**LINQ:**

Console.WriteLine ($"{file.name, -20} : {file. Length, 10: N0}");     //N-->number, 0-->decimal point

Extension Method[Symbol] static class and static method[Symbol] use this method as extension method as ToString()      method.

**LINQ with Lambda expression:**

Below method behaves as where condition in LINQ. (custom filter method)

Public static IEnumerable<T> Filter<T> (this IEnumerable<T> source, Func<T, bool> predicate)

{

 var result=new Result<T> ();

 foreach (var item in source)

 {

     If (predicate (item))

     result. Add (item);

     //or yield return item;

    //yield[Symbol] return IEnumerable and returns control to the caller

 }

}

var query= movies. Filter (m=> m. Year>2000);

var query2= movies. Filter (m=> m. Year>2000). Take (1);      //Like Top operator in SQL query

foreach (var item in query)

{

   Console.WriteLine(item.name);

}

**OR //**

var enumerator= query. GetEnumerator ();

while (enumerator. MoveNext ())

{

    Console.WriteLine(enumerator. Current. Title);

}

**Deferred Execution:**

\* foreach[Symbol] force query to execute.

// Here query executes twice as filter is deferred execution

var query= movies. Filter (m=> m. Year>2000);

Console.WriteLine(query. Count ());                                //executes once

var enumerator= query. GetEnumerator ();    //executes again

while (enumerator. MoveNext ())

{

    Console.WriteLine(enumerator. Current. Title);

}

// To Avoid the above condition, should use ToList () method.

var query=movies. Filter (m=> m. Year>2000).ToList ();   //gets concrete list rather getting abstract result

var query= movies. Where (m=> m. Year>2000). OrderBydescending (m=>m. Rating). Take (10);

// Since where is a deferred execution it returns one value every time to the caller and the time being query will have 10 values (because of Take (10)) it returns the command and won’t execute for entire list.

// where is streaming operator while orderBy is non-streaming operator

public static IEnumerable<double> Random ()

{

  var random=new Random ();

  while(true)

  {

    yield return random. NextDouble ();

  }

}

var numbers=MyLinq.Random(). Where(n=> n>0.5). Take (10);

//Take avoids infinite loop here because yield return the control and where is deferred execution i.e. it checks for condition and return each value.

var query= File.ReadAllLines (path). skip (1). where(line=> line. Length>1). select (Car.ParseFromCsv). ToList ();

//passed extension method in select to manipulate the data

var query=cars. OrderByDescending (c=> c. Rating). ThenBy (c=>c. Name);

Take (1)              [Symbol]gives IEnumerable value

First ()             [Symbol]gives one concrete value

FirstOrDefault ()             [Symbol]handles null value

var result=cars. Any(c=> c. Manufacturer=="Ford"); [Symbol]output is true or false

var result=cars. All(c=> c. Manufacturer=="Ford");

//Any () and Contains () works same

var query= File.ReadAllLines(path). skip (1). where(line=> line. Length>1). ToCar ();

//ToCar () is an extension method which takes list of lines and yield each car at a time converting them into Car class.

select new {car. Manufacture, car. Name, Car. Combined};

var result=cars. Select(c=> new {car. Manufacture, car. Name, Car. Combined});

SelectMany-->Flattens a collection of collections

**JOIN:**

var query=cars. Join (manufacturers,

c=> new {c. Manufacturer, c. Year},           // join based on two values so given an object as condition

m=> new {Manufacturer=m.Name, m. Year}, // name of properties in object should be same

(c, m) => new

{

  m. Headquarters,

  c. Name,

  c. Combined

})

. OrderBy(c=> c. Combined)

. ThenBy(c=> c. Name);

**GroupBy:**

var query =

    from car in cars

    group car by car. manufacturer. ToUpper ()

    into manufacturer

    orderby manufacturer. Key

    select manufacturer;

   //” into” because here we need further manipulation with grouped objects.

(or)

var query =

    cars. GroupBy(c=>c. Manufacturer.ToUpper ())

    . orderBy(g=> g. Key);

foreach (var group in query)

{

   Console.WriteLine(group. Key);

   foreach (var car in group. OrderByDescending (c=>c. Combined). Take (2))

   {

      Console.WriteLine($"\t{car.name}: {car. Combined}");

   }

}

**Aggregate Functions:**

var query= from car in Cars

group car by car. Manufacturer into carGroup

select new

{

  Name=carGroup. Key,

  Max=carGroup.max(c=> c. Combined),

  Min=carGroup.Min (c=> c. Combined),

  Avg=carGHroup.Avereage(c=> c. Combined)

} into result

orderby result. Max descending

select result;

**IQueryable:**

Expression<Func<int, int, int>> add= (x, y) => x+y;

var result=add (3,5);

 It gives error because expression just gives the expression or data structure doesn’t compile the value.

That why in Iqueryable doesn’t compile like iEnumerable rather it just gives an expression.

If we explicitly want to compile, then we have to use compile method.

var result=add. Compile () (3,5); //it compiles the value

//IEnumerable has always to happen in-memory

Db. database. Log=Console.WriteLine;

var query=

from car in db. Cars

group car by car. Manufacturer into manufacturer

select new

{

  Name= manufacturer. Key,

  Cars= (from car in manufacturer

         orderby car. Combined descending

         select car). Take (2)

};