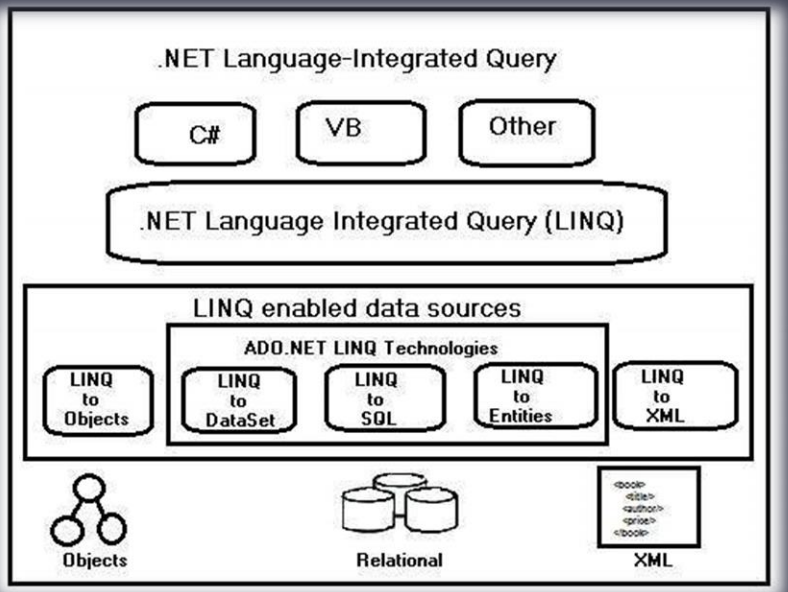
Language Integrated Query

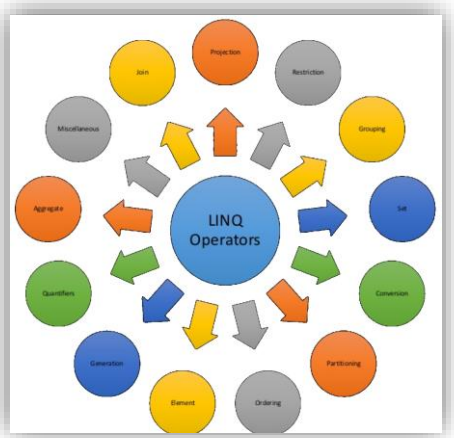
SEW



LINQ

Microsoft® .NET

Overview



- What & why Linq
- Linq API
- Linq Query & Method Syntax
- Lambda Expression
  - Delegate, Func, Action, Predicate, Anonymous Type, Var, Anonymous Method
- Linq to Object
- Standard Query Operators
- Linq to XML

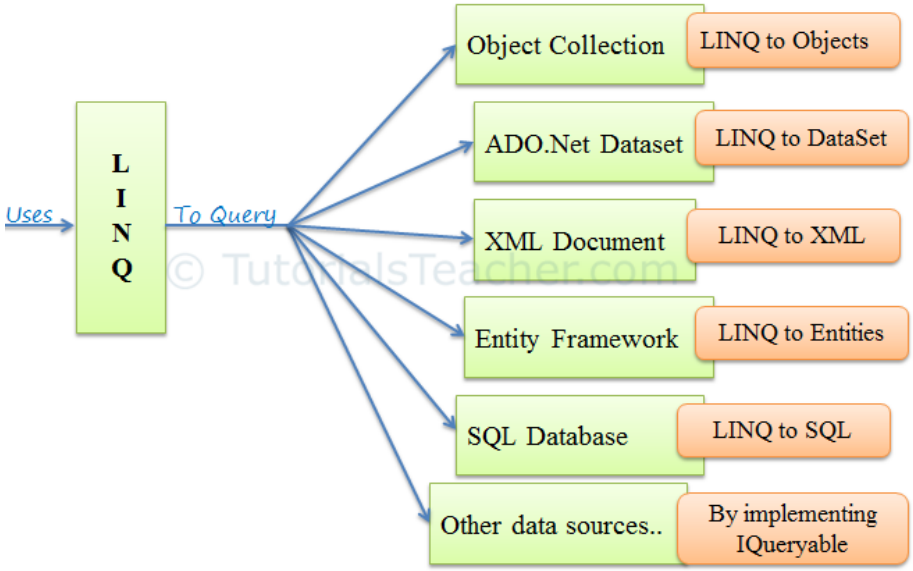
- DAO vs Repositories
- MS SQL DB Server
- Object Relational Mapping

What is LINQ

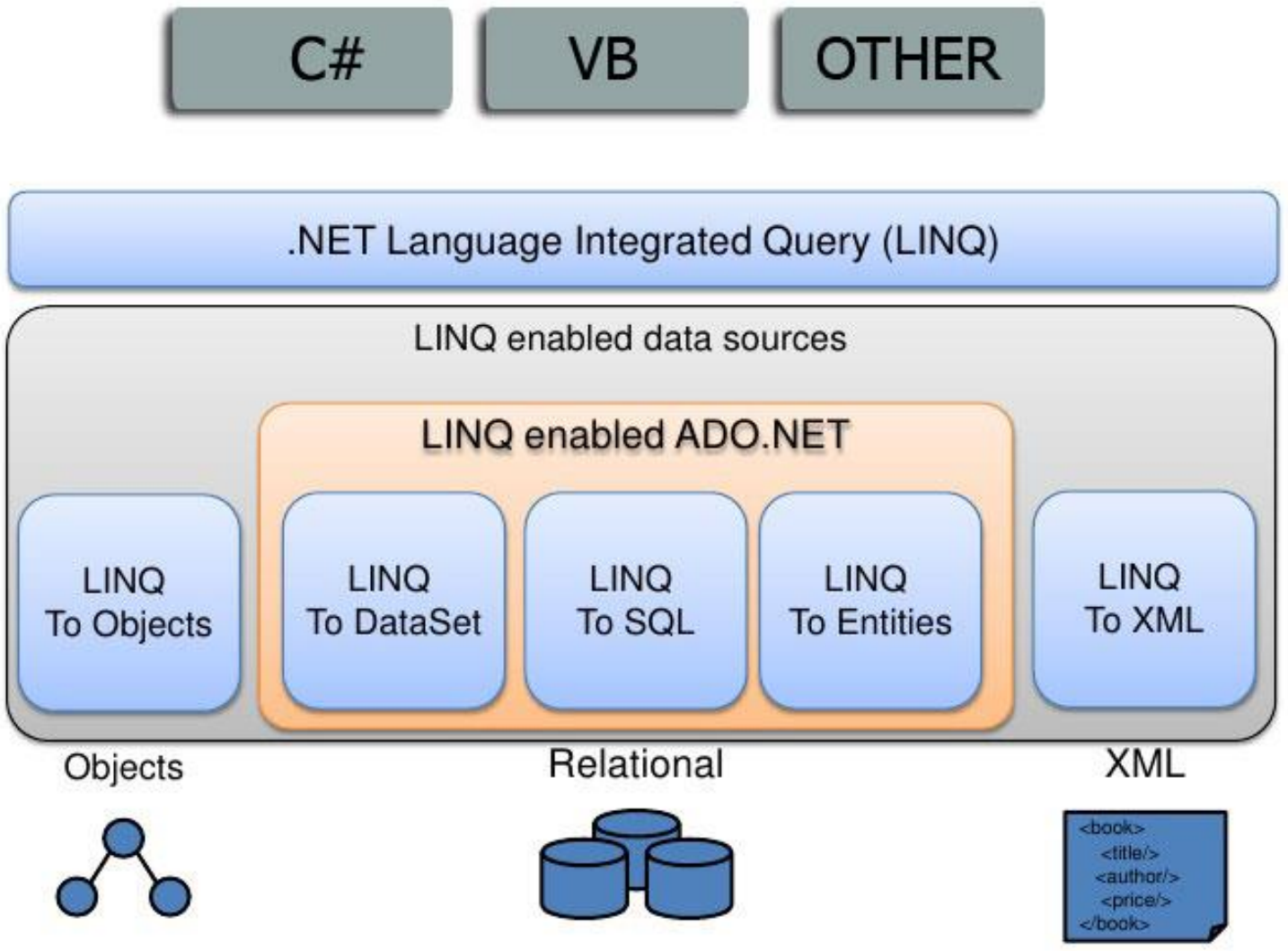
http://www.tutorialsteacher.com/linq/what-is-linq

Language Integrated Query (LINQ)

- provides new and exciting features to query and manipulate data in a consistent and uniform way
- writing or embedding queries on different type of data sources such as
  - Arrays
  - collection of objects
  - XML documents
  - a relational databases
  - Active Directory
  - or even excel sheets

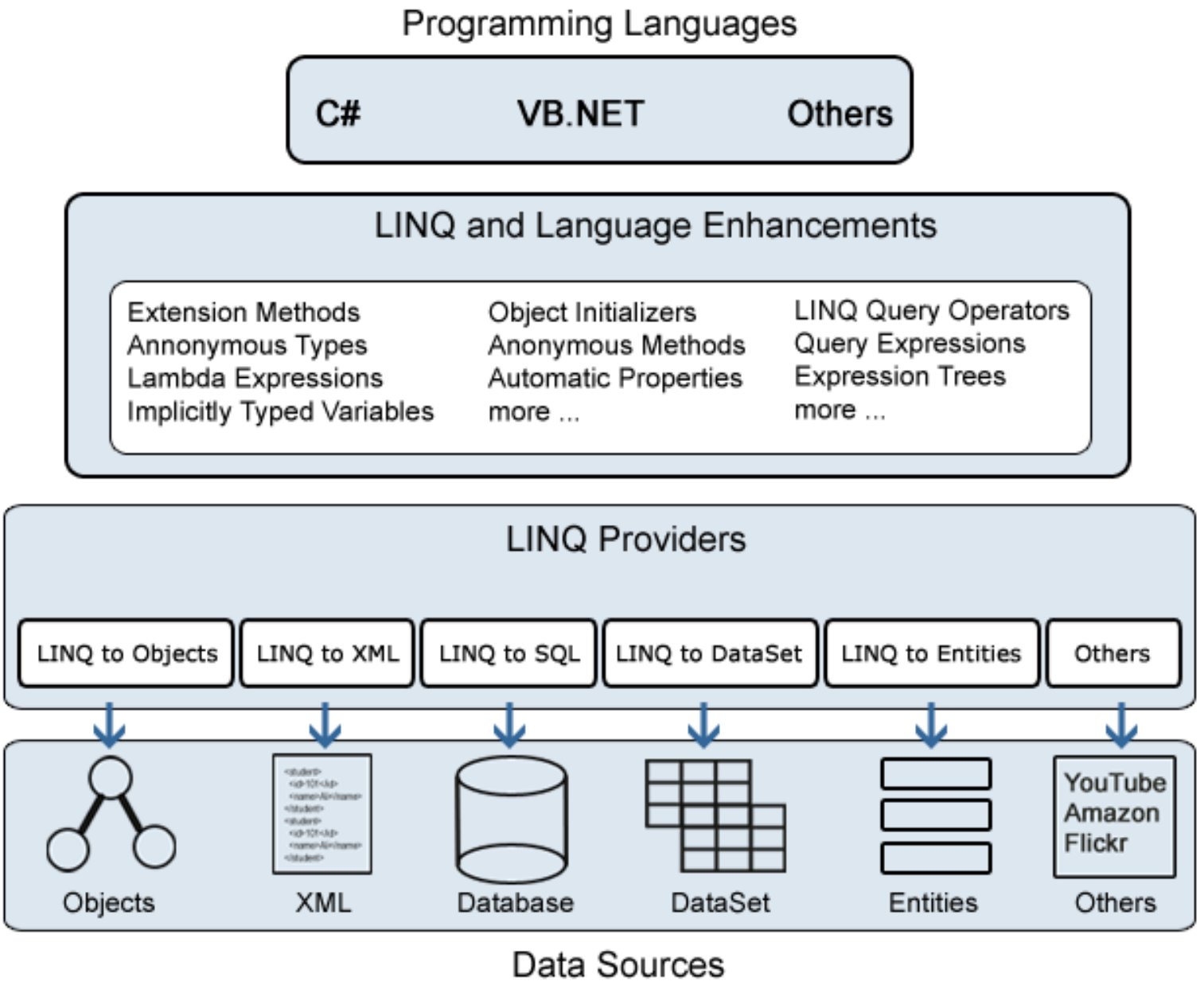


Linq Architecture



LINQ to DataSet

Linq to	Explanation
Linq to Objects	applying Linq queries to arrays and collections
Linq to DataSet	applying Linq queries to Ado.net dataset objects
Linq to XML	using Linq to manipulate and query element tree structured (xml) documents
Linq to Entities	use Linq queries within the Ado.net Entity Framework api
Linq to SQL	is the outdated way to use Linq to access SQL Server databases (Linq to Entity is preferred now)


Programming Languages
C#
VB.NET
Others
LINQ and Language Enhancements
Extension Methods
Annonymous Types
Lambda Expressions
Implicitly Typed Variables
Object Initializers
Anonymous Methods
Automatic Properties
more ...
LINQ Query Operators
Query Expressions
Expression Trees
more ...
LINQ Providers
LINQ to Objects
LINQ to XML
LINQ to SQL
LINQ to DataSet
LINQ to Entities
Others
Objects
XML
Database
DataSet
Entities
YouTube
Amazon
Flickr
Others
Data Sources

Example: Find Teenager Students

- You got an array with 7 Students
  - Find teenager students
  - Find first student whos name is „Bill“
  - Find student whos id is 5

```
Student[] studentArray = {
            new Student() { StudentID = 1, StudentName = "John", age = 18 } ,
            new Student() { StudentID = 2, StudentName = "Steve",  age = 21 } ,
            new Student() { StudentID = 3, StudentName = "Bill",  age = 25 } ,
            new Student() { StudentID = 4, StudentName = "Ram" , age = 20 } ,
            new Student() { StudentID = 5, StudentName = "Ron" , age = 31 } ,
            new Student() { StudentID = 6, StudentName = "Chris",  age = 17 } ,
            new Student() { StudentID = 7, StudentName = "Rob",age = 19  } ,
        };
```

Find elements from the collection

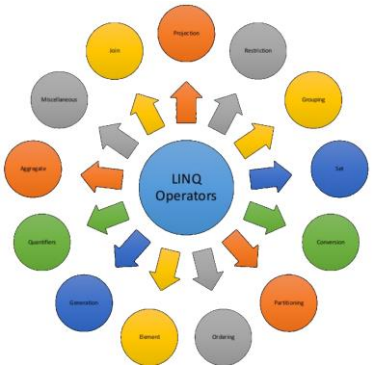
```
class Student
{
    public int StudentID { get; set; }
    public String StudentName { get; set; }
    public int Age { get; set; }
}

class Program
{
    static void Main(string[] args)
    {
        Student[] studentArray = {
            new Student() { StudentID = 1, StudentName = "John", Age = 18 },
            new Student() { StudentID = 2, StudentName = "Steve",  Age = 21 },
            new Student() { StudentID = 3, StudentName = "Bill",  Age = 25 },
            new Student() { StudentID = 4, StudentName = "Ram" , Age = 20 },
            new Student() { StudentID = 5, StudentName = "Ron" , Age = 31 },
            new Student() { StudentID = 6, StudentName = "Chris",  Age = 17 },
            new Student() { StudentID = 7, StudentName = "Rob",Age = 19  },
        };
```

```
Student[] students = new Student[10];

int i = 0;

foreach (Student std in studentArray)
{
    if (std.Age > 12 && std.Age < 20)
    {
        students[i] = std;
        i++;
    }
}
```

Find elements with Linq

```
Student[] studentArray = {
            new Student() { StudentID = 1, StudentName = "John", age = 18 } ,
            new Student() { StudentID = 2, StudentName = "Steve",  age = 21 } ,
            new Student() { StudentID = 3, StudentName = "Bill",  age = 25 } ,
            new Student() { StudentID = 4, StudentName = "Ram" , age = 20 } ,
            new Student() { StudentID = 5, StudentName = "Ron" , age = 31 } ,
            new Student() { StudentID = 6, StudentName = "Chris",  age = 17 } ,
            new Student() { StudentID = 7, StudentName = "Rob",age = 19  } ,
        };

// Use LINQ to find teenager students
Student[] teenAgerStudents = studentArray.Where(s => s.age > 12 && s.age < 20).ToArray();

// Use LINQ to find first student whose name is Bill
Student bill = studentArray.Where(s => s.StudentName == "Bill").FirstOrDefault();

// Use LINQ to find student whose StudentID is 5
Student student5 = studentArray.Where(s => s.StudentID == 5).FirstOrDefault();
```

Advantages of LINQ



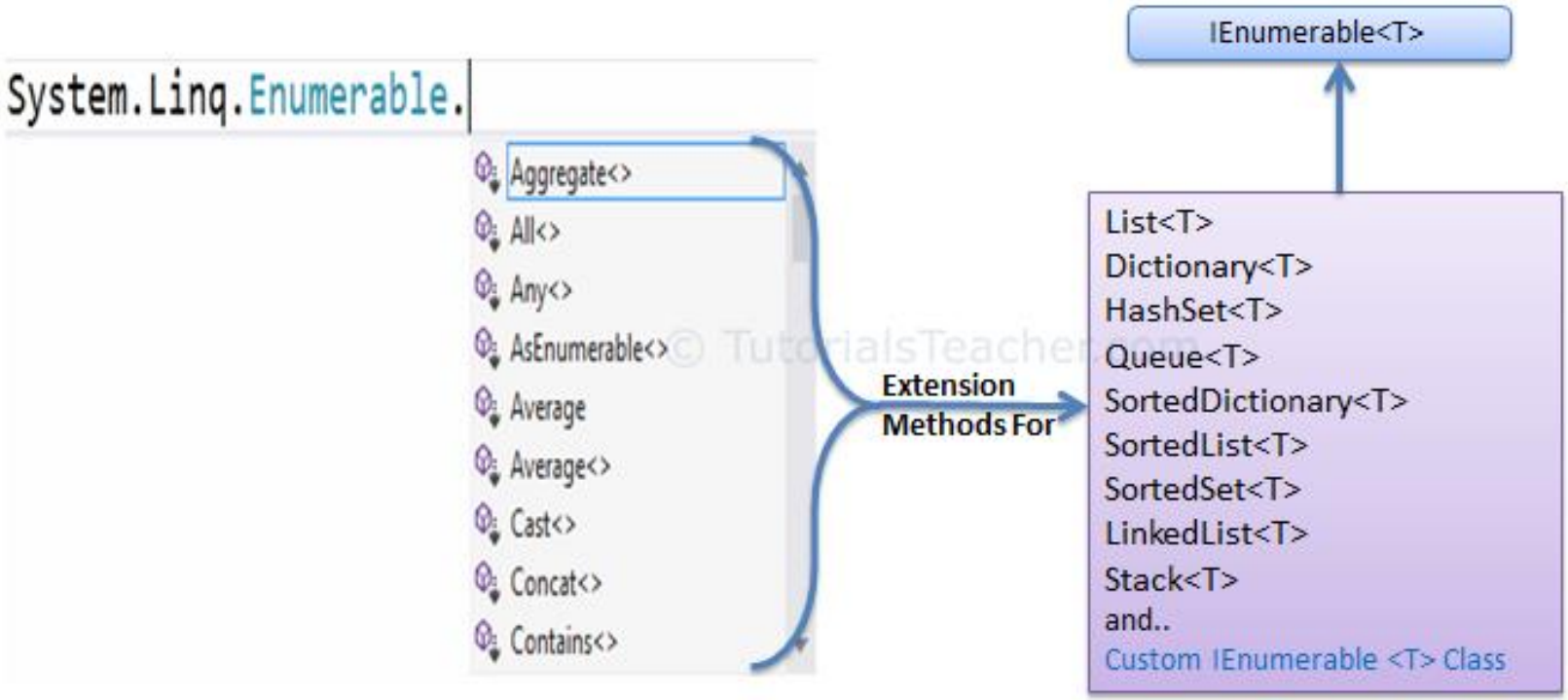
- Familiar language
- Less coding
- Readable code
- Standardized way of querying multiple data sources
- Compile time safety of queries
- IntelliSense Support
- Retrieve data in different shapes

Linq is

- a collection of extension methods for classes that implements IEnumerable and IQueryable interface
- use **System.Linq** namespace to use LINQ
- visit MSDN to know all the extension methods of Enumerable and Queryable class

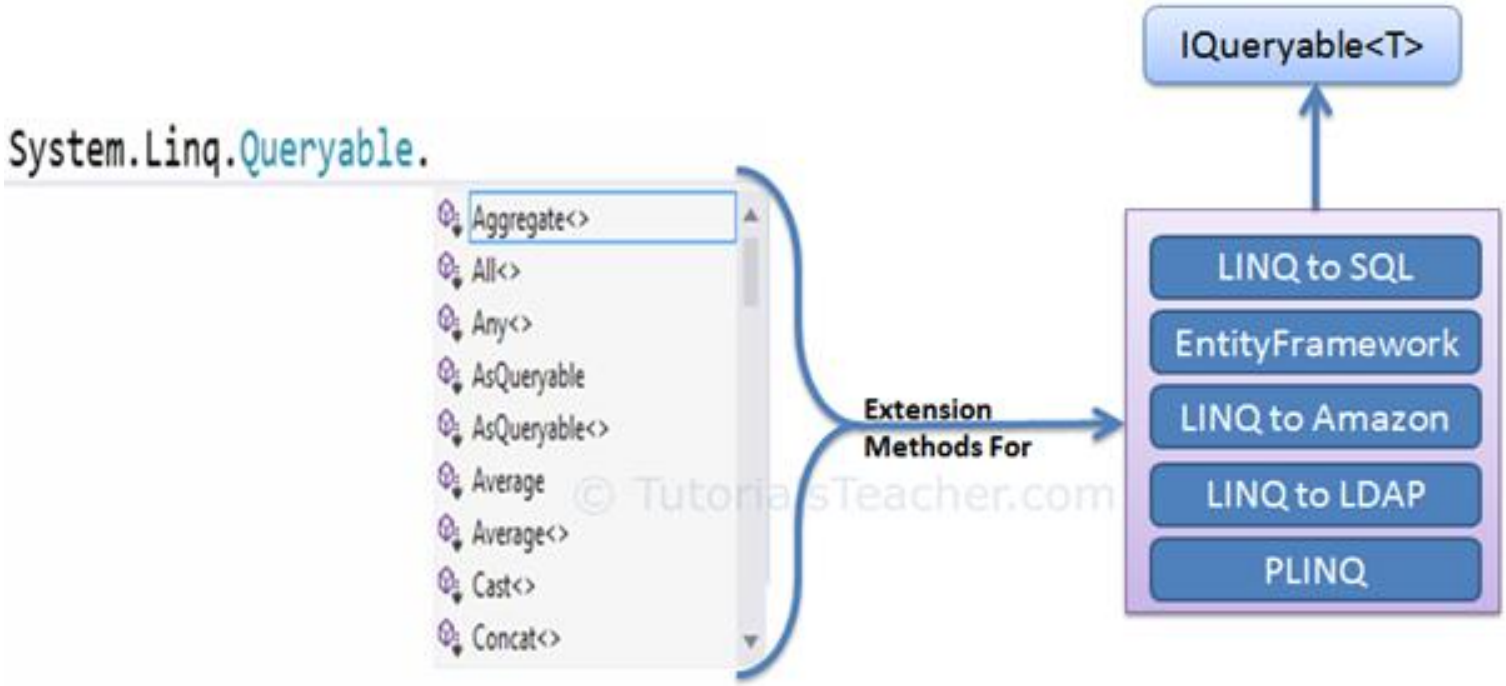
IEnumerable<T>

The static **Enumerable** class includes extension methods for classes that implements IEnumerable<T> interface.



IQueryable<T>

The static **Queryable** class includes extension methods for classes that implements IQueryable<T> interface



Query Syntax

```
var vendorQuery = from v in vendors
                  where v.CompanyName.Contains("Toy")
                  orderby v.CompanyName
                  select v;
```

Method Syntax

```
var vendorQuery = vendors
                  .Where(v => v.CompanyName.Contains("Toy"))
                  .OrderBy(v=> v.CompanyName);
```

LINQ Query Syntax

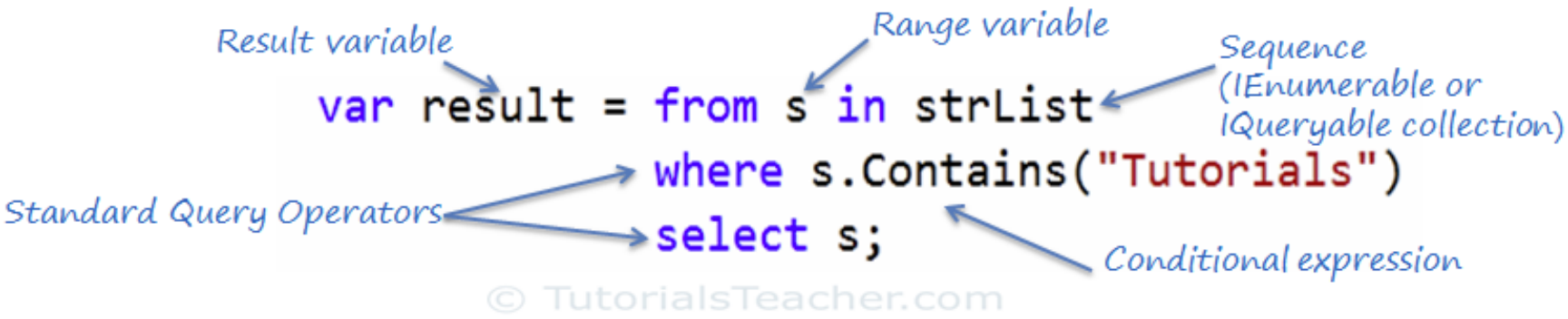
http://www.tutorialsteacher.com/linq/linq-query-syntax

Linq Query Syntax

- is same like SQL syntax
- starts with ***from*** clause and
  can be end with ***select*** or ***groupBy*** clause
  - use various other operators like filtering, joining, grouping, sorting operators to construct the desired result.
  - Implicitly typed variable - var
    can be used to hold the result of the LINQ query

Linq Query Syntax

- from <range variable> in <IEnumerable<T>
  or IQueryable<T> Collection>
  <Standard Query Operators> <lambda expression>
  <select or groupBy operator> <result formation>



Linq Query Syntax Example

```
// 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 Query Syntax to find out teenager students
var teenAgerStudent = from s in studentList
                      where s.Age > 12 && s.Age < 20
                      select s;
```

Query Syntax

```
var vendorQuery = from v in vendors
                  where v.CompanyName.Contains("Toy")
                  orderby v.CompanyName
                  select v;
```

Method Syntax

```
var vendorQuery = vendors
                  .Where(v => v.CompanyName.Contains("Toy"))
                  .OrderBy(v=> v.CompanyName);
```

LINQ Method Syntax

http://www.tutorialsteacher.com/linq/linq-method-syntax

Linq Method Syntax

- is like calling extension method
- allows series of extension methods call
- **var** (implicitly typed variable) can be used to hold the result of the LINQ query

Linq Method Syntax

- Uses extension methods combined with lambda expression

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

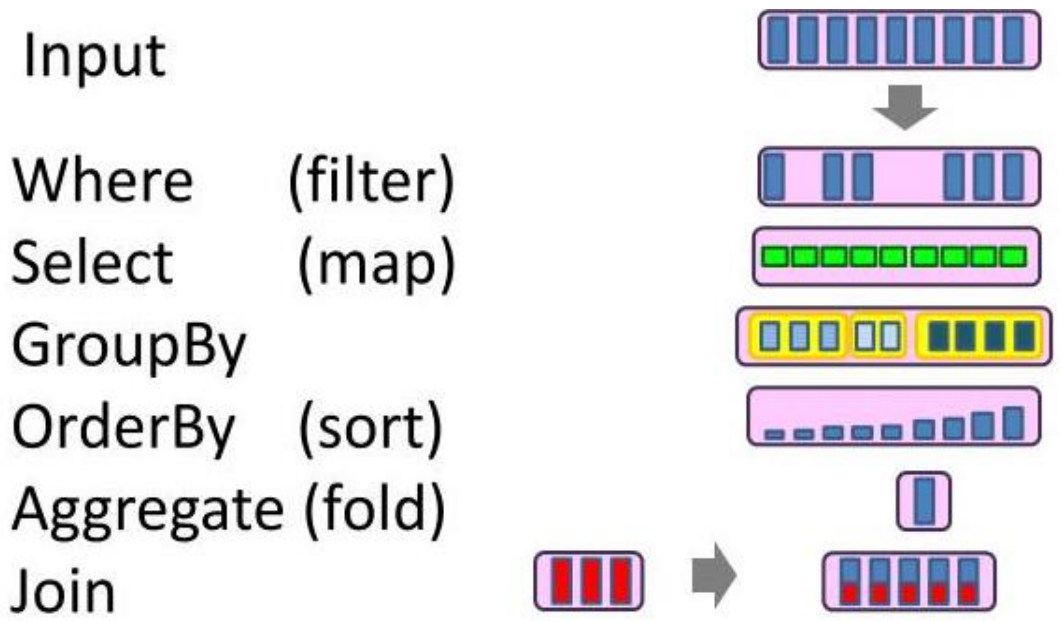

- **Advantages:**
  - Query syntax is automatically converted to method syntax at compilation time
  - Not all LINQ methods can be utilized with query syntax
  - Method syntax is stylistically more similar to other C# code

Linq Method Syntax Example

```
// 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>();
```

Standard Query Operators

- Linq offers over 50 standard query operators that provide different functionalities like
  - Filtering
  - Sorting
  - Grouping
  - Aggregation
  - Concatenation
  - …

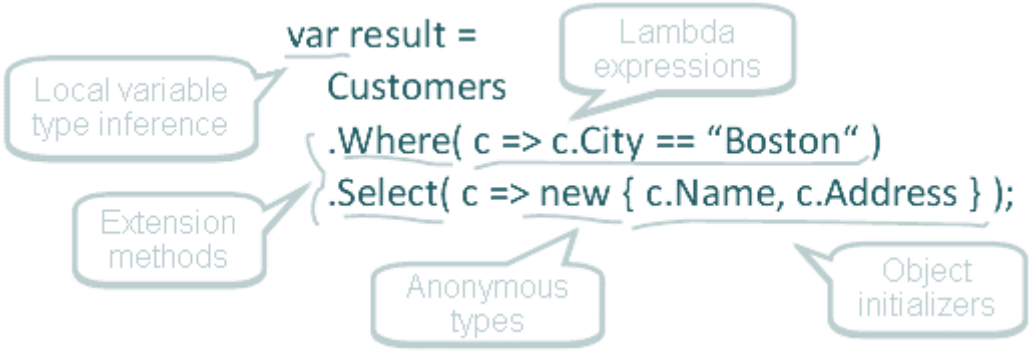


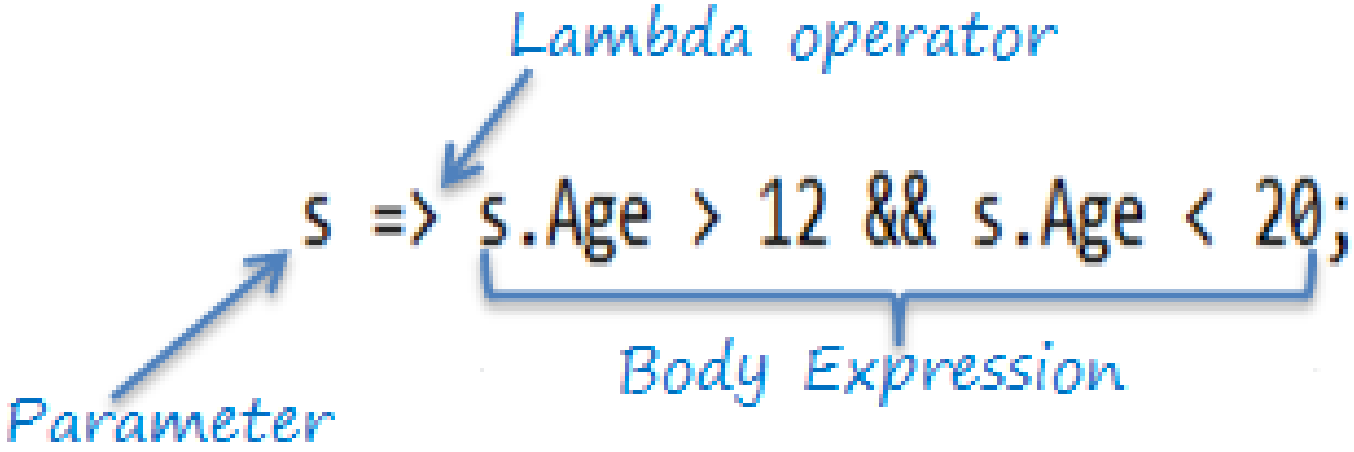
http://www.tutorialsteacher.com/linq/linq-standard-query-operators

Lambda Expression

Anonymos Method - delegate

Funk, Action, Predicate





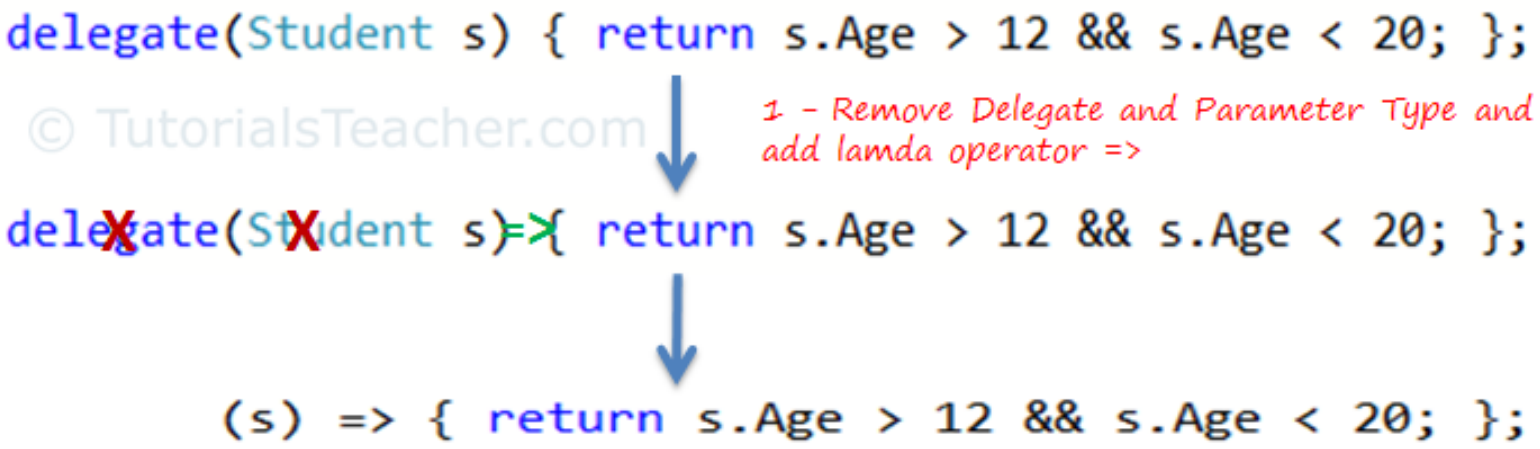
Lambda Expression

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

C# =>

Anonymous Method vs Lambda Expression

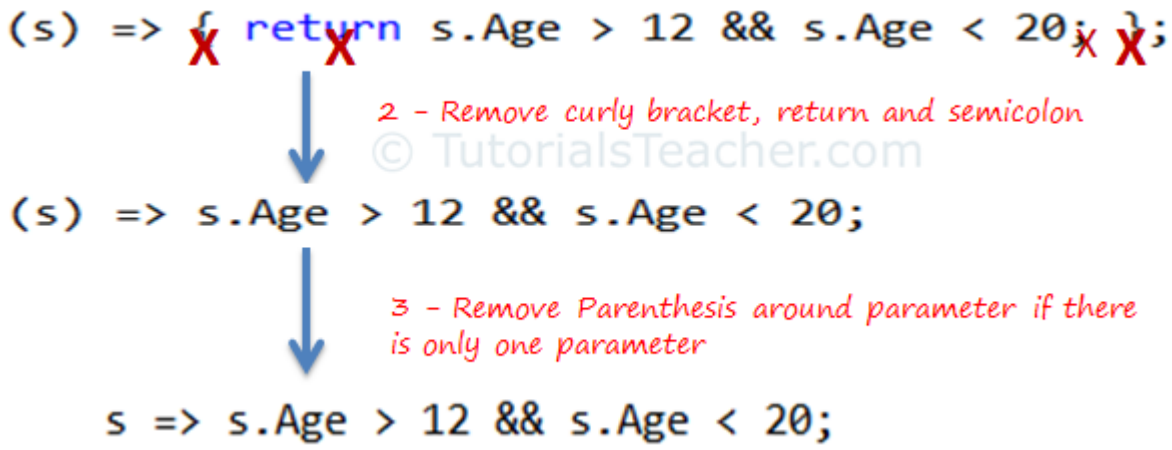
- Anonymous Method Syntax:



- Lambda Expression evolves from anonymous method

Lambda Expression

- Omit Curly braces & semicolon

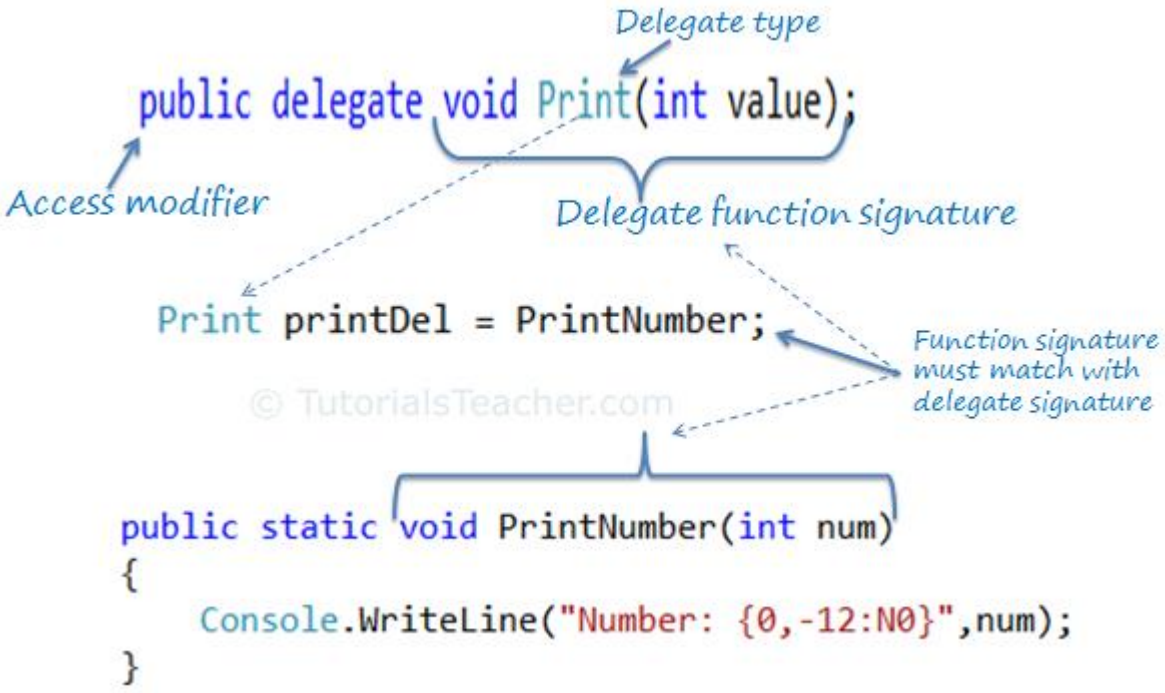


Lambda Expression

- With multiple parameters
  - (s, youngAge) => s.Age >= youngage;
- Without any parameters
  - () => Console.WriteLine("lambda expression")
- Multiple statements in body expression
  - (s, youngAge) => { Console.WriteLine(“lambda expression "); return s.Age >= youngAge; }
- Local variable in Lambda Expression body
  - s => { int youngAge = 18; Console.WriteLine("Lambda expression"); return s.Age >= youngAge; }

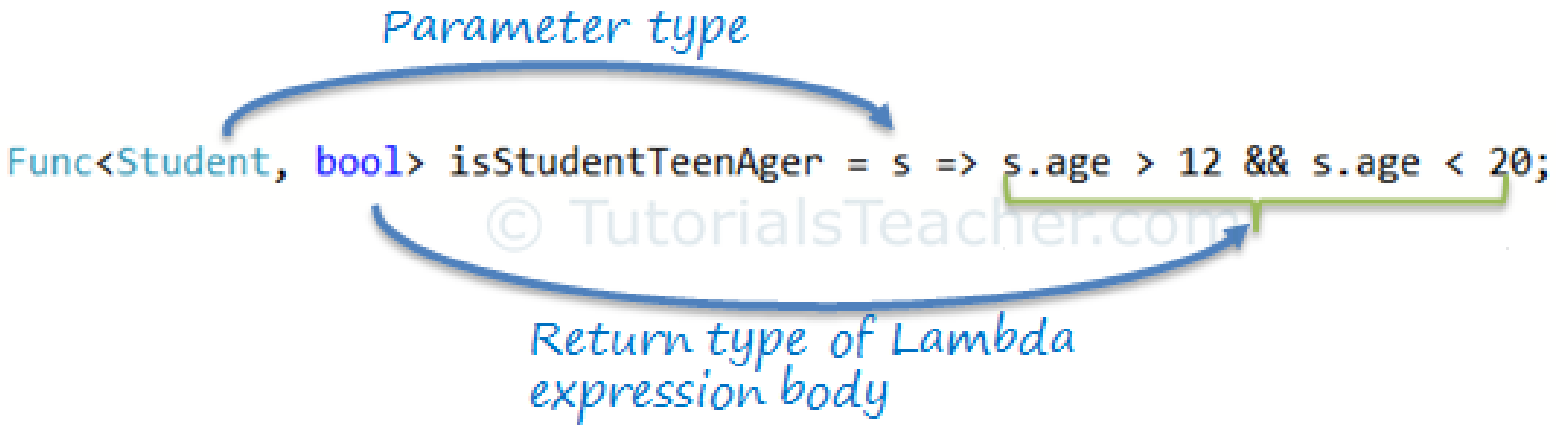
Delegate

- http://www.tutorialsteacher.com/csharp/csharp-delegates



Func

- http://www.tutorialsteacher.com/csharp/csharp-func-delegate

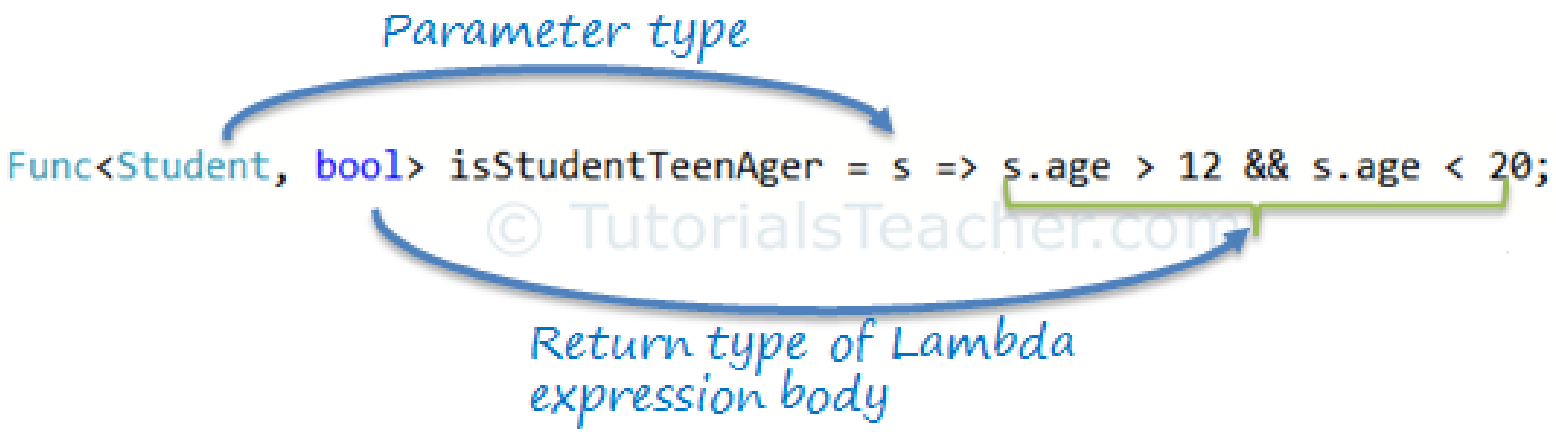


Action

- http://www.tutorialsteacher.com/csharp/csharp-action-delegate

- returns void

```
public delegate void Action<in T>(
        T obj
)
```

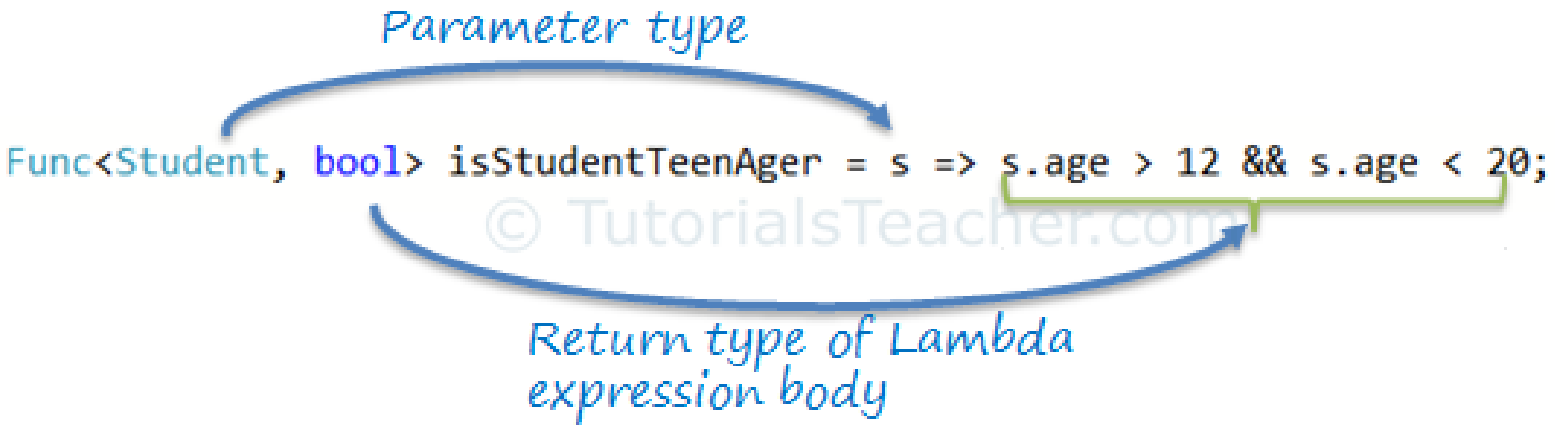


Predicate

- http://www.tutorialsteacher.com/csharp/csharp-predicate

```
public delegate bool Predicate<in T>(
        T obj
)
```

- returns bool



Lambda Expression - Summary

- is a shorter way of representing anonymous method.
- Syntax: parameters => body expression
  - can have zero parameter.
  - can have multiple parameters in parenthesis ().
  - can have multiple statements in body expression in curly brackets {}
- can be assigned to Func, Action or Predicate delegate
- can be invoked in a similar way to delegate

Var - implicitly typed variable

http://www.tutorialsteacher.com/csharp/csharp-var-implicit-typed-local-variable

var is used to hold the reference of anonymous types

var can have a different type based on its value

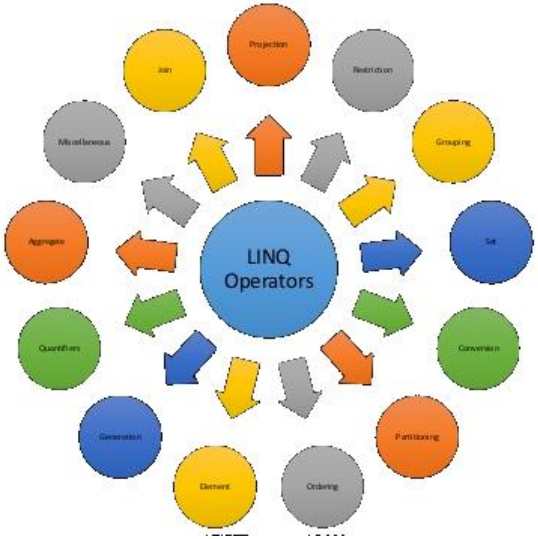
```
static void Main(string[] args)
{
    var i = 10;
    Console.WriteLine("Type of i is {0}",i.GetType().ToString());

    var str = "Hello World!!";
    Console.WriteLine("Type of str is {0}", str.GetType().ToString());

    var d = 100.50d;
    Console.WriteLine("Type of d is {0}", d.GetType().ToString());

    var b = true;
    Console.WriteLine("Type of b is {0}", b.GetType().ToString());

}
```

Output:
Type of i is System.Int32
Type of str is System.String
Type of d is System.Double
Type of b is System.Boolean

The LINQ to Objects provider contains a handy set of standard query operators to work with in-memory IEnumerable<T> collections.

LINQ to Object

https://www.tutorialspoint.com/linq/linq_objects.htm



Standard Query Operators

in Query Syntax & Method Syntax

can be classified based
on the functionality they provide.

Standard Query Operators – Part 1

Classification	Standard Query Operators
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

Standard Query Operators – Part 2

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

Filtering Operators

Where & TypeOf

Filtering operators: Where & OfType

- filter the sequence (collection) based on some given criteria

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.

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

Filtering Operators - Where

- is a Linq extension method
- filters the collection based on a given criteria expression and returns a new collection

- criteria can be specified as
  - lambda expression or
  - Func delegate type

Where

- Select all books which include „tutorial“

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

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

Output:
C# Tutorials
VB.NET Tutorials
MVC Tutorials

Where

- with two conditions
- query syntax

Output:
Teen age Students:
John
Bill
Ron

```
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

- with method syntax

Output:
Teen age Students:
John
Bill
Ron

```
// 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 teenAgerStudent = studentList.Where(s=>s.Age>12 && s.Age<20);

Console.WriteLine("Teen age Students:");

foreach(Student std in teenAgerStudent){
    Console.WriteLine(std.StudentName);
}
```

Func type delegate

- Func type delegate with an anonymous method to pass as a predicate function

```
Func<Student,bool> isTeenAger = delegate(Student s) {
                                    return s.Age > 12 && s.Age < 20;
                                };

var filteredResult = from s in studentList
                     where isTeenAger(s)
                     select s;
```

any method that matches

- with one of Where() method overloads

```
public static void Main()
{
    var filteredResult = from s in studentList
                         where isTeenAger(s)
                         select s;
}

public static bool IsTeenAger(Student stud)
{
    return stud.Age > 12 && stud.Age < 20;
}
```

Multiple Where Clause

- In Query Syntax

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

- In Method Syntax

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

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

TypeOf

```
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;

var intResult = from s in mixedList.OfType<int>()
                select s;

var stdResult = from s in mixedList.OfType<Student>()
                select s;

foreach (var str in stringResult)
    Console.WriteLine(str);

foreach (var integer in intResult)
    Console.WriteLine(integer);

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

Output:
One
Two
0
3
Bill

Sorting Operators:

OrderBy & OrderByDescending

Sorting operator: OrderBy, …

- arranges the 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.

http://www.tutorialsteacher.com/linq/linq-sorting-operators-orderby-orderbydescending

Oderby – Query Syntax

```
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;
```

Bill	Steve
John	Ron
Ram	Ram
Ron	John
Steve	Bill

Orderby – Method Syntax

```
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);
```

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

Multiple Sorting

```
StudentName: Bill, Age: 25
StudentName: John, Age: 18
StudentName: Ram, Age: 18
StudentName: Ram, Age: 20
StudentName: Ron, Age: 19
StudentName: Steve, Age: 15
```

```
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 };
```

Multiple sorting in method syntax works differently. Use ThenBy or ThenByDecending extension methods for secondary sorting.

ThenBy – Extention Method

- OrderBy and ThenBy sorts collections in ascending order by default.
- ThenBy or ThenByDescending is used for second level sorting
- ThenByDescending method sorts the collection in decending order on another field

```
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 }
};
```

```
StudentName: Bill, Age: 25
StudentName: John, Age: 18
StudentName: Ram, Age: 18
StudentName: Ram, Age: 20
StudentName: Ron, Age: 19
StudentName: Steve, Age: 15
```

```
var thenByResult = studentList.OrderBy(s => s.StudentName).ThenBy(s => s.Age);

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

```
StudentName: Bill, Age: 25
StudentName: John, Age: 18
StudentName: Ram, Age: 20
StudentName: Ram, Age: 18
StudentName: Ron, Age: 19
StudentName: Steve, Age: 15
```

Sorting operators - Summery

- LINQ includes five sorting operators:
  - OrderBy, OrderByDescending, ThenBy, ThenByDescending and Reverse

- LINQ query syntax does not support OrderByDescending, ThenBy, ThenByDescending and Reverse.
  - It only supports 'Order By' clause with 'ascending' and 'descending' sorting direction.
- LINQ query syntax supports multiple sorting fields seperated by comma whereas you have to use ThenBy & ThenByDescending methods for secondary sorting.

Grouping Operators

GroupBy & ToLookup

```
var groupedResult = studentList.GroupBy(
```

▲ 1 of 8 ▼ (extension) IEnumerable<IGrouping<TKey,Student>> IEnumerable<Student>.GroupBy(**Func<Student,TKey> keySelector**)

Groups the elements of a sequence according to a specified key selector function.

keySelector: *A function to extract the key for each element.*

Grouping Operators

- create a group of elements based on the given key
- 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.

http://www.tutorialsteacher.com/linq/linq-grouping-operator-groupby-tolookup

GroupBy in Query Syntax C#

```
AgeGroup: 18
StudentName: John
StudentName: Bill
AgeGroup: 21
StudentName: Steve
StudentName: Abram
AgeGroup: 20
StudentName: Ram
```

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

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

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

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

GroupBy in Method Syntax C#

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

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

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

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

```
AgeGroup: 18
StudentName: John
StudentName: Bill
AgeGroup: 21
StudentName: Steve
StudentName: Abram
AgeGroup: 20
StudentName: Ram
```

ToLookup

- ToLookup is the same as GroupBy;
  - the only difference is GroupBy execution is deferred, whereas ToLookup execution is immediate

- ToLookup is only applicable in Method syntax.
- **ToLookup is not supported in the query syntax.**

ToLookup

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

var lookupResult = studentList.ToLookup(s => s.age);

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

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

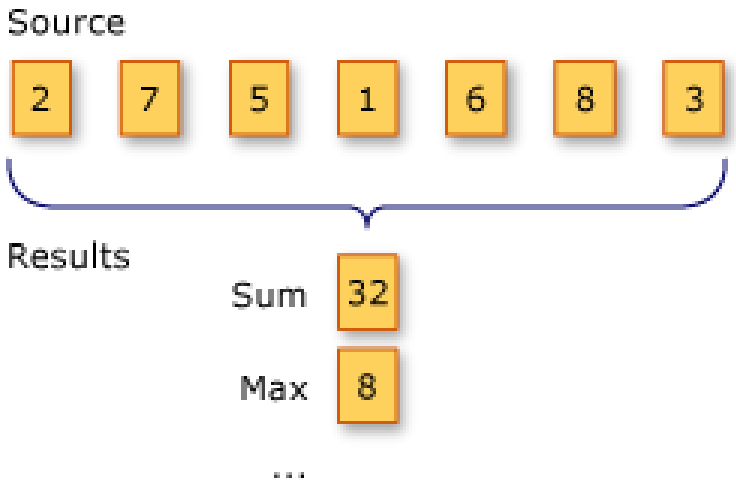
```
AgeGroup: 18
StudentName: John
StudentName: Bill
AgeGroup: 21
StudentName: Steve
StudentName: Abram
AgeGroup: 20
StudentName: Ram
```

Grouping Operators - Summery

- GroupBy & ToLookup return a collection
  - that has a key and
  - an inner collection based on a key field value

- execution of GroupBy is deferred whereas that of ToLookup is immediate

- A LINQ query syntax can be end with the GroupBy or Select clause



Aggregation Operators

Average, Count, Max, Min, Sum,

http://www.tutorialsteacher.com/linq/linq-aggregation-operator-aggregate

Aggregation Operators

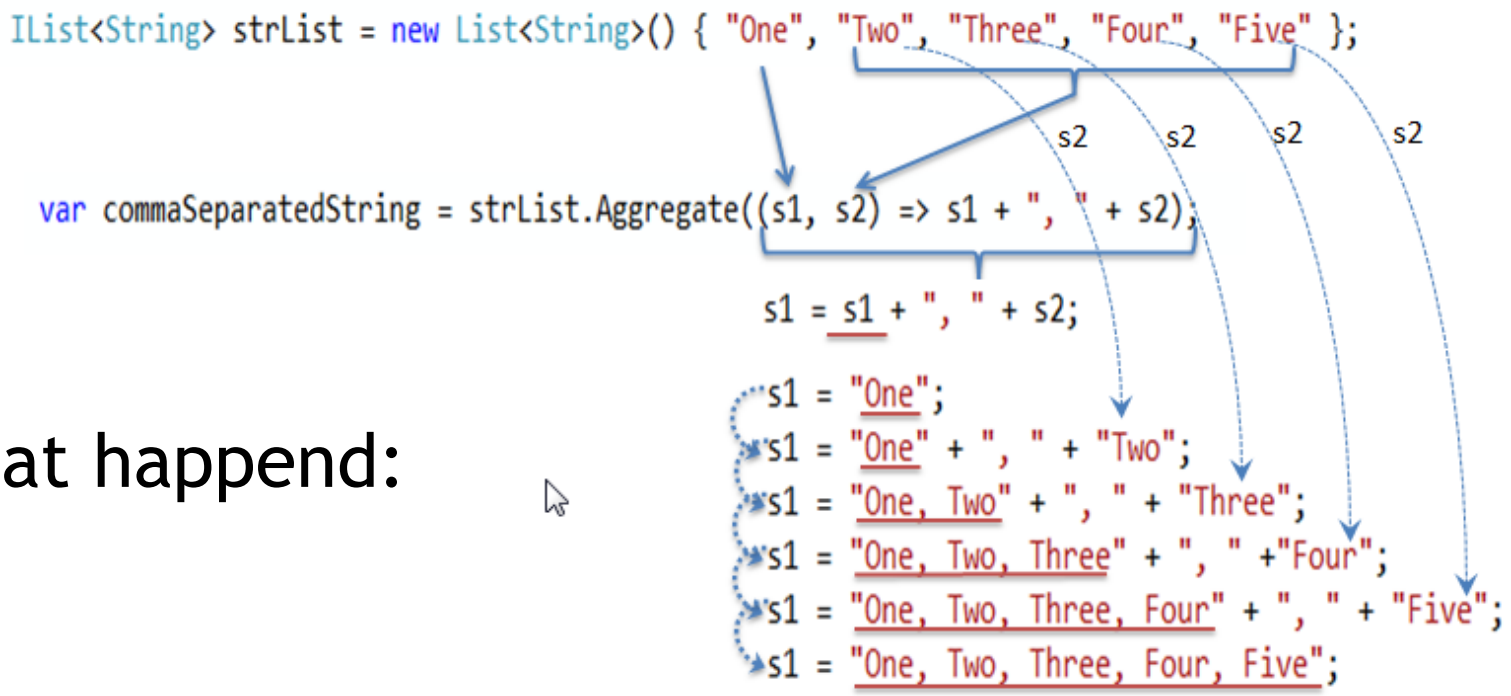
- perform mathematical operations on the numeric property of the elements in the collection

Method	Description
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.

Aggregate Operation

Output:
One, Two, Three, Four, Five

```
IList<String> strList = new List<String>()
{ "One", "Two", "Three", "Four", "Five"};

var commaSeperatedString =
    strList.Aggregate((s1, s2) => s1 + ", " + s2);

Console.WriteLine(commaSeperatedString);
```



- what happend:

Aggreagte with Studentnames

```
public static void Main()
{
    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 }
    };

    string commaSeparatedStudentNames = studentList.Aggregate<Student, string>(
                                        "Student Names: ",  // seed value
                                        (str, s) => str += s.StudentName + "," );

    Console.WriteLine(commaSeparatedStudentNames);
```

- What‘s the Output?
  - Student Names: John,Moin,Bill,Ram,Ron,

Aggregate operator is **Not Supported** with query syntax in C#

Average Method

```
IList<int> intList = new List<int>>() { 10, 20, 30 };

var avg = intList.Average();

Console.WriteLine("Average: {0}", avg);
```

- What‘s the output?

Average: 20

Count Method

```
IList<int> intList = new List<int>() { 10, 21, 30, 45, 50 };

var totalElements = intList.Count();

Console.WriteLine("Total Elements: {0}", totalElements);

var evenElements = intList.Count(i => i%2 == 0);

Console.WriteLine("Even Elements: {0}", evenElements);
```

- What‘s the output?

Total Elements: 5
Even Elements: 3

Sum Method

```
IList<int> intList = new List<int>()
{ 1, 2, 3, 4, 5, 7 };

var total = intList.Sum();

Console.WriteLine("Sum: {0}", total);

var sumOfEvenElements = intList.Sum(i => {
    if(i%2 == 0)
        return i;

    return 0;
});

Console.WriteLine("Sum of Even Element: {0}",
                  sumOfEvenElements );
```

- Whats the output?

Sum: 22
Sum of Even Element: 6

Sum on Age

```
public class Student
{
    public int StudentID { get; set; }
    public string StudentName { get; set; }
    public int Age { get; set; }
}
```

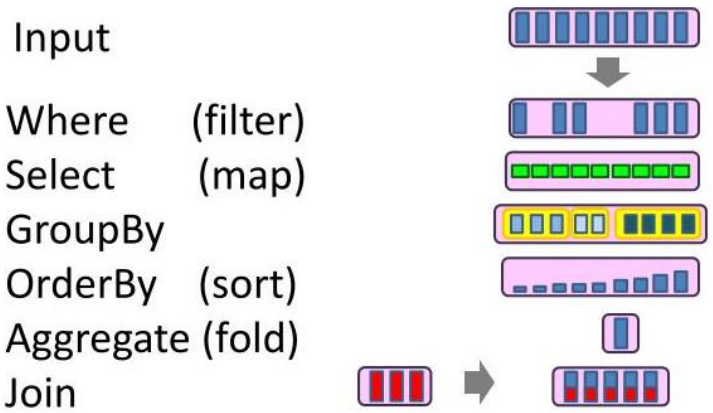
```
IList<Student> studentList = new List<Student>() {
    new Student() { StudentID = 1, StudentName = "John", Age = 20 } ,
    new Student() { StudentID = 2, StudentName = "Moin",  Age = 20 } ,
    new Student() { StudentID = 3, StudentName = "Bill",  Age = 25 } ,
    new Student() { StudentID = 4, StudentName = "Ram" , Age = 20 } ,
    new Student() { StudentID = 5, StudentName = "Steve" , Age = 17 }
};
 var sumOfAge =

Console.WriteLine("Sum of all student's age: {0}", sumOfAge);

var totalAdults = s
    if(s.Age >= 18)
    else
});
Console.WriteLine("Total Adult Students: {0}", totalAdults);
```

Sum of all student's age: 102
Total Adult Students: 4

Standard Query Operators

Extension Methods…

Write your own &
use the existing ones



Create Extension Methods

- Write your own WordCount-Method:

```
public static void Main(string[] args)
{
    string s = "The quick brown fox jumped over the lazy dog.";
    int i = s.WordCount();
    System.Console.WriteLine("Word count of s is {0}", i);
}
```

Output:
Word count of s is 9

Create Extension Method: WordCount

```
namespace CustomExtensions
{
    //Extension methods must be defined in a static class
    public static class StringExtension
    {
        // This is the extension method.
        // The first parameter takes the "this" modifier
        // and specifies the type for which the method is defined.
        public static int WordCount(this String str)
        {
            return str.Split(new char[] {' ', '.','?'},
                             StringSplitOptions.RemoveEmptyEntries).Length;
        }
    }
}

    public static void Main(string[] args)
    {
        string s = "The quick brown fox jumped over the lazy dog.";
        //  Call the method as if it were an
        //  instance method on the type. Note that the first
        //  parameter is not specified by the calling code.
        int i = s.WordCount();
        System.Console.WriteLine("Word count of s is {0}", i);
    }
```

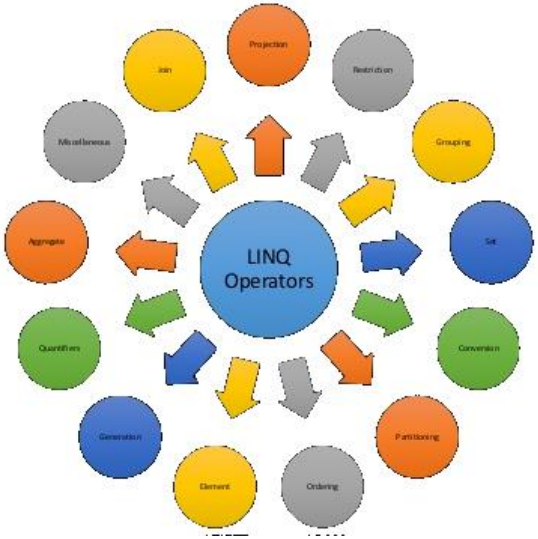
Output:
Word count of s is 9

To ObservableCollecition

```
public static class Extension
{
    public static ObservableCollection<T> ToObservableCollection<T>(this IEnumerable<T> list)
    {
        ObservableCollection<T> collection = new ObservableCollection<T>();
        foreach (T item in list)
        {
            collection.Add(item);
        }
        return collection;
    }
}
```

ExtensionGetCars

```
public static class Extension
{
    public static List<Car> ExtensionGetCarsByColor(this CarRepository c, string s)
    {
        List<Car> carList = (from car in c.c.Cars
                             where car.Color == s
                             select car).ToList();
        return carList;
    }

    public static List<Car> ExtensionGetCarsByMake(this CarRepository c, string s)
    {
        return c.GetCarByMake(s);
    }
}
```



Lots and lots of existing extension methods…

Set Operators
Partitioning Operators
Quantifier Operators
Element Operators
Generation Operators

Linq operators

- a brief overview of all kinds of operators …

Aggregate	Conversion	Ordering	Partitioning	Sets
Aggregate	Cast	OrderBy	Skip	Concat
Average	OfType	ThenBy	SkipWhile	Distinct
Count	ToArray	Descending	Take	Except
Max	ToDictionary	Reverse	TakeWhile	Intersect
Min	ToList			Union
Sum	ToLookup			
	ToSequence			
	and many others			

Concat


```
IList<Student> studentList1 = 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 = 5, StudentName = "Ron" , Age = 19 }
};

IList<Student> studentList2 = new List<Student>() {
    new Student() { StudentID = 3, StudentName = "Bill",  Age = 25 } ,
    new Student() { StudentID = 5, StudentName = "Ron" , Age = 19 } ,
    new Student() { StudentID = 6, StudentName = "Ram" , Age = 29 }
};

var resultedCol = studentList1.Concat(studentList2);

foreach(Student std in resultedCol)
    Console.WriteLine(std.StudentName);
```

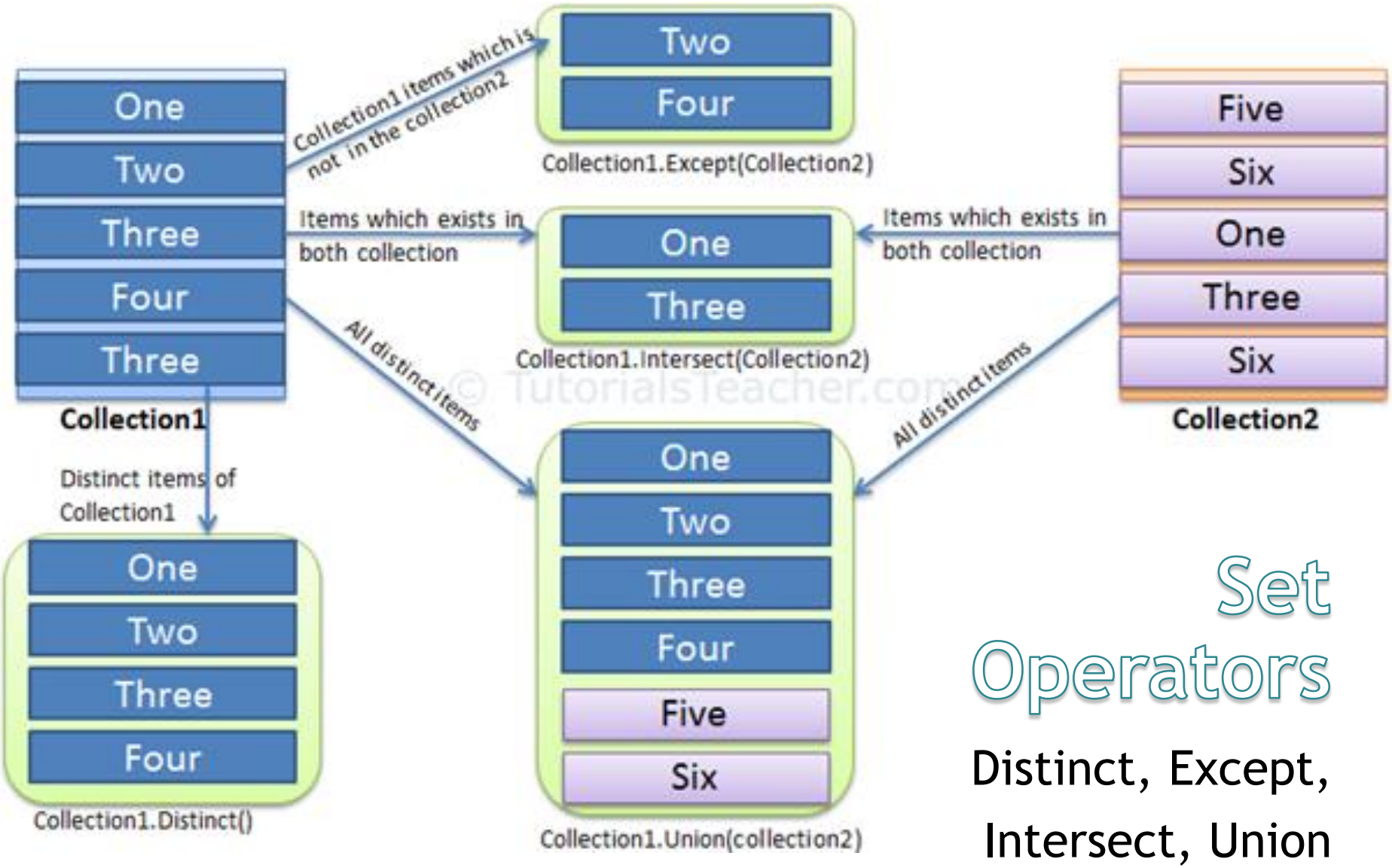
John
Steve
Bill
Ron
Bill
Ron
Ram

Set Operators: Distinct, Except, Intersect & Union

The following table lists all Set operators available in LINQ.

Set Operators	Usage
Distinct	Returns distinct values from a collection (removes duplicate)
Except	Returns the difference between two sequences, which means the elements of one collection that do not appear in the second collection.
Intersect	Returns the intersection of two sequences, which means elements that appear in both the collections.
Union	Returns unique elements from two sequences, which means unique elements that appear in either of the two sequences.

http://www.tutorialsteacher.com/linq/linq-set-operators-distinct



Set Operators

Distinct, Except, Intersect, Union

Comperer

Distinct, Except, … extension method
doesn't compare values of complex type objects

in order to compare the values of complex types,
need to implement the IEqualityComparer<T> interface

StudentComperer

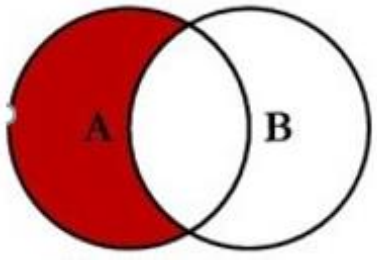
```
public class Student
{
    public int StudentID { get; set; }
    public string StudentName { get; set; }
    public int Age { get; set; }
}

class StudentComparer : IEqualityComparer<Student>
{
    public bool Equals(Student x, Student y)
    {
        if (x.StudentID == y.StudentID
                && x.StudentName.ToLower() == y.StudentName.ToLower())
            return true;

        return false;
    }

    public int GetHashCode(Student obj)
    {
        return obj.StudentID.GetHashCode();
    }
}
```

Distinct with StudentComperer

```
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 = 3, StudentName = "Bill",  Age = 25 } ,
        new Student() { StudentID = 3, StudentName = "Bill",  Age = 25 } ,
        new Student() { StudentID = 3, StudentName = "Bill",  Age = 25 } ,
        new Student() { StudentID = 5, StudentName = "Ron" , Age = 19 }
    };


var distinctStudents = studentList.Distinct(new StudentComparer());

foreach(Student std in distinctStudents)
    Console.WriteLine(std.StudentName);
```

John
Steve
Bill
Ron

Except with StudentComparer

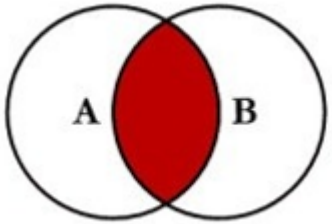


- The Except() method returns a new collection with elements from the first collection which do not exist in the second collection:

```
IList<Student> studentList1 = 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 = 5, StudentName = "Ron" , Age = 19 }
};

IList<Student> studentList2 = new List<Student>() {
    new Student() { StudentID = 3, StudentName = "Bill",  Age = 25 } ,
    new Student() { StudentID = 5, StudentName = "Ron" , Age = 19 } ,
    new Student() { StudentID = 6, StudentName = "Ram" , Age = 29 }
};

var resultedCol = studentList1.Except(studentList2,new StudentComparer());

foreach(Student std in resultedCol)
    Console.WriteLine(std.StudentName);
```

John
Steve

Intersect with StudentComparer

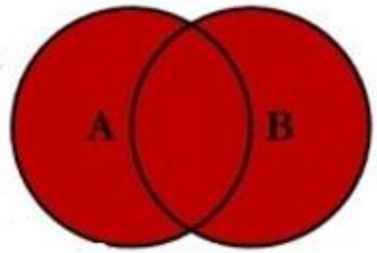


- returns a new collection that includes common elements that exists in both the collection

```
IList<Student> studentList1 = 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 = 5, StudentName = "Ron" , Age = 19 }
};

IList<Student> studentList2 = new List<Student>() {
        new Student() { StudentID = 3, StudentName = "Bill",  Age = 25 } ,
        new Student() { StudentID = 5, StudentName = "Ron" , Age = 19 } ,
        new Student() { StudentID = 6, StudentName = "Ram" , Age = 29 }
    };

var resultedCol = studentList1.Intersect(studentList2,new StudentComparer());

foreach(Student std in resultedCol)
    Console.WriteLine(std.StudentName);
```

Bill
Ron

Union with StudentComparer




```
IList<Student> studentList1 = 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 = 5, StudentName = "Ron" , Age = 19 }
};

IList<Student> studentList2 = new List<Student>() {
        new Student() { StudentID = 3, StudentName = "Bill",  Age = 25 } ,
        new Student() { StudentID = 5, StudentName = "Ron" , Age = 19 },
    new Student() { StudentID = 5, StudentName = "Micheal" , Age = 19 }
    };

var result = studentList1.Union(studentList2,new StudentComparer());

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

John
Steve
Bill
Ron
Micheal

Partitioning Operators: Take, …

- split the sequence (collection) into two parts and returns 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 given collection until the specified condition is true. If the first element itself doesn't satisfy the condition then returns an empty collection.

http://www.tutorialsteacher.com/linq/linq-partitioning-operators-skip-skipwhile

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

Operator	Description
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

http://www.tutorialsteacher.com/linq/linq-quantifier-operators

Element Operators

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

Element Operators (Methods)	Description
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.

http://www.tutorialsteacher.com/linq/linq-element-operator-elementat-elementatordefault

Element Operators

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

Element Operators	Description
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.

http://www.tutorialsteacher.com/linq/linq-element-operator-elementat-elementatordefault

Single Or SingleOrDefault Example

```
IList<int> oneElementList =
    new List<int>() { 7 };
IList<int> intList =
    new List<int>() { 7, 10, 21, 30, 45, 50, 87 };
IList<string> strList =
    new List<string>() { null, "Two", "Three", "Four", "Five" };
IList<string> emptyList = new List<string>();

Console.WriteLine("The only element in oneElementList: {0}",
                  oneElementList.Single());
Console.WriteLine("The only element in oneElementList: {0}",
              oneElementList.SingleOrDefault());
Console.WriteLine("Element in emptyList: {0}",
                  emptyList.SingleOrDefault());
Console.WriteLine("The only element which is less than 10 in intList: {0}",
              intList.Single(i => i < 10));


//Followings throw an exception
//Console.WriteLine("The only Element in intList: {0}", intList.Single());
//Console.WriteLine("The only Element in intList: {0}", intList.SingleOrDefault());
//Console.WriteLine("The only Element in emptyList: {0}", emptyList.Single());
```

```
The only element in oneElementList: 7
The only element in oneElementList: 7
Element in emptyList:
The only element which is less than 10 in intList: 7
```

Generation Operators:

- LINQ includes generation operators DefaultIfEmpty, Empty, Range & Repeat.

Method	Description
Empty	Returns an empty collection
Range	Generates collection of IEnumerable<T> type with specified number of elements with sequential values, starting from first element.
Repeat	Generates a collection of IEnumerable<T> type with specified number of elements and each element contains same specified value.

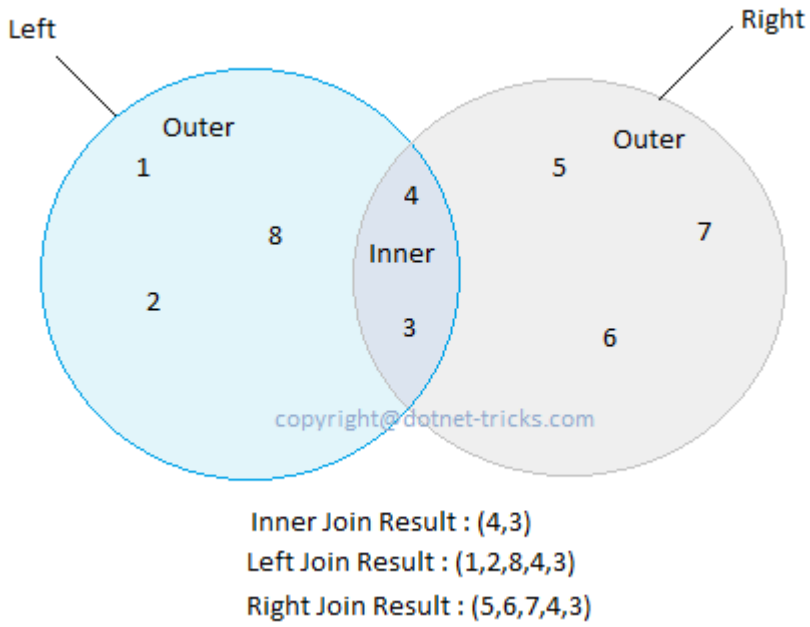
http://www.tutorialsteacher.com/linq/linq-generation-operators

DefaultIfEmpty

- Retuns nothing or a default value, instead of an empty collection:

```
IList<string> emptyList = new List<string>();

var newList1 = emptyList.DefaultIfEmpty();
var newList2 = emptyList.DefaultIfEmpty("None");

Console.WriteLine("Count: {0}" , newList1.Count());
Console.WriteLine("Value: {0}" , newList1.ElementAt(0));

Console.WriteLine("Count: {0}" , newList2.Count());
Console.WriteLine("Value: {0}" , newList2.ElementAt(0));
```

Output:
Count: 1
Value:
Count: 1
Value: None



Join

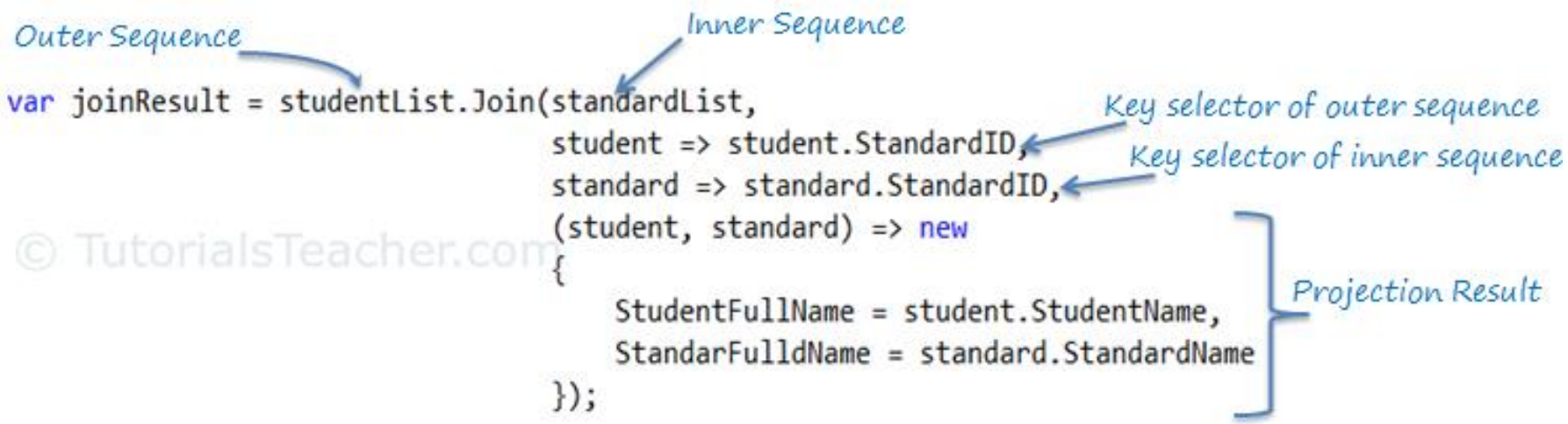
operates on two collections,
inner collection & outer collection.

returns a new collection
that contains elements from both the collections
which satisfies specified expression

http://www.tutorialsteacher.com/linq/linq-joining-operator-join

Join Syntax

- joins two sequences (collections) based on a key and returns a resulted sequence



Join Operator – Inner Join:

- Inner Join:

Output:
One
Two

```
IList<string> strList1 = new List<string>() {
    "One",
    "Two",
    "Three",
    "Four"
};

IList<string> strList2 = new List<string>() {
    "One",
    "Two",
    "Five",
    "Six"
};

var innerJoin = strList1.Join(strList2,
                    str1 => str1,
                    str2 => str2,
                    (str1, str2) => str1);
```

Join Operator

```
public class Student{
    public int StudentID { get; set; }
    public string StudentName { get; set; }
    public int StandardID { get; set; }
}

public class Standard{
    public int StandardID { get; set; }
    public string StandardName { get; set; }
}
```

```
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
                  });
```

Output:

John - Standard 1
Moin - Standard 1
Bill - Standard 2
Ram - Standard 2

Join in Query Syntax

- works slightly different than method syntax
- requires outer sequence,
  inner sequence,
  key selector
  and result selector.
- 'on' keyword is used
  for key selector
  where left side of 'equals'
  operator is outerKeySelector
  and right side of 'equals' is innerKeySelector

```
from ... in outerSequence

join ... in innerSequence

on outerKey equals innerKey

select ...
```

Join Operator 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
                           };
```

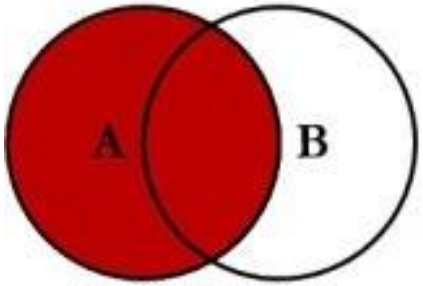
John - Standard 1
Moin - Standard 1
Bill - Standard 2
Ram - Standard 2

Join Operator - Summery

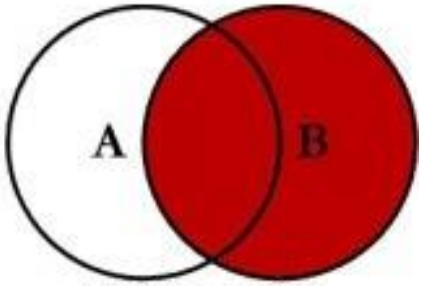
- Use the equals operator to match key selector in query syntax. == is not valid

- Join and GroupJoin are joining operators.
- Join is like inner join of SQL
  - It returns a new collection that contains common elements from two collections whosh keys matches.
- Join operates on two sequences inner sequence and outer sequence and produces a result sequence

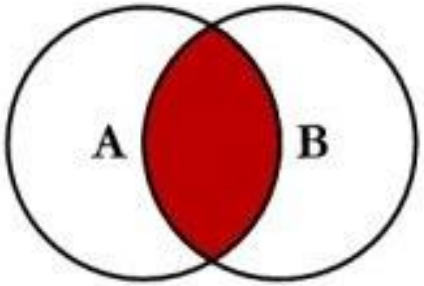
SQL JOINS



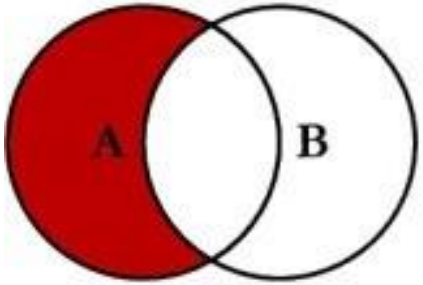

```
SELECT <select_list>
FROM TableA A
LEFT JOIN TableB B
ON A.Key = B.Key
```



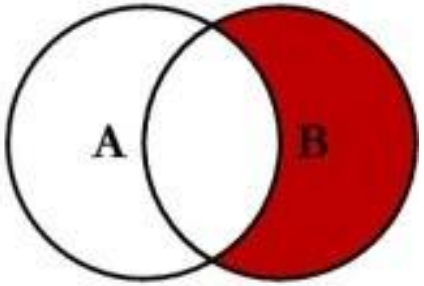

```
SELECT <select_list>
FROM TableA A
RIGHT JOIN TableB B
ON A.Key = B.Key
```



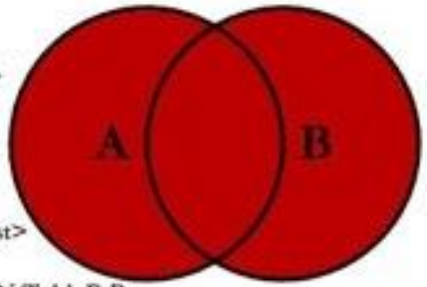

```
SELECT <select_list>
FROM TableA A
INNER JOIN TableB B
ON A.Key = B.Key
```



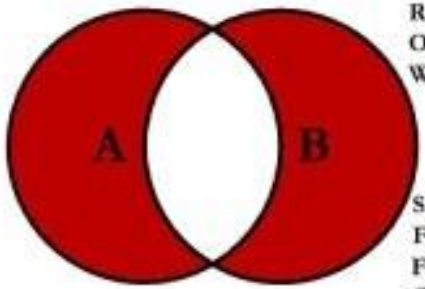

```
SELECT <select_list>
FROM TableA A
LEFT JOIN TableB B
ON A.Key = B.Key
WHERE B.Key IS NULL
```




```
SELECT <select_list>
FROM TableA A
RIGHT JOIN TableB B
ON A.Key = B.Key
WHERE A.Key IS NULL
```




```
SELECT <select_list>
FROM TableA A
FULL OUTER JOIN TableB B
ON A.Key = B.Key
```




```
SELECT <select_list>
FROM TableA A
FULL OUTER JOIN TableB B
ON A.Key = B.Key
WHERE A.Key IS NULL
OR B.Key IS NULL
```

THE STORY OF WHAT'S INSIDE YOUR BODY