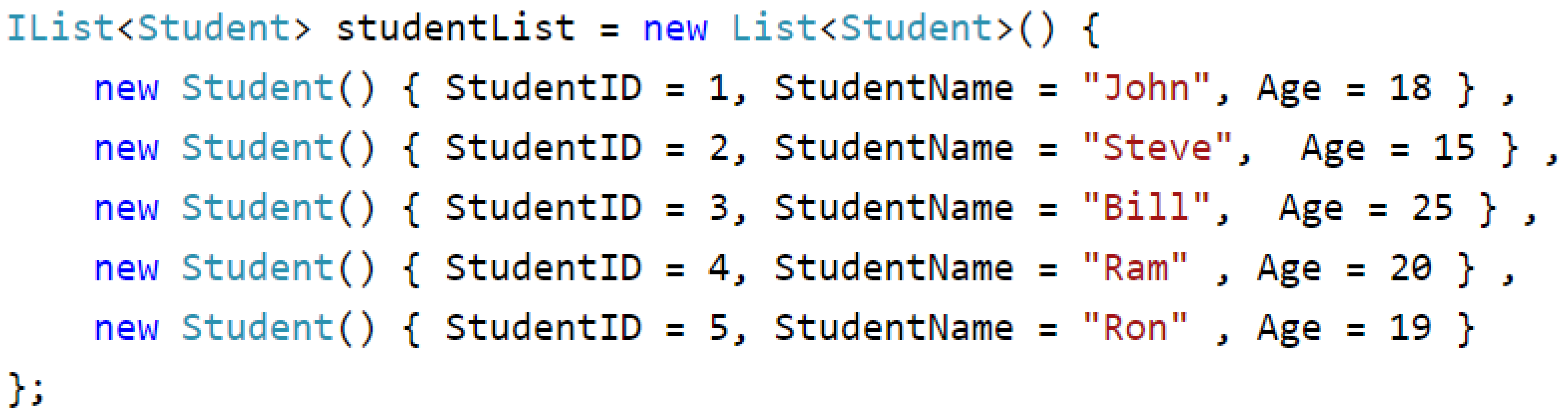
Sorting Operator	Description
OrderBy	Sorts the elements in the collection based on specified fields in ascending or decending order.
OrderByDescending	Sorts the collection based on specified fields in descending order. Only valid in method syntax.
ThenBy	Only valid in method syntax. Used for second level sorting in ascending order.
ThenByDescending	Only valid in method syntax. Used for second level sorting in descending order.
Reverse	Only valid in method syntax. Sorts the collection in reverse order.

OrderBy

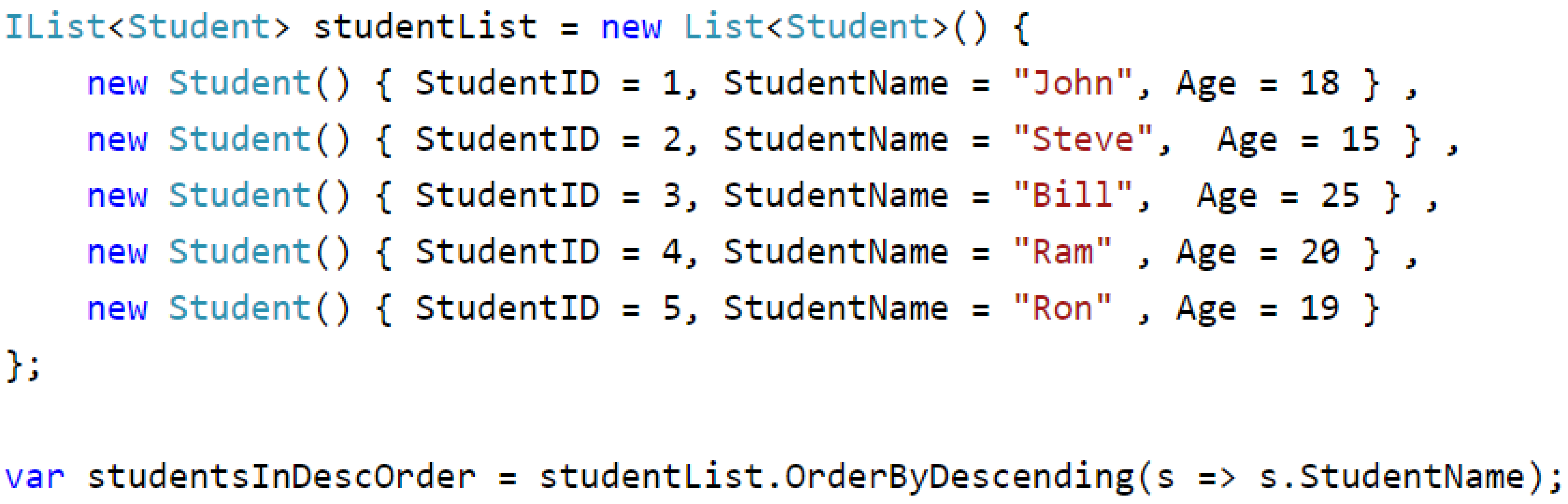
OrderBy sorts the values of a collection in ascending or descending order. It sorts the collection in ascending order by default because `ascending` keyword is optional here. Use descending keyword to sort collection in descending order.



```
IList<Student> studentList = new List<Student>() {
    new Student() { StudentID = 1, StudentName = "John", Age = 18 } ,
    new Student() { StudentID = 2, StudentName = "Steve",  Age = 15 } ,
    new Student() { StudentID = 3, StudentName = "Bill",  Age = 25 } ,
    new Student() { StudentID = 4, StudentName = "Ram" , Age = 20 } ,
    new Student() { StudentID = 5, StudentName = "Ron" , Age = 19 }
};

var orderByResult = from s in studentList
                   orderby s.StudentName
                   select s;

var orderByDescendingResult = from s in studentList
                   orderby s.StudentName descending
                   select s;
```



```
IList<Student> studentList = new List<Student>() {
    new Student() { StudentID = 1, StudentName = "John", Age = 18 } ,
    new Student() { StudentID = 2, StudentName = "Steve",  Age = 15 } ,
    new Student() { StudentID = 3, StudentName = "Bill",  Age = 25 } ,
    new Student() { StudentID = 4, StudentName = "Ram" , Age = 20 } ,
    new Student() { StudentID = 5, StudentName = "Ron" , Age = 19 }
};

var studentsInAscOrder = studentList.OrderBy(s => s.StudentName);
```



```
IList<Student> studentList = new List<Student>() {
    new Student() { StudentID = 1, StudentName = "John", Age = 18 } ,
    new Student() { StudentID = 2, StudentName = "Steve",  Age = 15 } ,
    new Student() { StudentID = 3, StudentName = "Bill",  Age = 25 } ,
    new Student() { StudentID = 4, StudentName = "Ram" , Age = 20 } ,
    new Student() { StudentID = 5, StudentName = "Ron" , Age = 19 }
};

var studentsInDescOrder = studentList.OrderByDescending(s => s.StudentName);
```

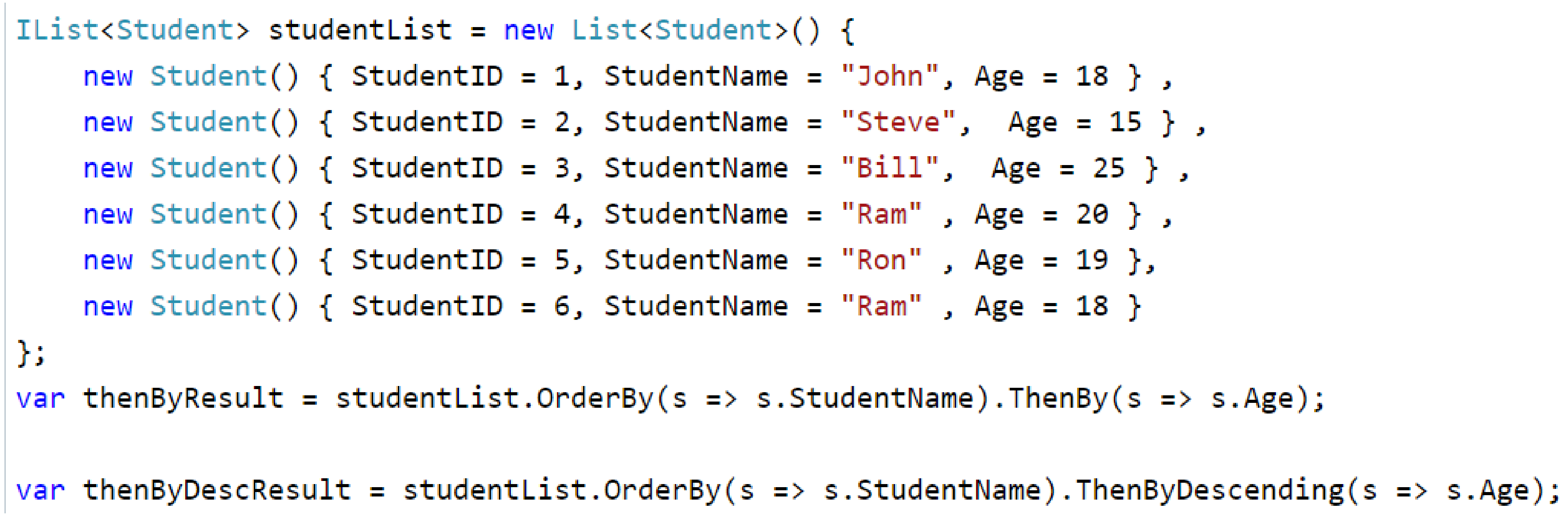
Multiple Sorting

```
IList<Student> studentList = new List<Student>() {
    new Student() { StudentID = 1, StudentName = "John", Age = 18 } ,
    new Student() { StudentID = 2, StudentName = "Steve",  Age = 15 } ,
    new Student() { StudentID = 3, StudentName = "Bill",  Age = 25 } ,
    new Student() { StudentID = 4, StudentName = "Ram" , Age = 20 } ,
    new Student() { StudentID = 5, StudentName = "Ron" , Age = 19 },
    new Student() { StudentID = 6, StudentName = "Ram" , Age = 18 }
};

var orderByResult = from s in studentList
                   orderby s.StudentName, s.Age
                   select new { s.StudentName, s.Age };
```

Sorting Operators: ThenBy & ThenByDescending

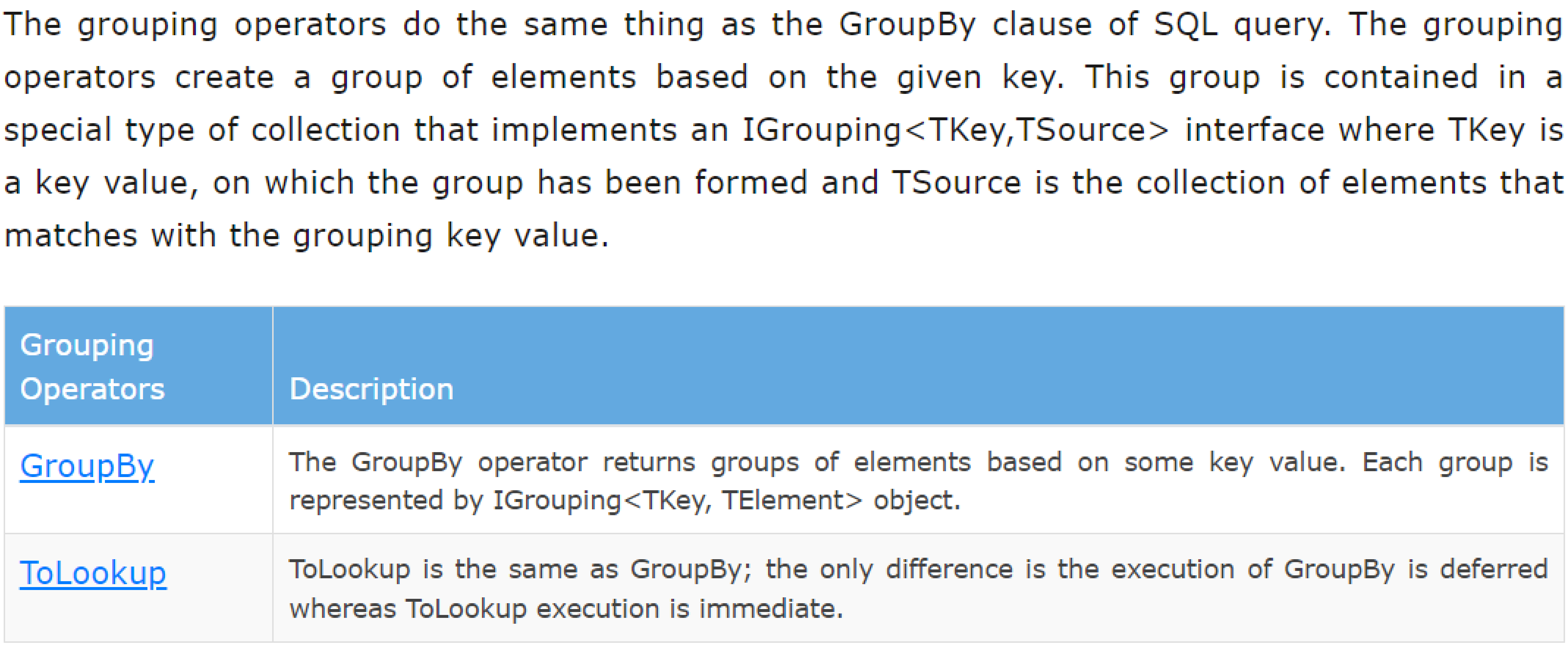
The `ThenBy` and `ThenByDescending` extension methods are used for sorting on multiple fields.

The `OrderBy()` method sorts the collection in ascending order based on specified field. Use ThenBy() method after OrderBy to sort the collection on another field in ascending order. Linq will first sort the collection based on primary field which is specified by OrderBy method and then sort the resulted collection in ascending order again based on secondary field specified by `ThenBy` method.



```
IList<Student> studentList = new List<Student>() {
    new Student() { StudentID = 1, StudentName = "John", Age = 18 } ,
    new Student() { StudentID = 2, StudentName = "Steve",  Age = 15 } ,
    new Student() { StudentID = 3, StudentName = "Bill",  Age = 25 } ,
    new Student() { StudentID = 4, StudentName = "Ram" , Age = 20 } ,
    new Student() { StudentID = 5, StudentName = "Ron" , Age = 19 },
    new Student() { StudentID = 6, StudentName = "Ram" , Age = 18 }
};
var thenByResult = studentList.OrderBy(s => s.StudentName).ThenBy(s => s.Age);

var thenByDescResult = studentList.OrderBy(s => s.StudentName).ThenByDescending(s => s.Age);
```

Grouping Operators: GroupBy & ToLookup



The grouping operators do the same thing as the GroupBy clause of SQL query. The grouping operators create a group of elements based on the given key. This group is contained in a special type of collection that implements an IGrouping<TKey,TSource> interface where TKey is a key value, on which the group has been formed and TSource is the collection of elements that matches with the grouping key value.

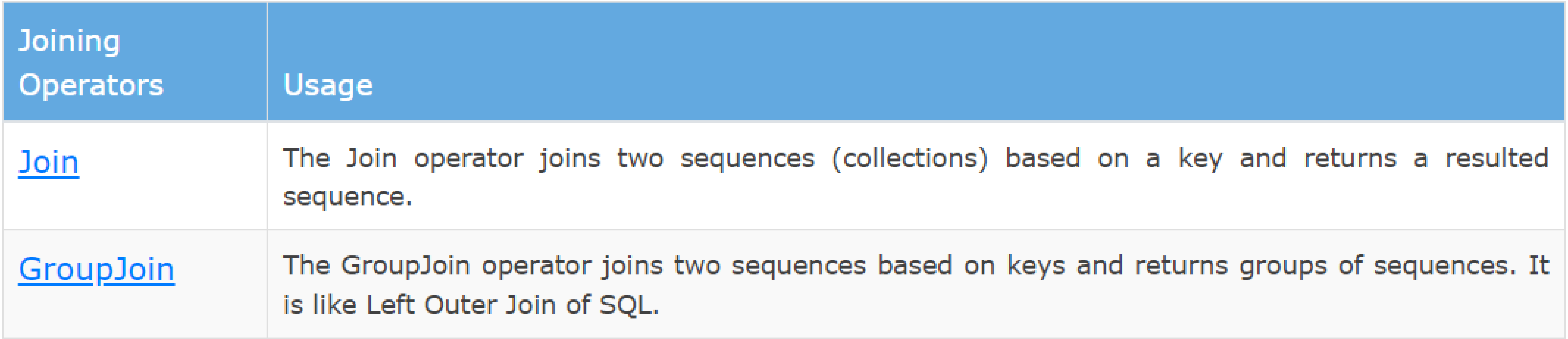
Grouping Operators	Description
GroupBy	The GroupBy operator returns groups of elements based on some key value. Each group is represented by IGrouping<TKey, TElement> object.
ToLookup	ToLookup is the same as GroupBy; the only difference is the execution of GroupBy is deferred whereas ToLookup execution is immediate.

LINQ Microsoft .NET

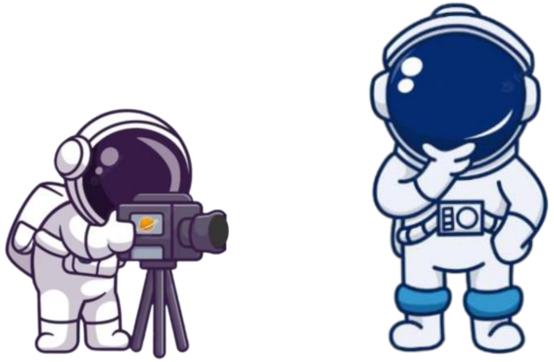
Ver Megadocumento CSharp

Joining Operator: Join

The joining operators joins the two sequences (collections) and produce a result.

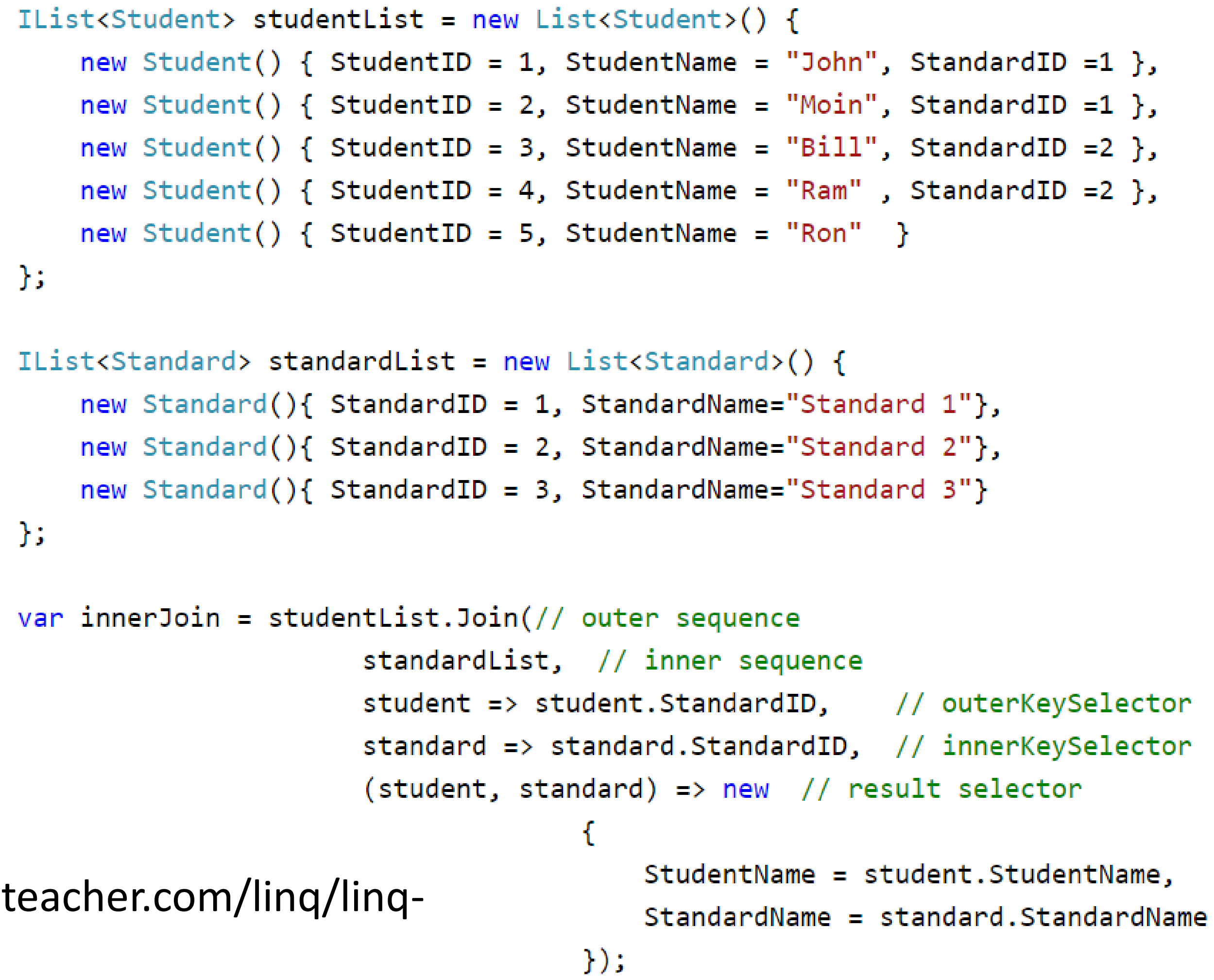


Joining Operators	Usage
Join	The Join operator joins two sequences (collections) based on a key and returns a resulted sequence.
GroupJoin	The GroupJoin operator joins two sequences based on keys and returns groups of sequences. It is like Left Outer Join of SQL.



Join

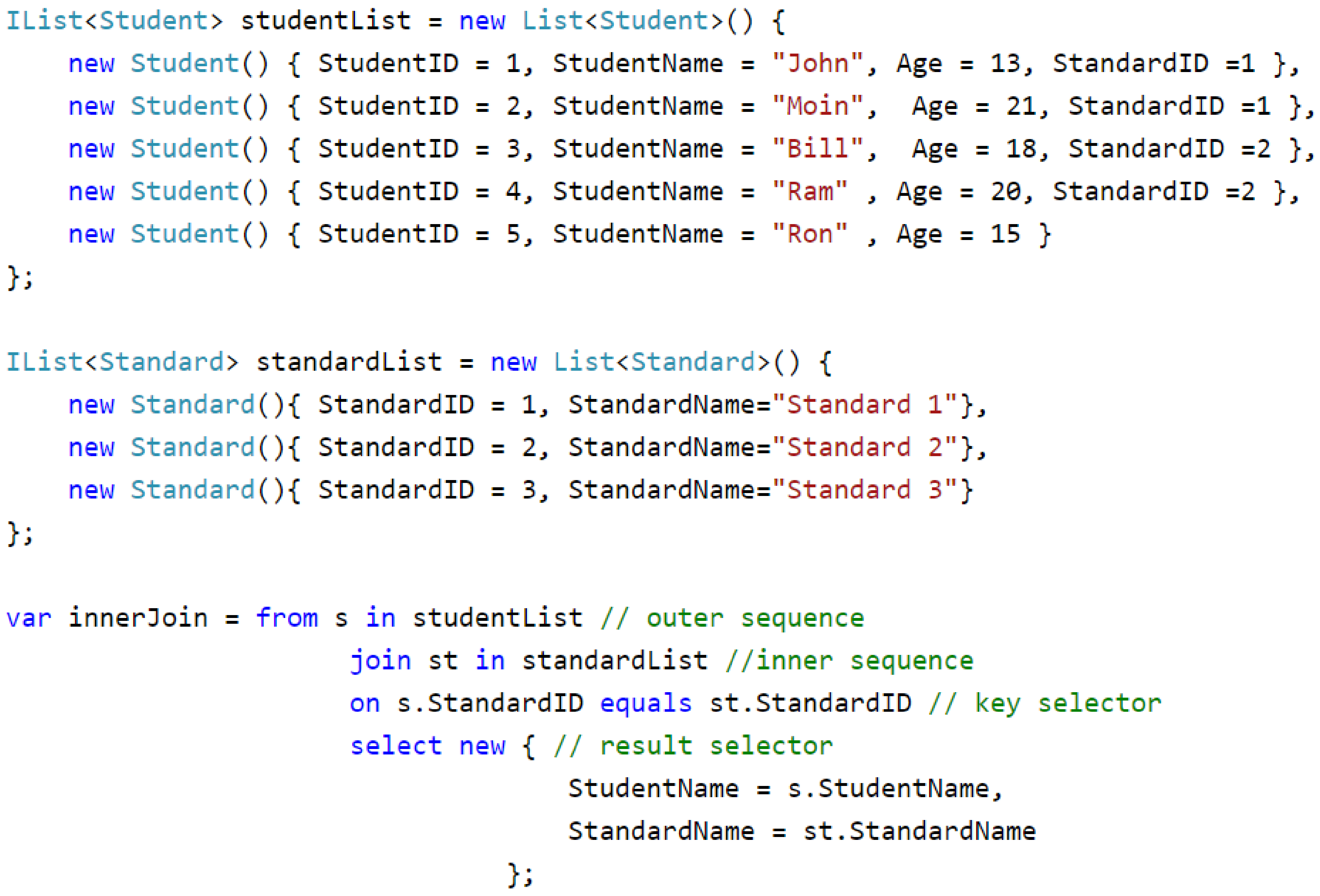
The Join operator operates on two collections, inner collection & outer collection. It returns a new collection that contains elements from both the collections which satisfies specified expression. It is the same as **inner join** of SQL.



```
IList<Student> studentList = new List<Student>() {
    new Student() { StudentID = 1, StudentName = "John", StandardID =1 },
    new Student() { StudentID = 2, StudentName = "Moin", StandardID =1 },
    new Student() { StudentID = 3, StudentName = "Bill", StandardID =2 },
    new Student() { StudentID = 4, StudentName = "Ram" , StandardID =2 },
    new Student() { StudentID = 5, StudentName = "Ron"  }
};

IList<Standard> standardList = new List<Standard>() {
    new Standard(){ StandardID = 1, StandardName="Standard 1"},
    new Standard(){ StandardID = 2, StandardName="Standard 2"},
    new Standard(){ StandardID = 3, StandardName="Standard 3"}
};

var innerJoin = studentList.Join(// outer sequence
                      standardList,  // inner sequence
                      student => student.StandardID,    // outerKeySelector
                      standard => standard.StandardID,  // innerKeySelector
                      (student, standard) => new  // result selector
                                    {
                                        StudentName = student.StudentName,
                                        StandardName = standard.StandardName
                                    });
```

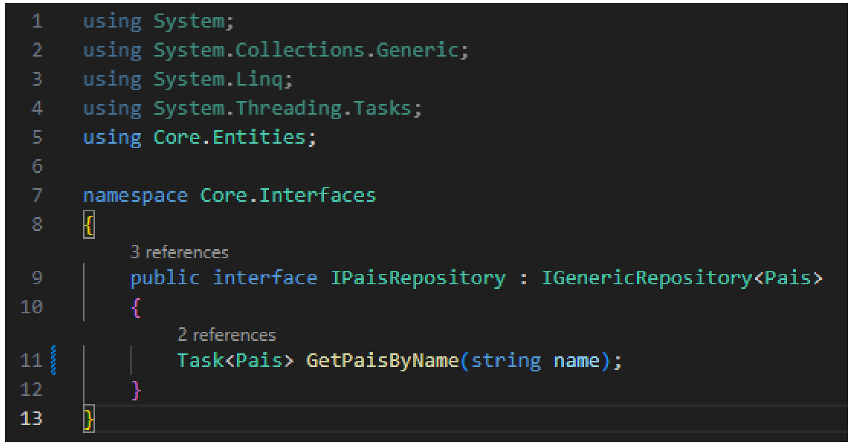
Fuente : https://www.tutorialsteacher.com/linq/linq-joining-operator-join

Join in Query Syntax

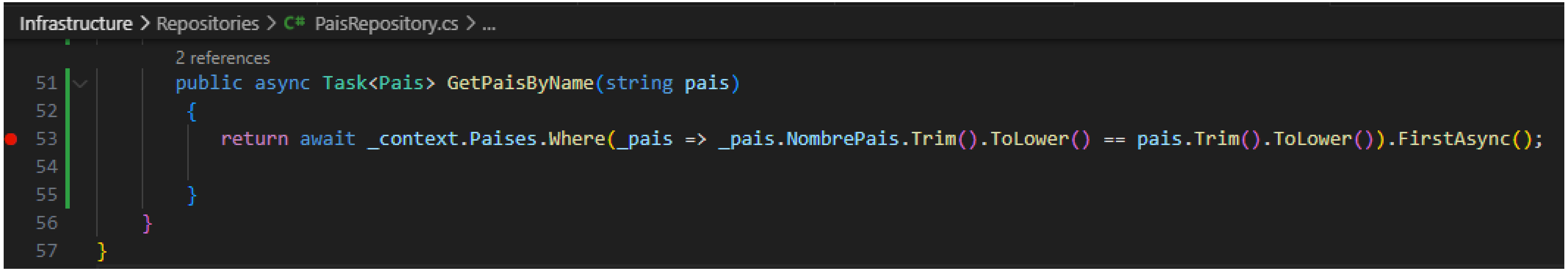


```
IList<Student> studentList = new List<Student>() {
    new Student() { StudentID = 1, StudentName = "John", Age = 13, StandardID =1 },
    new Student() { StudentID = 2, StudentName = "Moin",  Age = 21, StandardID =1 },
    new Student() { StudentID = 3, StudentName = "Bill",  Age = 18, StandardID =2 },
    new Student() { StudentID = 4, StudentName = "Ram" , Age = 20, StandardID =2 },
    new Student() { StudentID = 5, StudentName = "Ron" , Age = 15 }
};

IList<Standard> standardList = new List<Standard>() {
    new Standard(){ StandardID = 1, StandardName="Standard 1"},
    new Standard(){ StandardID = 2, StandardName="Standard 2"},
    new Standard(){ StandardID = 3, StandardName="Standard 3"}
};

var innerJoin = from s in studentList // outer sequence
                      join st in standardList //inner sequence
                      on s.StandardID equals st.StandardID // key selector
                      select new { // result selector
                                    StudentName = s.StudentName,
                                    StandardName = st.StandardName
                                };
```

Creando Consultas Usando Entity Framework

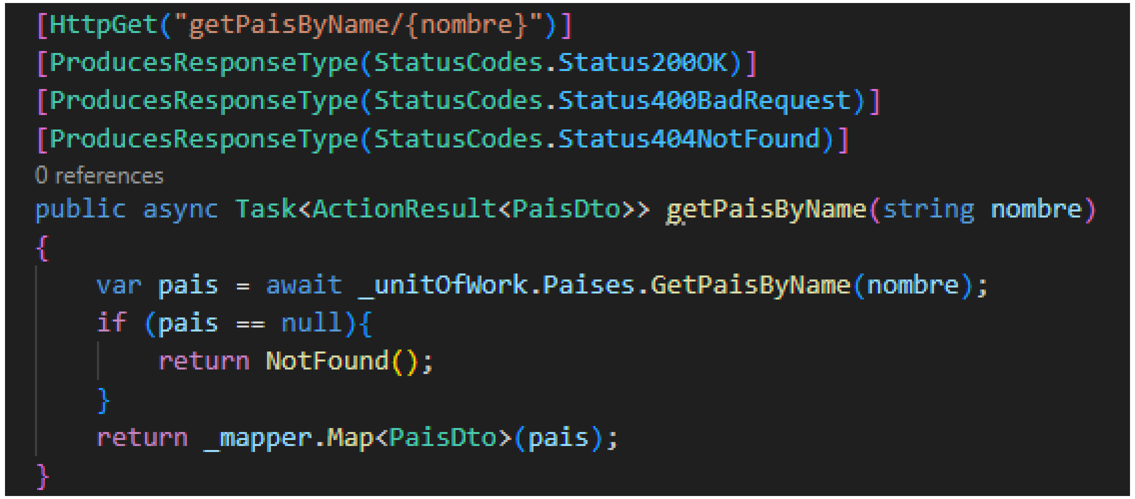


```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Core.Entities;

namespace Core.Interfaces
{
    3 references
    public interface IPaisRepository : IGenericRepository<Pais>
    {
        2 references
        Task<Pais> GetPaisByName(string name);
    }
}
```



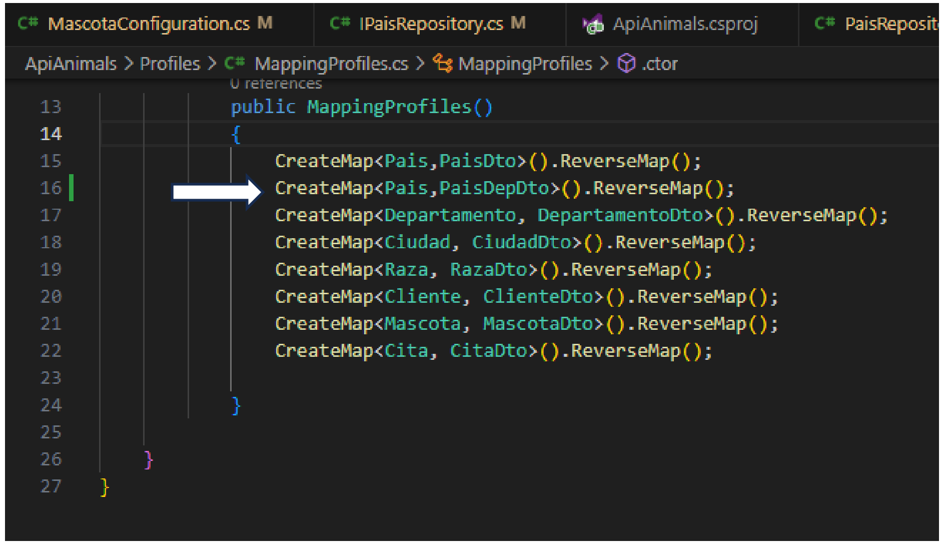
Infrastructure > Repositories > C# PaisRepository.cs > ...

```
        2 references
        public async Task<Pais> GetPaisByName(string pais)
         {
            return await _context.Paises.Where(_pais => _pais.NombrePais.Trim().ToLower() == pais.Trim().ToLower()).FirstAsync();

         }
    }
}
```

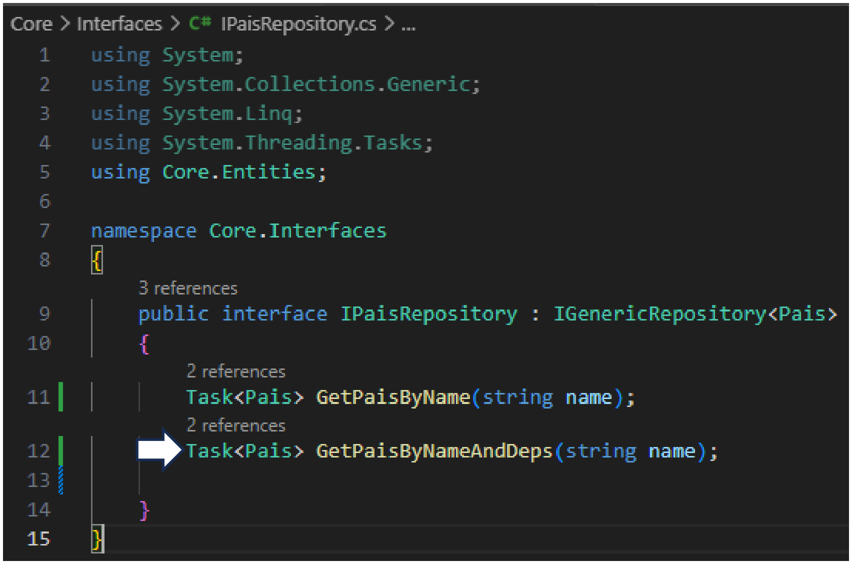


```
[HttpGet("getPaisByName/{nombre}")]
[ProducesResponseType(StatusCodes.Status200OK)]
[ProducesResponseType(StatusCodes.Status400BadRequest)]
[ProducesResponseType(StatusCodes.Status404NotFound)]
0 references
public async Task<ActionResult<PaisDto>> getPaisByName(string nombre)
{
    var pais = await _unitOfWork.Paises.GetPaisByName(nombre);
    if (pais == null){
        return NotFound();
    }
    return _mapper.Map<PaisDto>(pais);
}
```

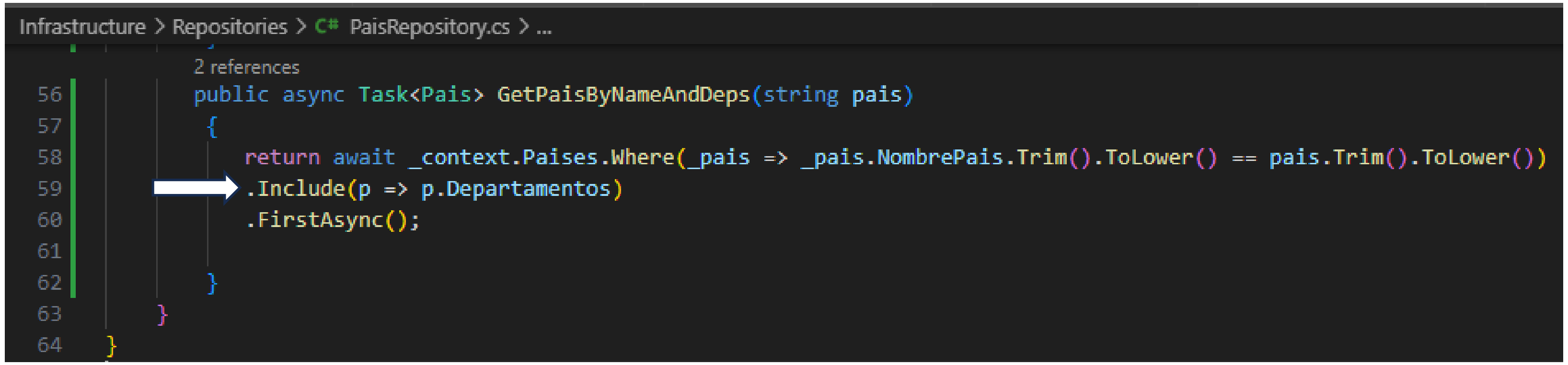
Buscar un país por el nombre.


MascotaConfiguration.cs M
IPaisRepository.cs M
ApiAnimals.csproj
PaisReposit
ApiAnimals > Profiles > MappingProfiles.cs > MappingProfiles > .ctor

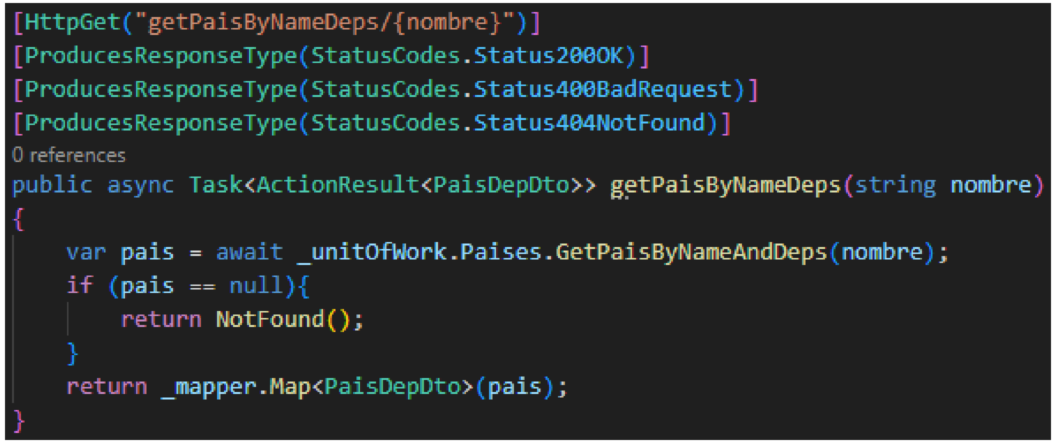
```
            0 references
            public MappingProfiles()
            {
                CreateMap<Pais,PaisDto>().ReverseMap();
                CreateMap<Pais,PaisDepDto>().ReverseMap();
                CreateMap<Departamento, DepartamentoDto>().ReverseMap();
                CreateMap<Ciudad, CiudadDto>().ReverseMap();
                CreateMap<Raza, RazaDto>().ReverseMap();
                CreateMap<Cliente, ClienteDto>().ReverseMap();
                CreateMap<Mascota, MascotaDto>().ReverseMap();
                CreateMap<Cita, CitaDto>().ReverseMap();

            }

    }
}
```



```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Core.Entities;

namespace Core.Interfaces
{
    3 references
    public interface IPaisRepository : IGenericRepository<Pais>
    {
        2 references
        Task<Pais> GetPaisByName(string name);
        2 references
        Task<Pais> GetPaisByNameAndDeps(string name);

    }
}
```



```
       2 references
       public async Task<Pais> GetPaisByNameAndDeps(string pais)
        {
           return await _context.Paises.Where(_pais => _pais.NombrePais.Trim().ToLower() == pais.Trim().ToLower())
           .Include(p => p.Departamentos)
           .FirstAsync();

        }
    }
}
```



```
[HttpGet("getPaisByNameDeps/{nombre}")]
[ProducesResponseType(StatusCodes.Status200OK)]
[ProducesResponseType(StatusCodes.Status400BadRequest)]
[ProducesResponseType(StatusCodes.Status404NotFound)]
0 references
public async Task<ActionResult<PaisDepDto>> getPaisByNameDeps(string nombre)
{
    var pais = await _unitOfWork.Paises.GetPaisByNameAndDeps(nombre);
    if (pais == null){
        return NotFound();
    }
    return _mapper.Map<PaisDepDto>(pais);
}
```

Buscar un país por el nombre y mostrar los Departamentos asociados.

LINQ

Microsoft® .NET

LINQ

Microsoft® .NET