PN
PASSERELLES NUMERIQUES

Microsoft®
.NET

UNIT 8:
LINQ

CONTENTS

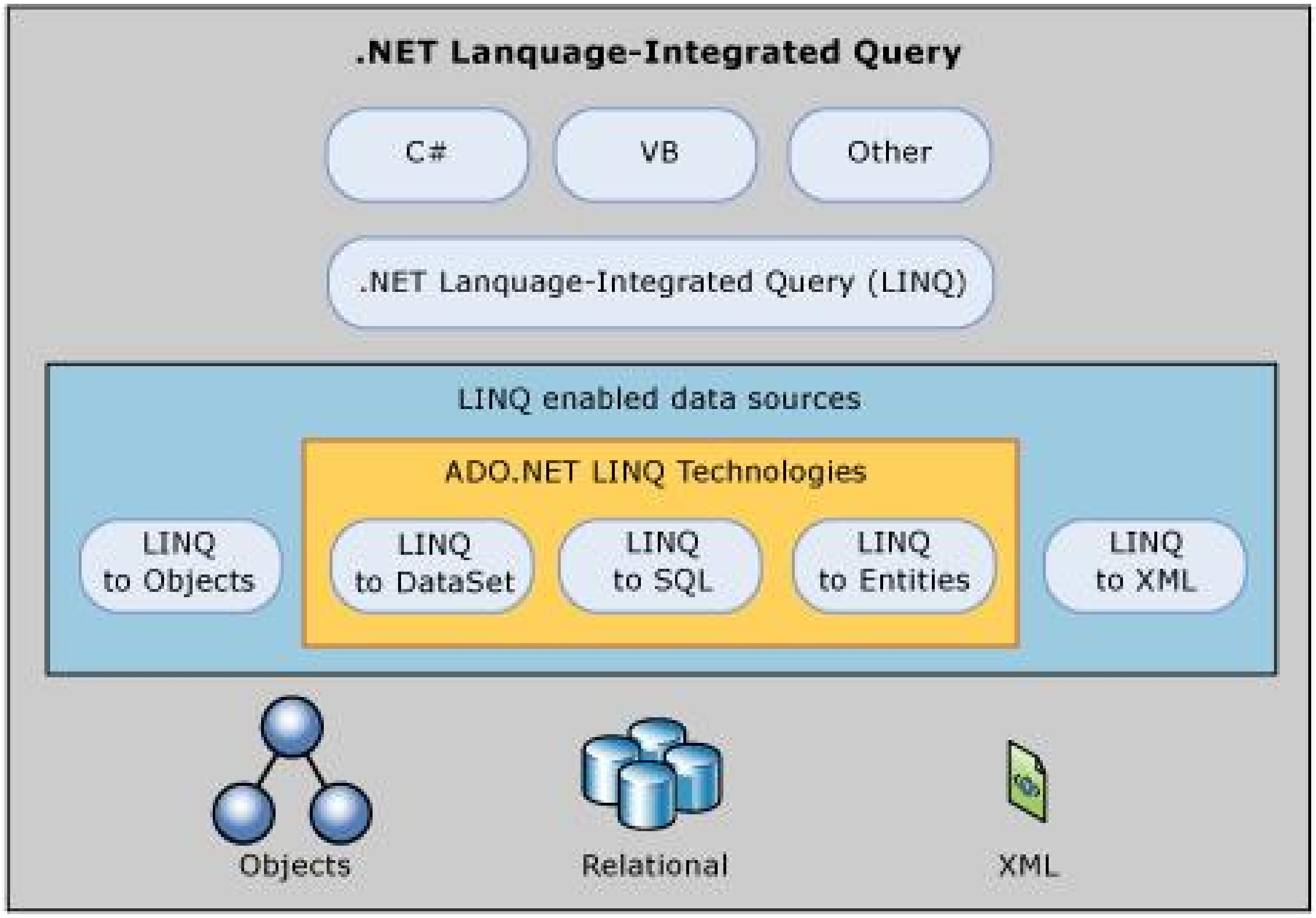
- What is LinQ?
- Advantages of LinQ
- LinQ Query Syntax
- LinQ Method Syntax
- Query operators

What is LinQ

- **LINQ (Language Integrated Query) is uniform query syntax in C# and VB.NET used to save and retrieve data from different sources.**
- **LINQ always works with objects so you can use the same basic coding patterns to query and transform data**

What is LinQ

- **LinQ Architecture in .NET**

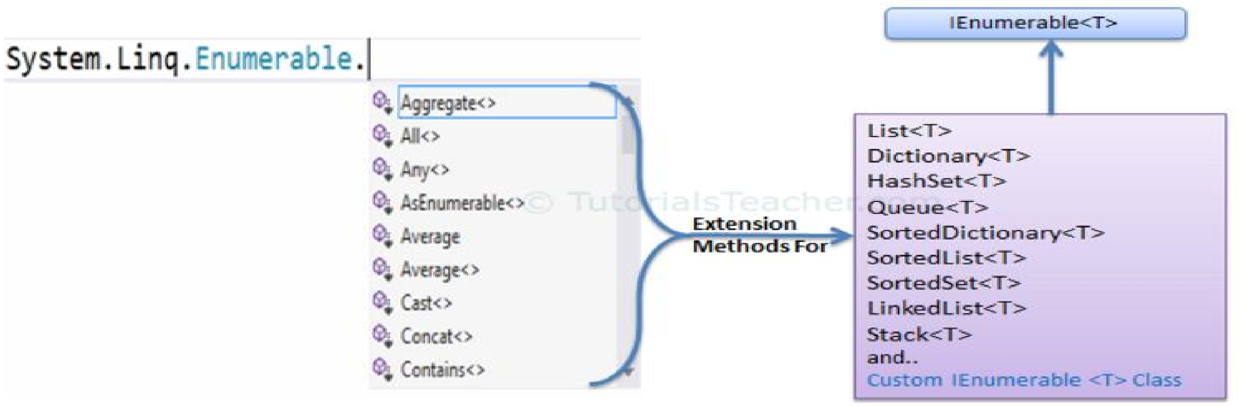

.NET Lanquage-Integrated Query
C#
VB
Other
.NET Lanquage-Integrated Query (LINQ)
LINQ enabled data sources
ADO.NET LINQ Technologies
LINQ to Objects
LINQ to DataSet
LINQ to SQL
LINQ to Entities
LINQ to XML
Objects
Relational
XML

Advantages of LinQ

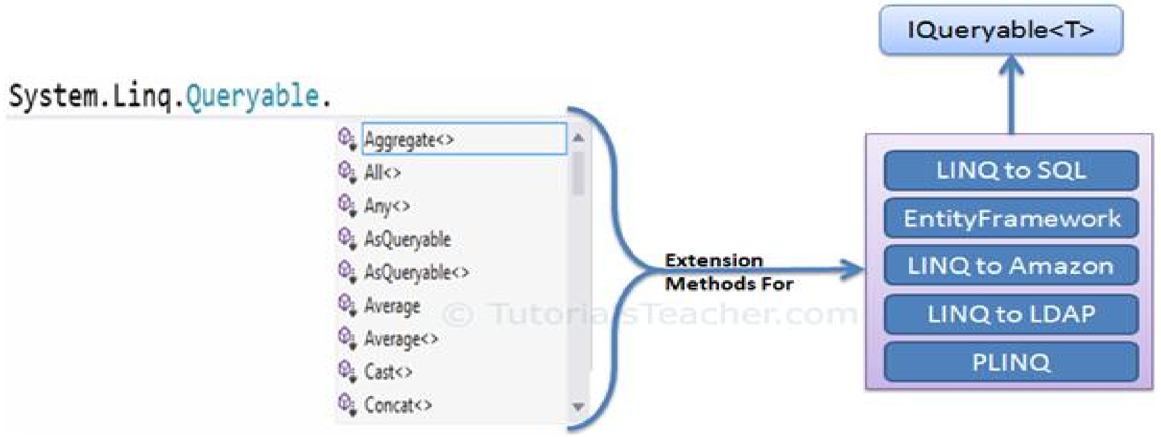
- Familiar language: Developers don’t have to learn a new query language for each type of data source or data format.
- Less coding: It reduces the amount of code to be written as compared with a more traditional approach.
- Readable code: LINQ makes the code more readable so other developers can easily 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 syntax

- **LinQ API includes two main static class:**



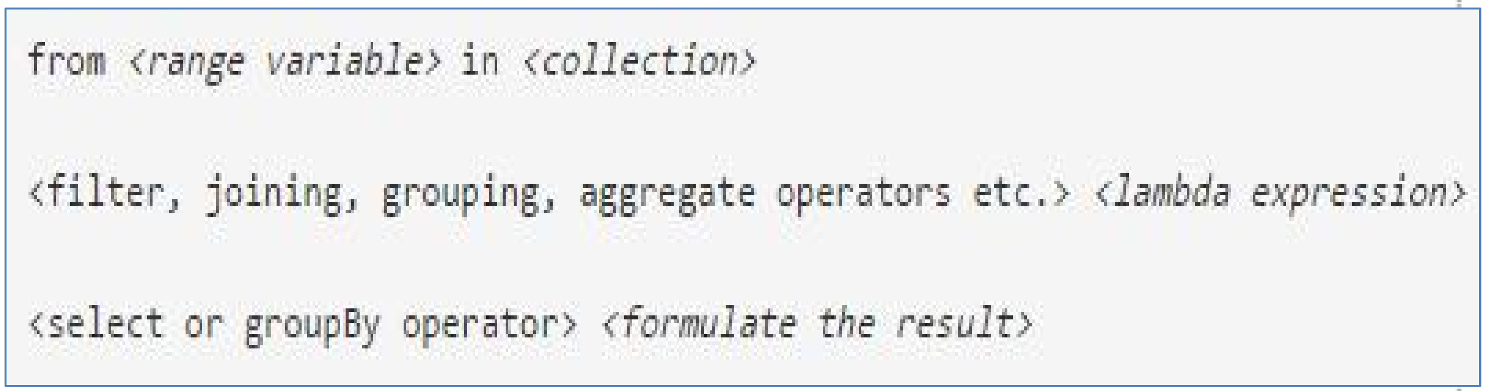
IEnumerable<T> extension methods in Enumerable class



IQueryable extension methods in Queryable class

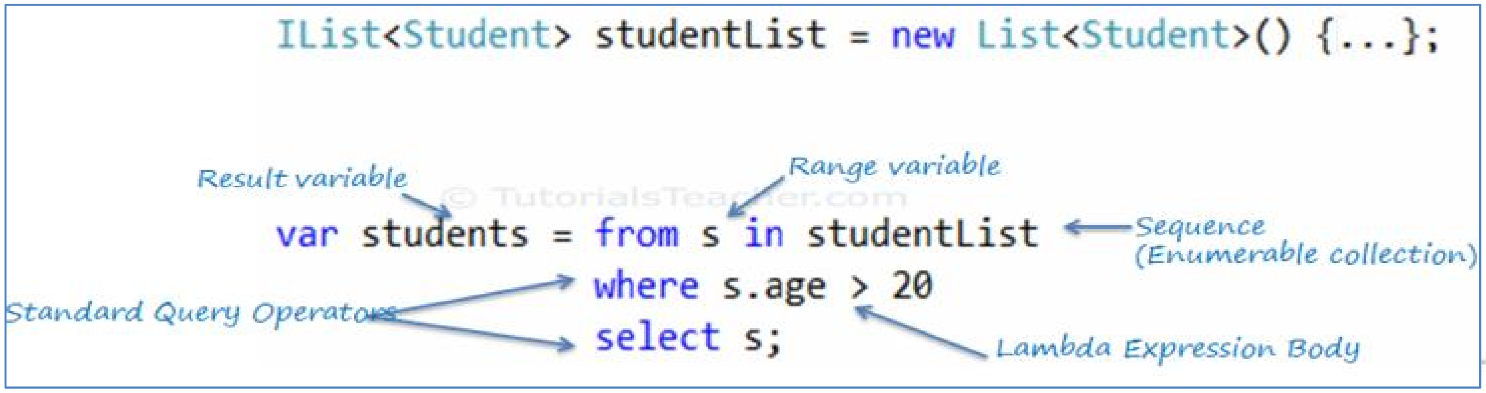
Query Syntax

- **Query syntax is similar to SQL (Structured Query Language) for a database. It is defined within the C# or VB code.**
- **Syntax:**



```
from <range variable> in <collection>

<filter, joining, grouping, aggregate operators etc.> <lambda expression>

<select or groupBy operator> <formulate the result>
```

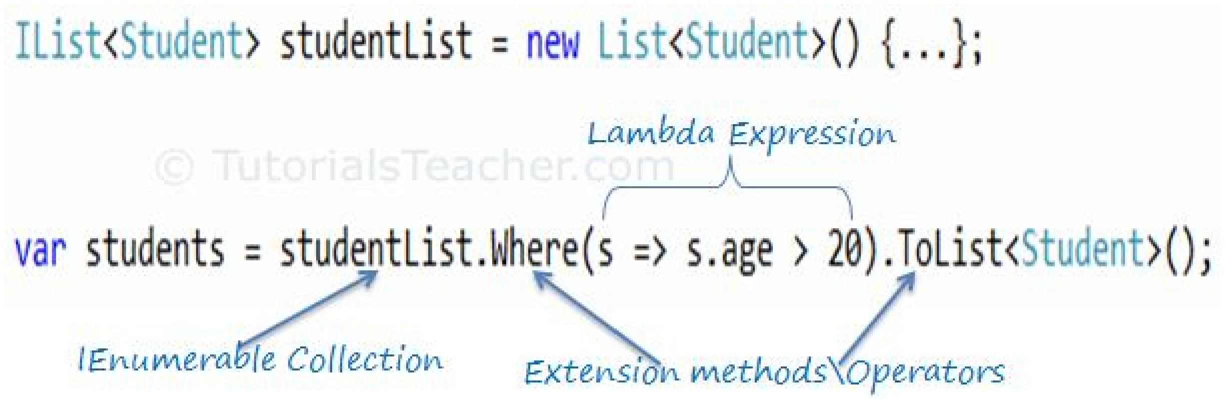
- **Ex:**



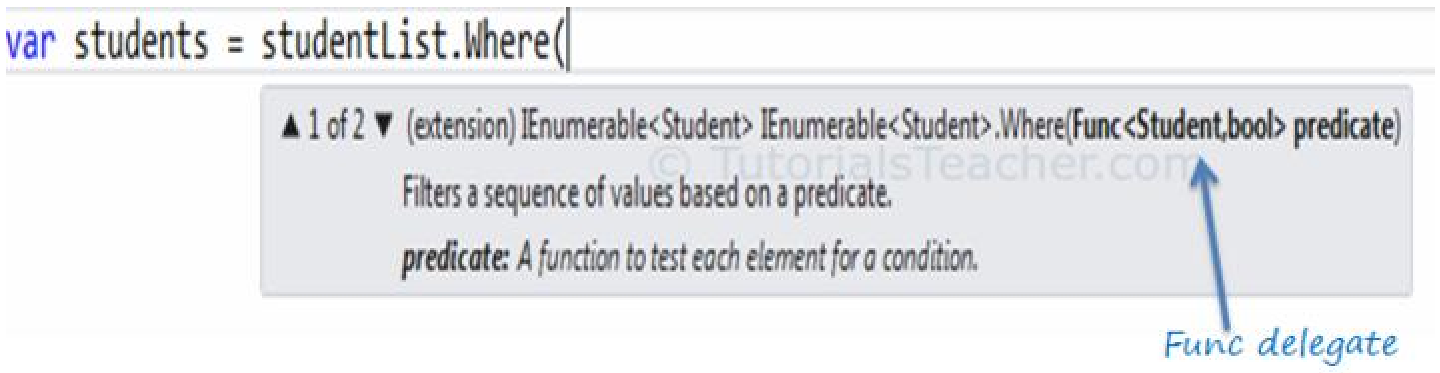
```
IList<Student> studentList = new List<Student>() {...};

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

Method Syntax

- **Method syntax uses extension methods included in the Enumerable or Queryable static class**
- **Ex:**



```
IList<Student> studentList = new List<Student>() {...};

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

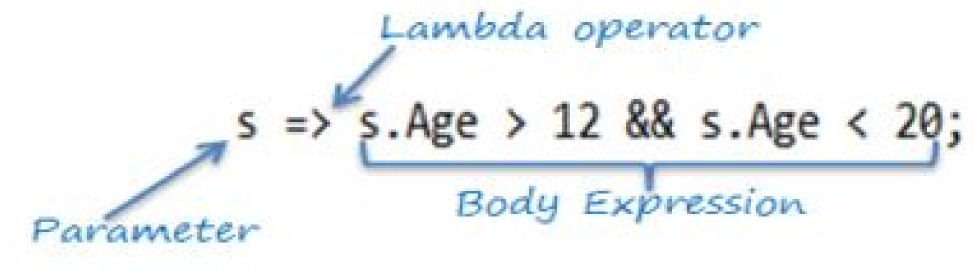


```
var students = studentList.Where(
```

▲ 1 of 2 ▼ (extension) IEnumerable<Student> IEnumerable<Student>.Where(**Func<Student,bool> predicate**)
Filters a sequence of values based on a predicate.
predicate: *A function to test each element for a condition.*

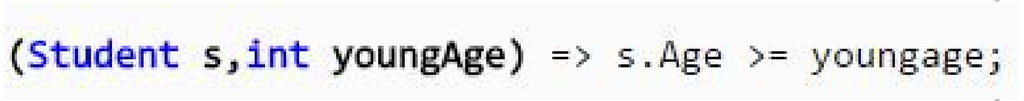
Lambda expression

- **Syntax:** ***parameters => body expression***
- **Ex:**



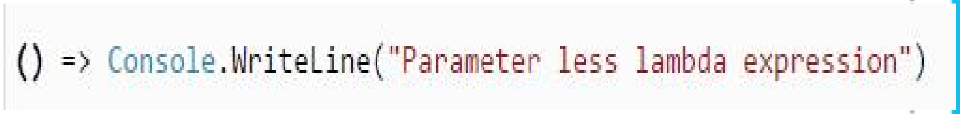
```
s => s.Age > 12 && s.Age < 20;
```

- **Multiple parameters:**



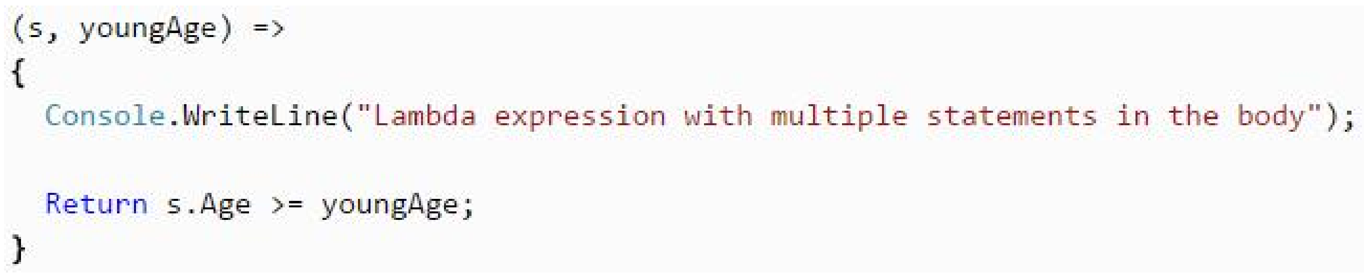
```
(Student s,int youngAge) => s.Age >= youngage;
```

- **Without any parameter:**



```
() => Console.WriteLine("Parameter less lambda expression")
```

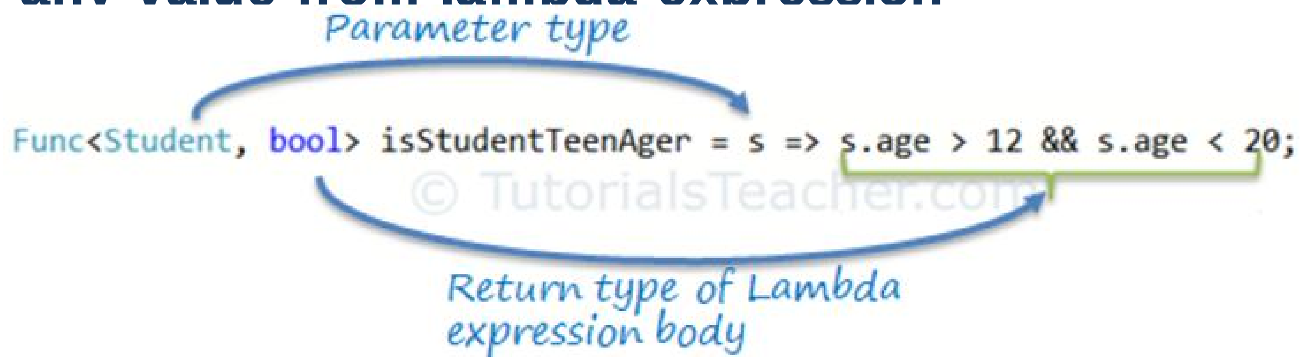
- **Multiple statements in body expression:**



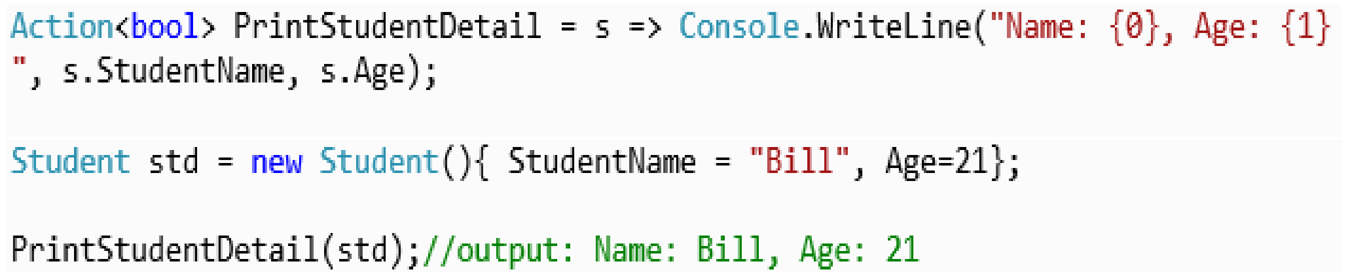
```
(s, youngAge) =>
{
  Console.WriteLine("Lambda expression with multiple statements in the body");

  Return s.Age >= youngAge;
}
```

Lambda expression

- **Invoke the lambda expression:**
  - **Use Func delegate type: when you want to return something from a lambda expression.**
  - **Use Action delegate type when you don't need to return any value from lambda expression**



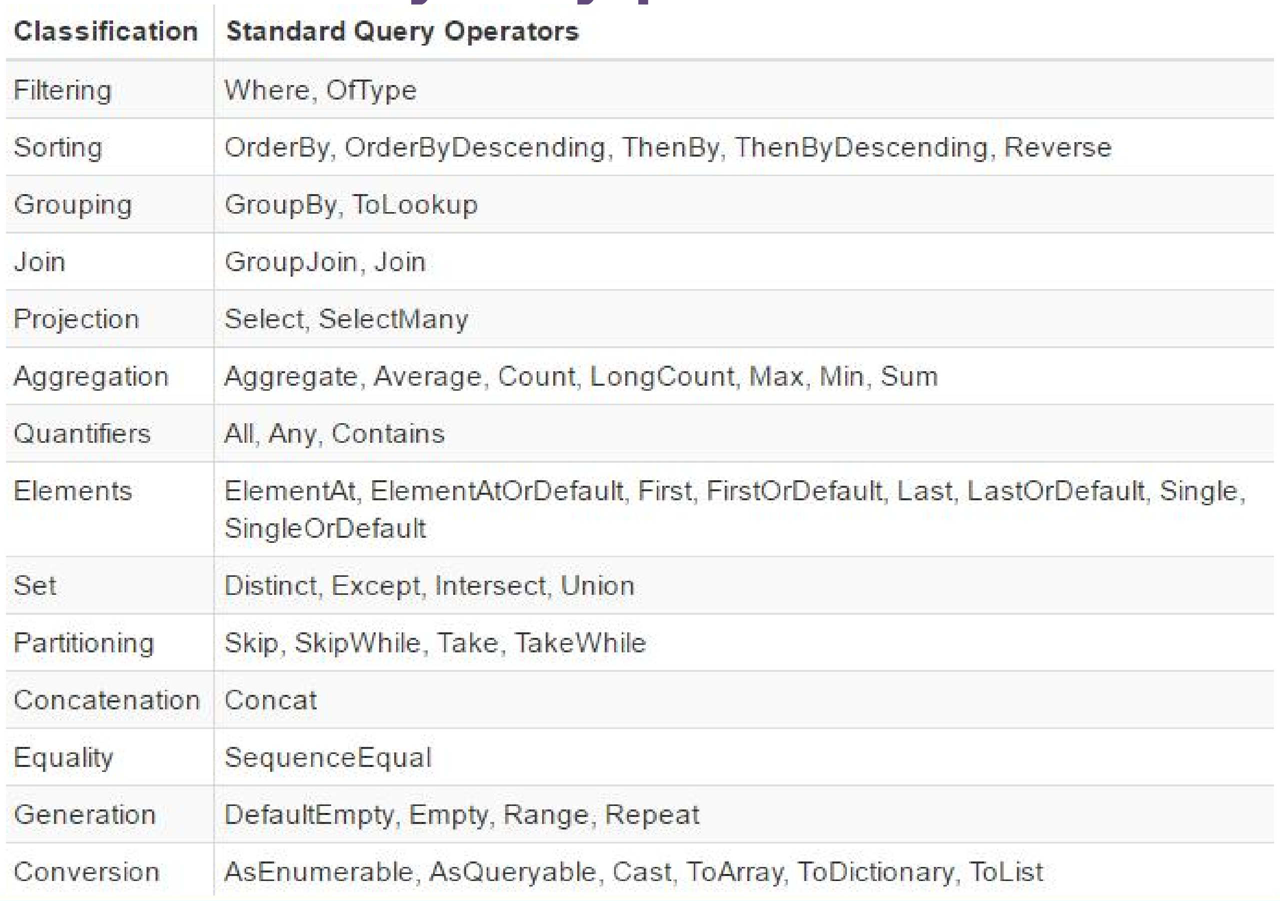
```
Func<Student, bool> isStudentTeenAger = s => s.age > 12 && s.age < 20;
```



```
Action<bool> PrintStudentDetail = s => Console.WriteLine("Name: {0}, Age: {1}
", s.StudentName, s.Age);

Student std = new Student(){ StudentName = "Bill", Age=21};

PrintStudentDetail(std);//output: Name: Bill, Age: 21
```

Query Operators

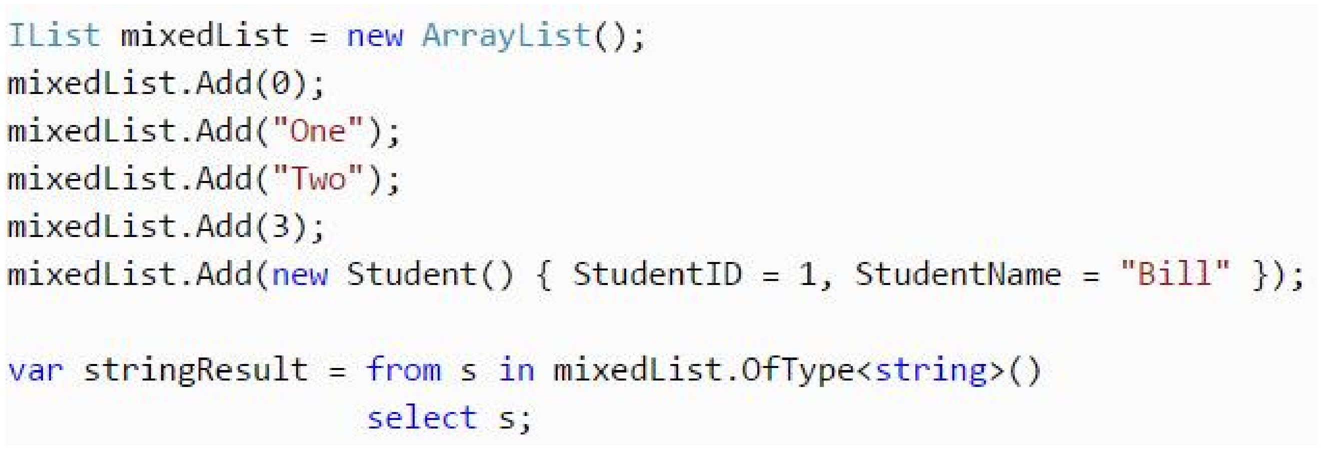
- **Query Operators can be classified based on the functionality they provide.**



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

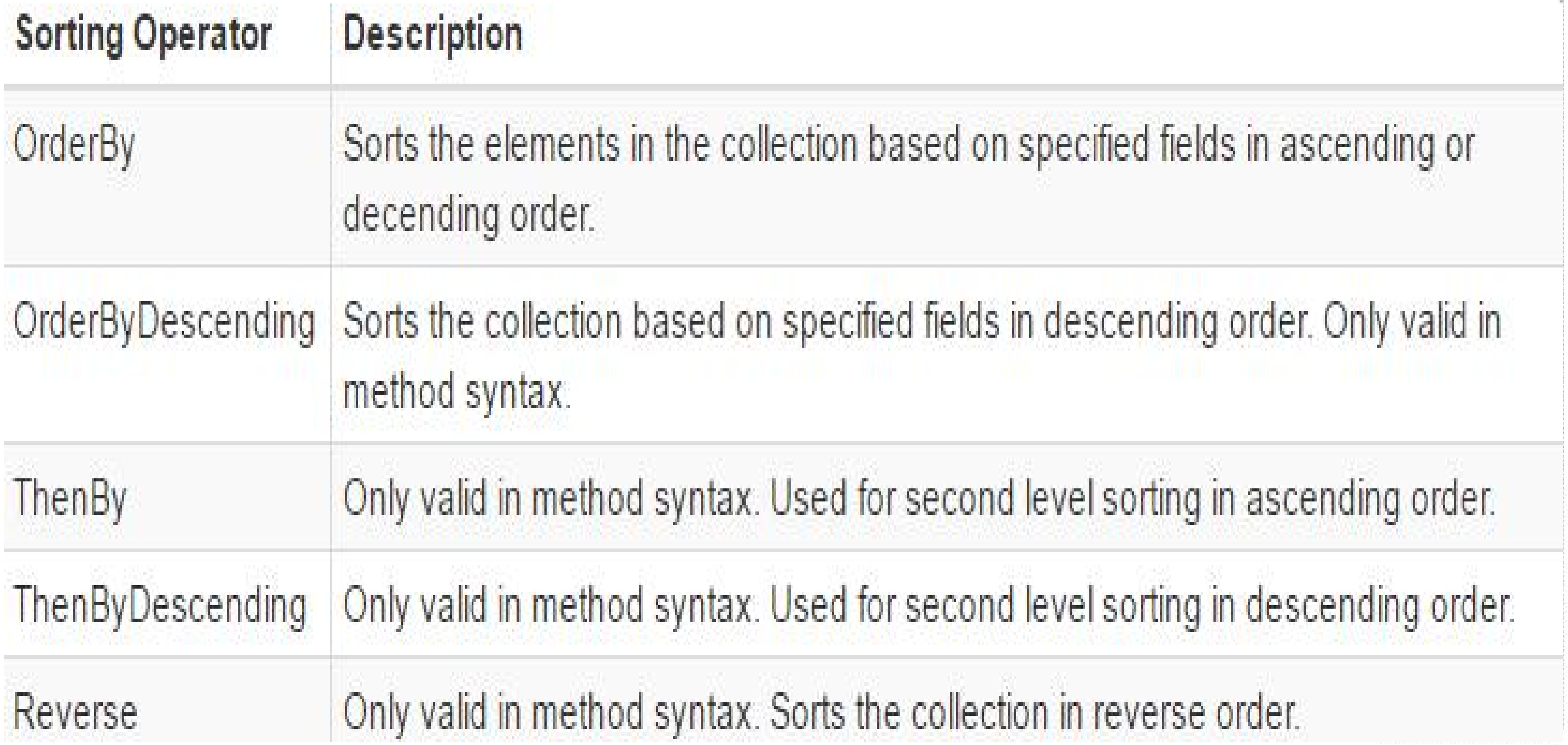
- **Where:**
  - filters the collection based on a given criteria. It accepts a predicate as a parameter.
  - Can call the Where() extension method more than one time in a single LINQ query.
- **OfType:**
  - filter collection based on each element's type



```
IList mixedList = new ArrayList();
mixedList.Add(0);
mixedList.Add("One");
mixedList.Add("Two");
mixedList.Add(3);
mixedList.Add(new Student() { StudentID = 1, StudentName = "Bill" });

var stringResult = from s in mixedList.OfType<string>()
                   select s;
```

Sorting operator

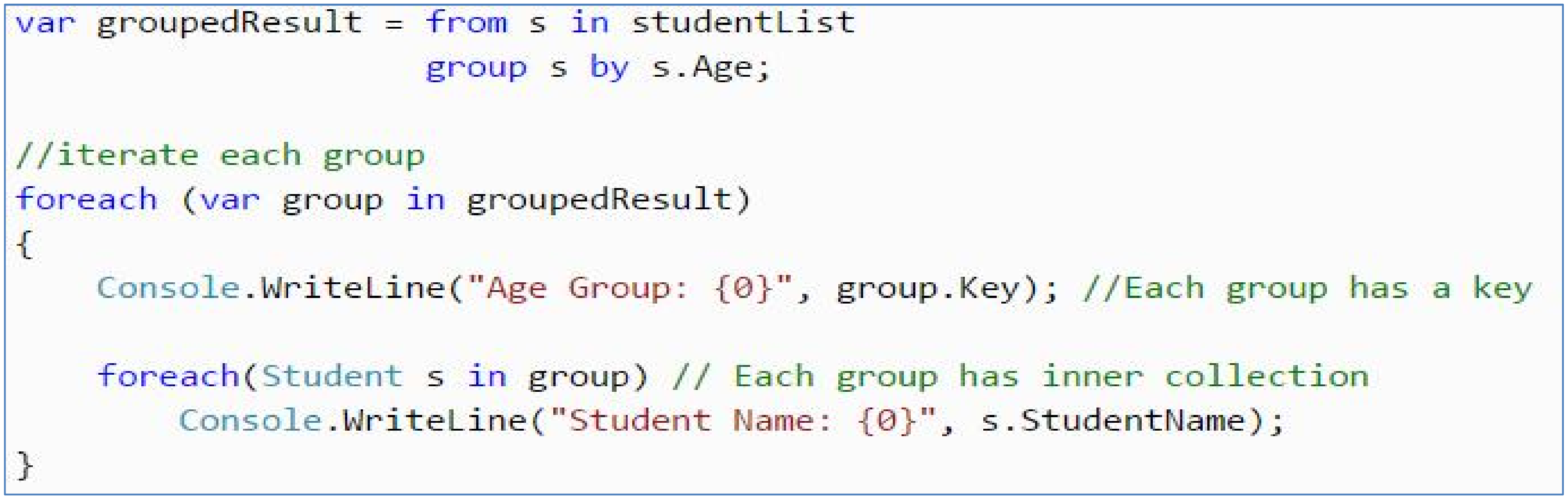
- **Use to arrange elements of the collection in ascending or descending order**



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.

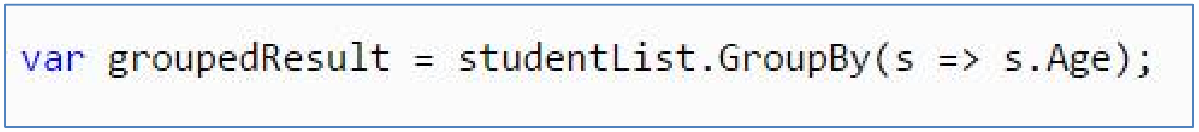
Group operator

- **The GroupBy operator returns a group of elements from the given collection based on some key value**
- **Query:**



```
var groupedResult = from s in studentList
                    group s by s.Age;

//iterate each group
foreach (var group in groupedResult)
{
    Console.WriteLine("Age Group: {0}", group.Key); //Each group has a key

    foreach(Student s in group) // Each group has inner collection
        Console.WriteLine("Student Name: {0}", s.StudentName);
}
```

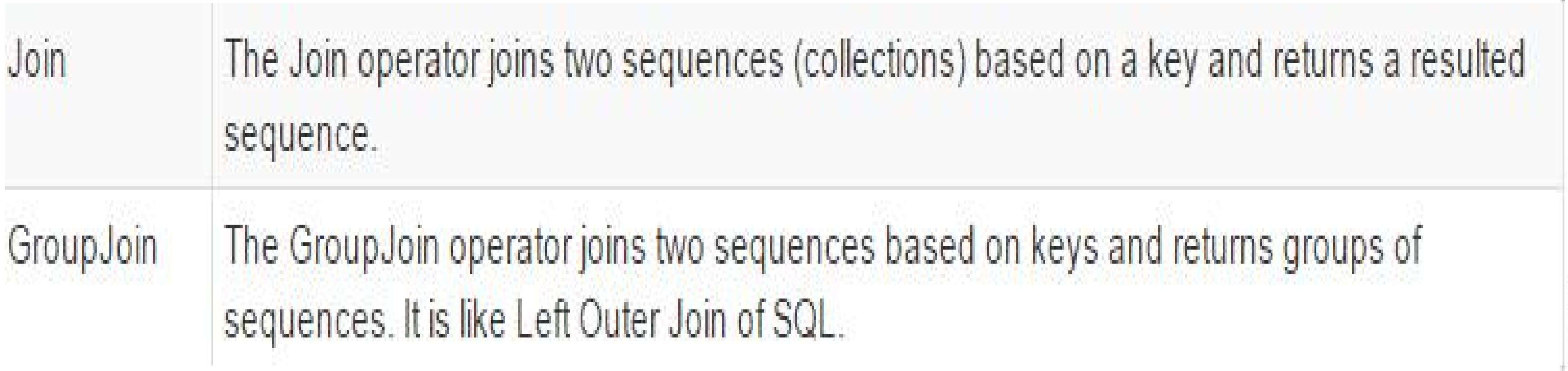
- **Method:**



```
var groupedResult = studentList.GroupBy(s => s.Age);
```

Joining Operator

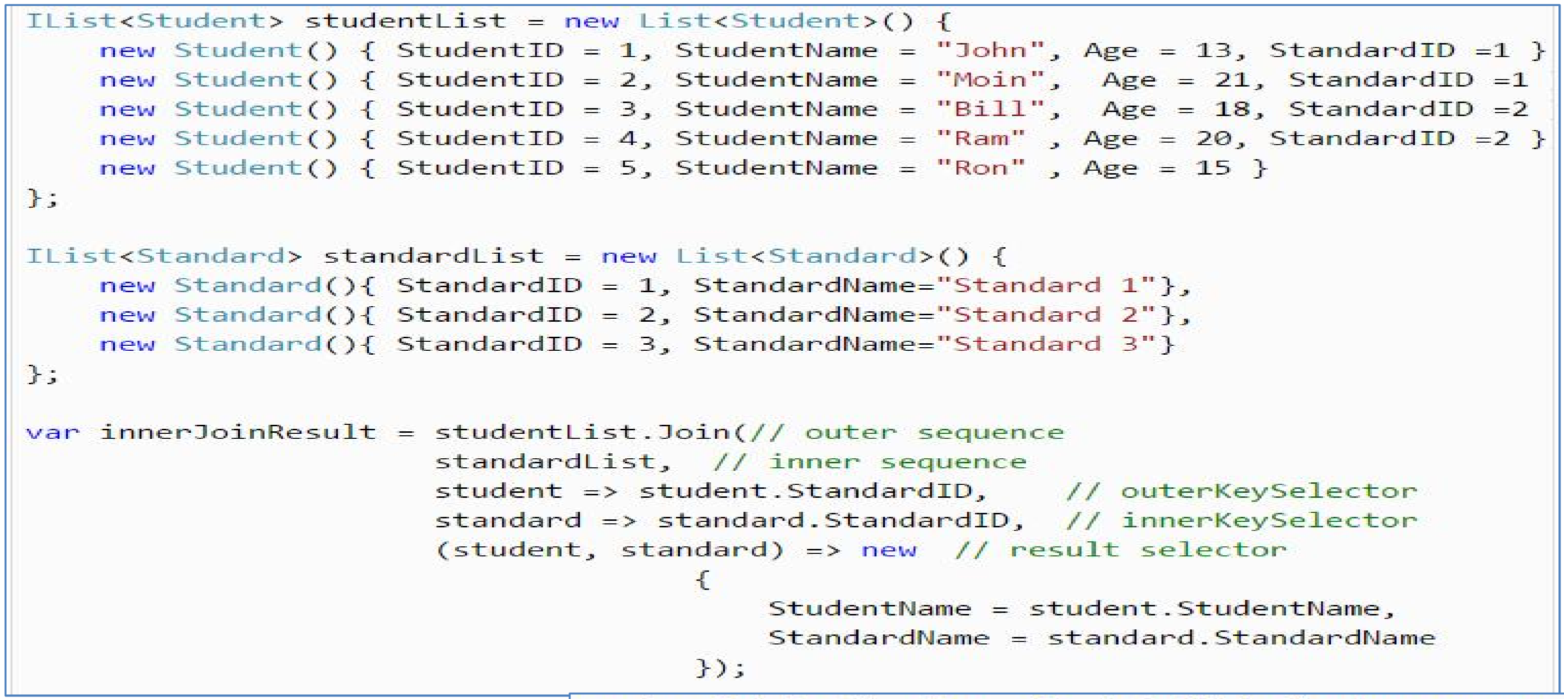
- The Join operator operates on two collections, inner collection & outer collection. It returns a new collection that contains elements from both the collections which satisfies specified expression.



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.

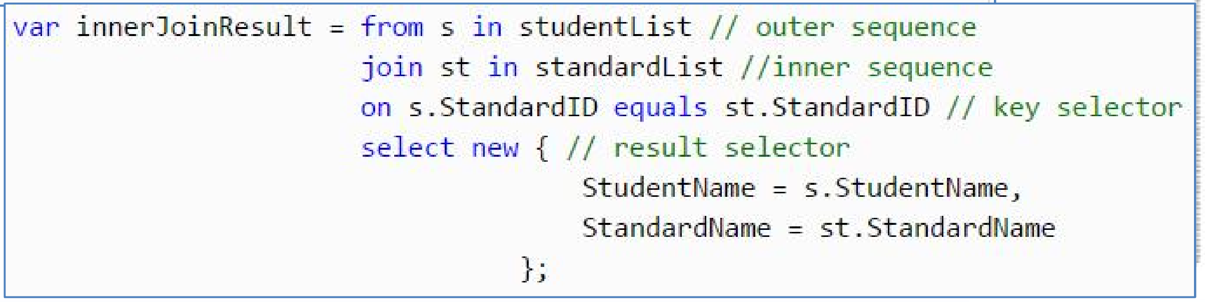
Join

- **Join in Method:**



```
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 innerJoinResult = 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
                                    });
```

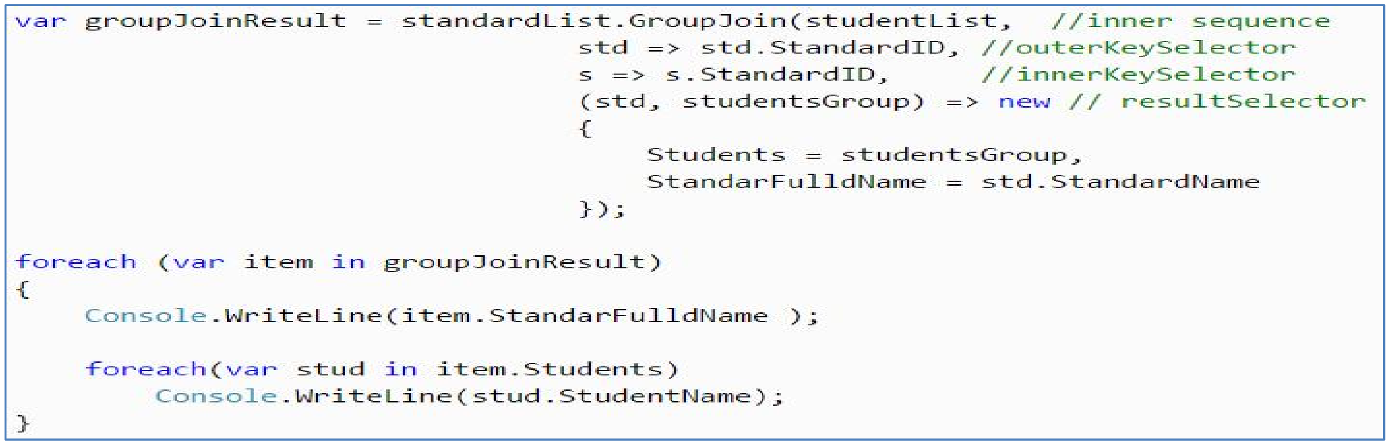
- **Join in Query:**



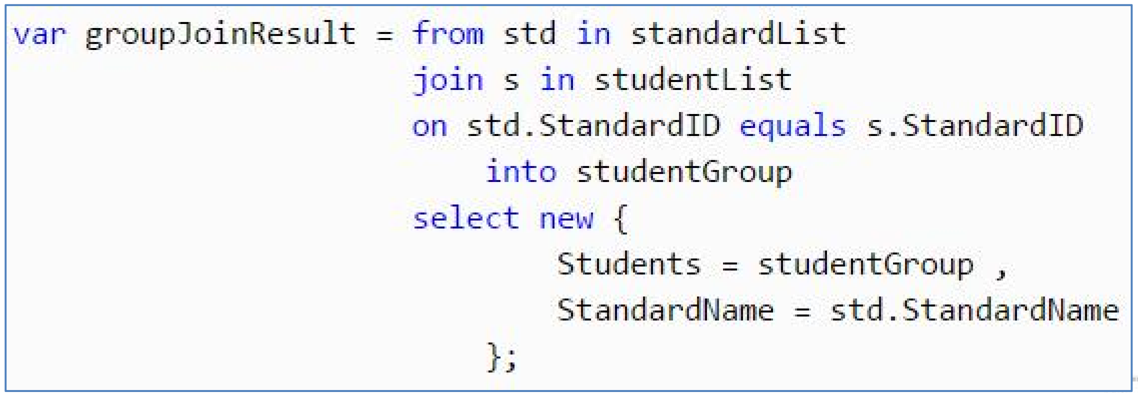
```
var innerJoinResult = from s in studentList // outer sequence
                      join st in standardList //inner sequence
                      on s.StandardID equals st.StandardID // key selector
                      select new { // result selector
                                    StudentName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

GoupJoin

- **The GroupJoin operator performs the same task as Join operator, except that it returns a group result**



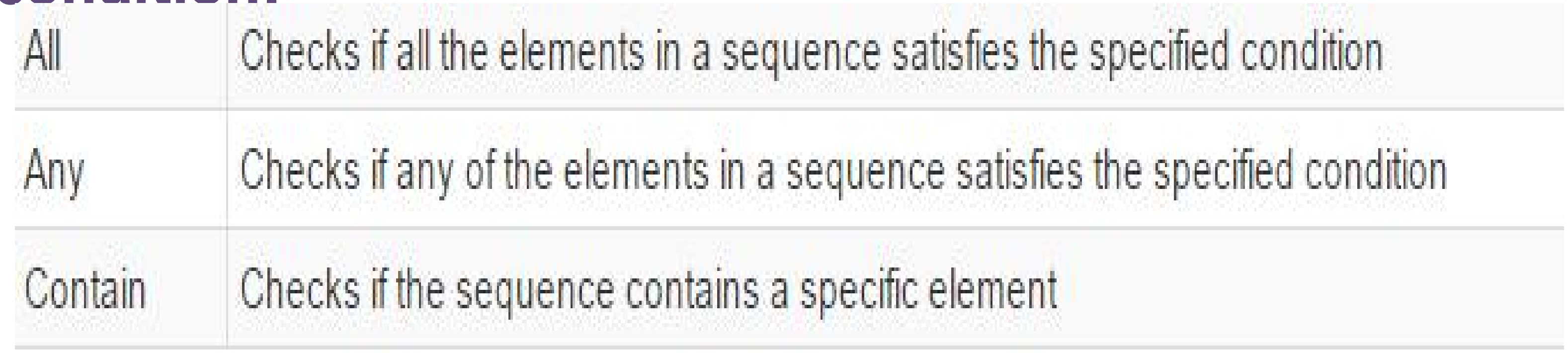
```
var groupJoinResult = standardList.GroupJoin(studentList,  //inner sequence
                                std => std.StandardID, //outerKeySelector
                                s => s.StandardID,     //innerKeySelector
                                (std, studentsGroup) => new // resultSelector
                                {
                                    Students = studentsGroup,
                                    StandarFulldName = std.StandardName
                                });

foreach (var item in groupJoinResult)
{
    Console.WriteLine(item.StandarFulldName );

    foreach(var stud in item.Students)
        Console.WriteLine(stud.StudentName);
}
```



```
var groupJoinResult = from std in standardList
                      join s in studentList
                      on std.StandardID equals s.StandardID
                          into studentGroup
                      select new {
                              Students = studentGroup ,
                              StandardName = std.StandardName
                          };
```

Quantifier Operators

- The quantifier operators evaluate elements of the sequence on some condition and return a boolean value to indicate that some or all elements satisfy the condition.

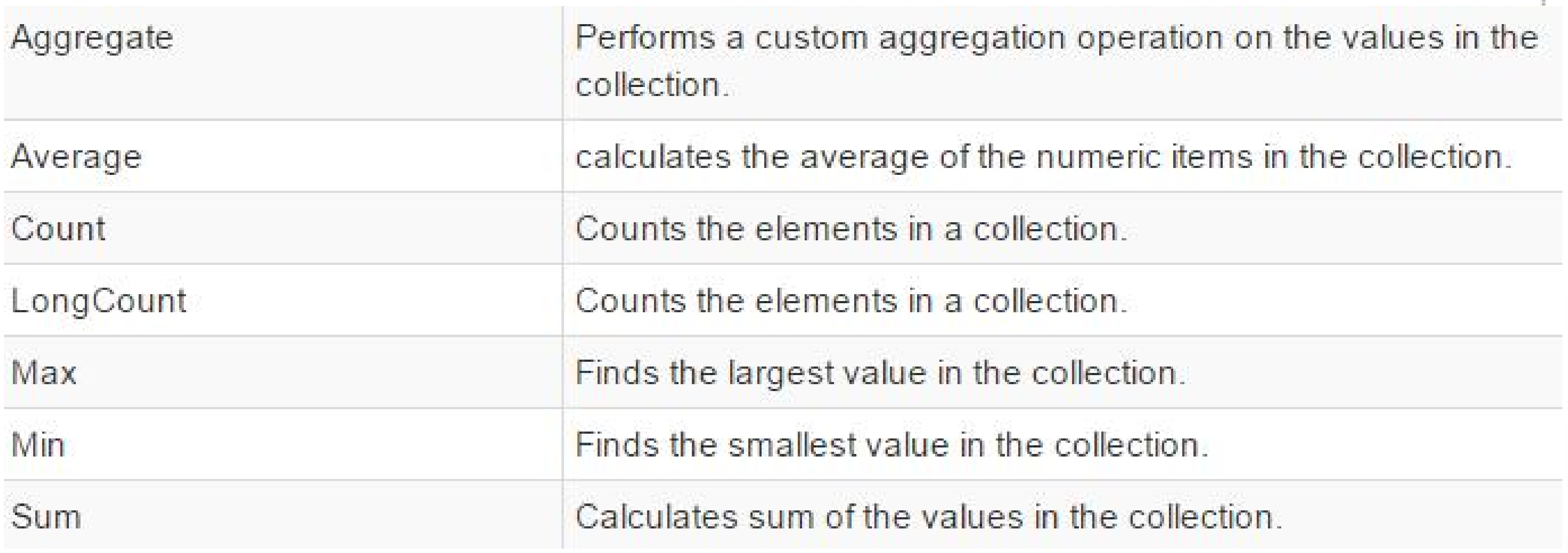


All	Checks if all the elements in a sequence satisfies the specified condition
Any	Checks if any of the elements in a sequence satisfies the specified condition
Contain	Checks if the sequence contains a specific element

- Quantifier operators are Not Supported with C# query syntax.
- Use custom class that derives IEqualityOperator with Contains to check for the object in the collection

Aggregation Operator

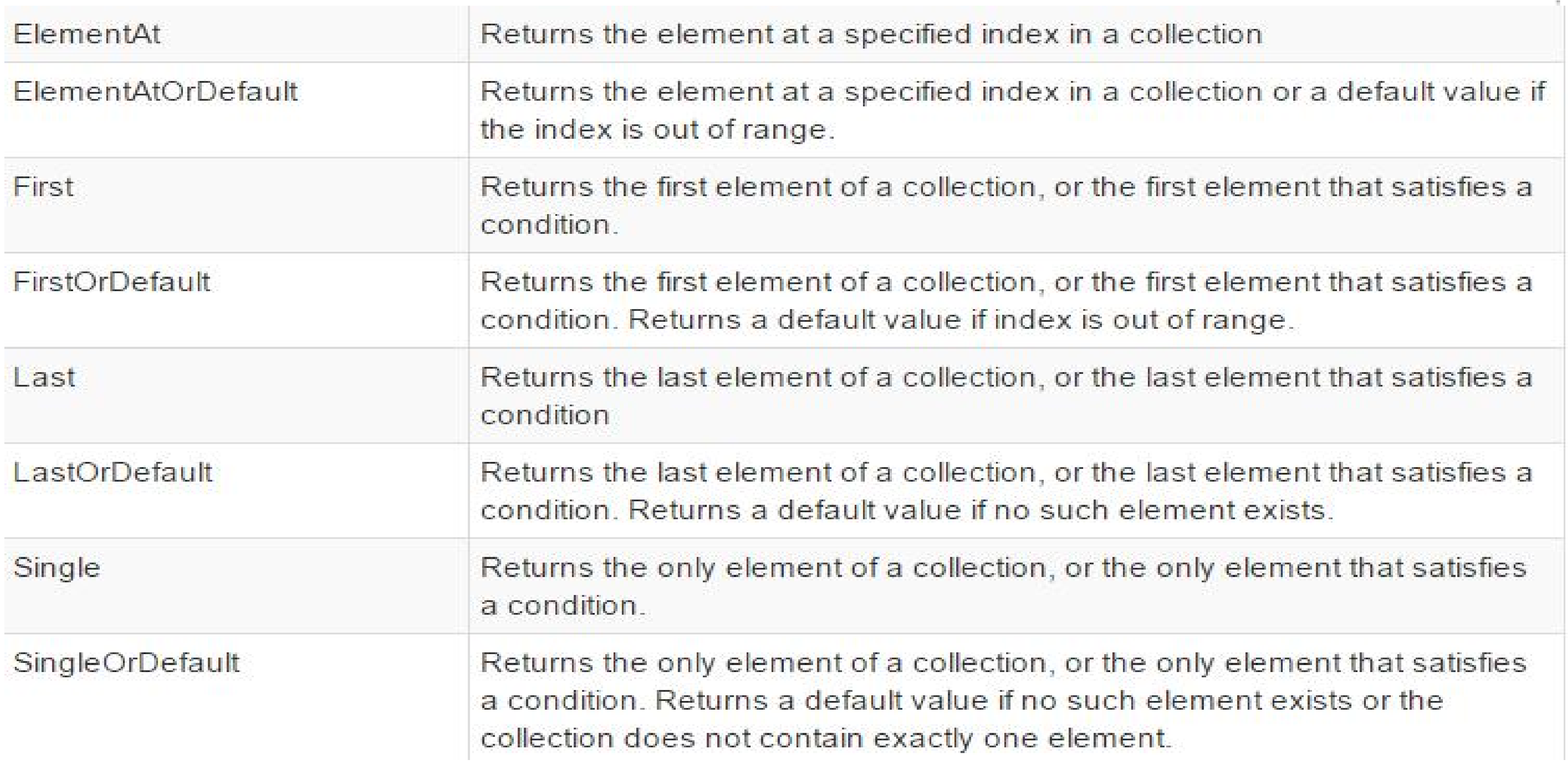
- **The aggregation operators perform mathematical operations like Average, Aggregate, Count, Max, Min and Sum, on the numeric property of the elements in the collection**



Aggregate	Performs a custom aggregation operation on the values in the collection.
Average	calculates the average of the numeric items in the collection.
Count	Counts the elements in a collection.
LongCount	Counts the elements in a collection.
Max	Finds the largest value in the collection.
Min	Finds the smallest value in the collection.
Sum	Calculates sum of the values in the collection.

Element Operators

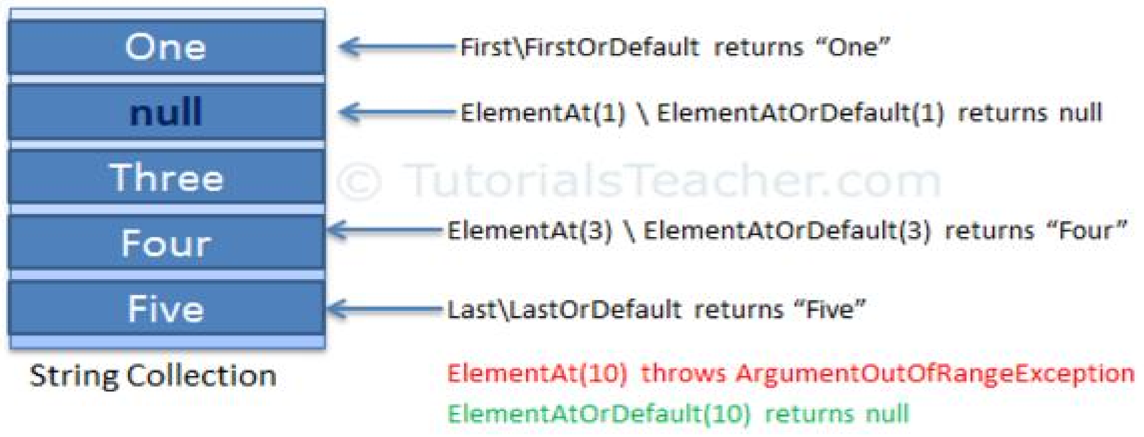
- **Element operators return a particular element from a sequence (collection).**



ElementAt	Returns the element at a specified index in a collection
ElementAtOrDefault	Returns the element at a specified index in a collection or a default value if the index is out of range.
First	Returns the first element of a collection, or the first element that satisfies a condition.
FirstOrDefault	Returns the first element of a collection, or the first element that satisfies a condition. Returns a default value if index is out of range.
Last	Returns the last element of a collection, or the last element that satisfies a condition
LastOrDefault	Returns the last element of a collection, or the last element that satisfies a condition. Returns a default value if no such element exists.
Single	Returns the only element of a collection, or the only element that satisfies a condition.
SingleOrDefault	Returns the only element of a collection, or the only element that satisfies a condition. Returns a default value if no such element exists or the collection does not contain exactly one element.

First / FirstOrDefault Operators

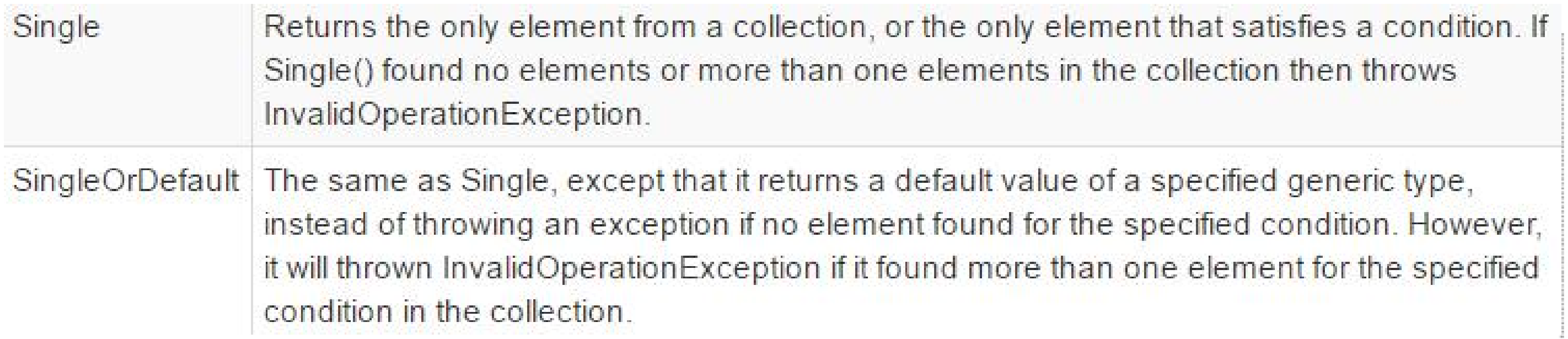
- **Elements Operators**


One
null
Three
Four
Five
String Collection
First\FirstOrDefault returns "One"
ElementAt(1) \ ElementAtOrDefault(1) returns null
ElementAt(3) \ ElementAtOrDefault(3) returns "Four"
Last\LastOrDefault returns "Five"
ElementAt(10) throws ArgumentOutOfRangeException
ElementAtOrDefault(10) returns null
© TutorialsTeacher.com

```
var firstStudent = studentList.First();
Console.WriteLine("First Student Name : {0}", firstStudent.StudentName);

var firstStudent1 = studentList.FirstOrDefault();
Console.WriteLine("First Student Name : {0}", firstStudent1.StudentName);

var firstTeenAgerStudent = studentList.FirstOrDefault(s => s.Age > 12 && s.Age < 20);
Console.WriteLine("First TeenAger Student Name : {0}", firstTeenAgerStudent.StudentName);

var student = studentList.FirstOrDefault(s => s.age > 30);
Console.WriteLine("Student with more than 30 yrs age: {0}", student);
```

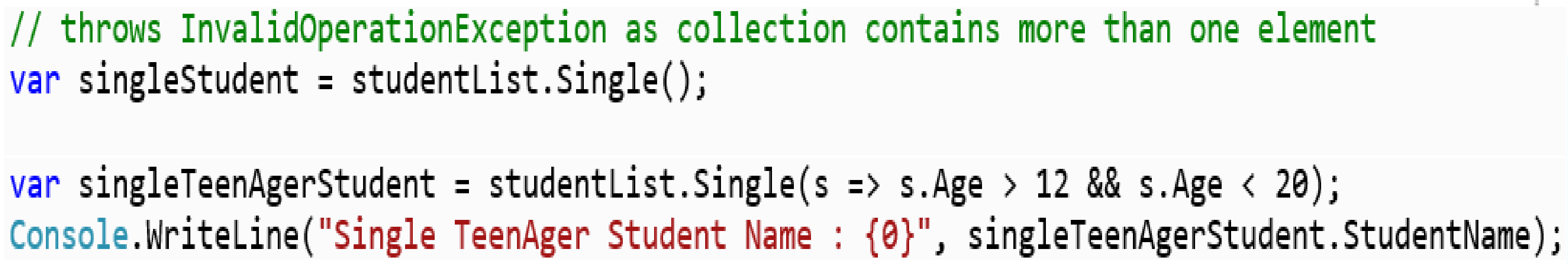
Single / SingleOrDefault Operators

- **Operator**



Single	Returns the only element from a collection, or the only element that satisfies a condition. If Single() found no elements or more than one elements in the collection then throws InvalidOperationException.
SingleOrDefault	The same as Single, except that it returns a default value of a specified generic type, instead of throwing an exception if no element found for the specified condition. However, it will thrown InvalidOperationException if it found more than one element for the specified condition in the collection.

- **Ex**



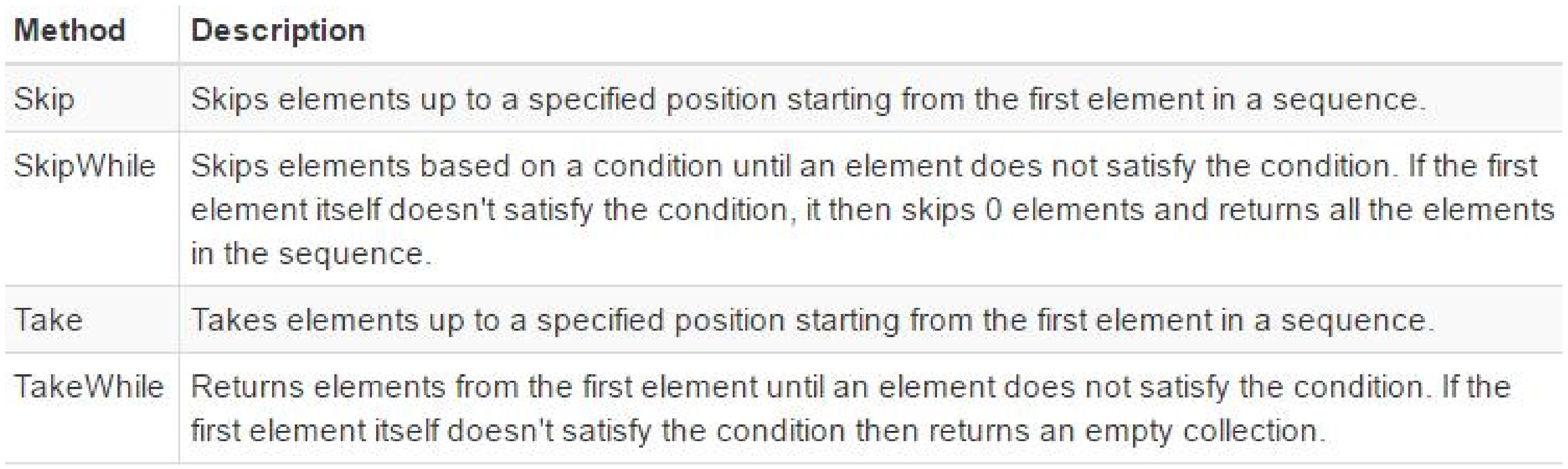
```
// throws InvalidOperationException as collection contains more than one element
var singleStudent = studentList.Single();

var singleTeenAgerStudent = studentList.Single(s => s.Age > 12 && s.Age < 20);
Console.WriteLine("Single TeenAger Student Name : {0}", singleTeenAgerStudent.StudentName);
```

Equality Operator: SequenceEqual

- The SequenceEqual method checks whether the number of elements and value of each element in two collection are equal or not.
- The SequenceEqual method compares the number of items and their values for primitive data types.
- The SequenceEqual method compares the reference of objects for complex data types.

Partioning Operators

- **Partitioning operators split the sequence (collection) into two parts and return one of the parts.**



Method	Description
Skip	Skips elements up to a specified position starting from the first element in a sequence.
SkipWhile	Skips elements based on a condition until an element does not satisfy the condition. If the first element itself doesn't satisfy the condition, it then skips 0 elements and returns all the elements in the sequence.
Take	Takes elements up to a specified position starting from the first element in a sequence.
TakeWhile	Returns elements from the first element until an element does not satisfy the condition. If the first element itself doesn't satisfy the condition then returns an empty collection.