

## Curso de Linq en C#

Linq es una librería de .Net que aparece en el año 2007, constantemente evolucionando.

LINQ son las siglas para Language-Integrated Query, es un lenguaje que se integra a C# y que permite trabajar con colecciones.

Tiene 2 maneras de implementarse o de usarse.

1. Query Expression, consulta muy parecida a SQL pero con sintaxis C#
2. Métodos de Extensión, son métodos que aparecen dentro de una colección y me permiten hacer filtros y diferentes transformaciones con funciones de Linq.

Estos métodos están incluidos dentro de Name space System.linq, viene por defecto en .Net a partir de la versión 5

Linq no es:

1. NO es un lenguaje de programación.
2. NO es un componente SQL.
3. NO es un componente de base de datos.
1. NO es una librería de terceros, Open source como .NET y propia de Microsoft.

Se podría decir que es un Meta-Lenguaje que se basa en un lenguaje de programación y lo extiende.

Es compatible con múltiples lenguajes de programación que sean compatibles con .NET como Visual Basic, F#, Visual C++ y en todas ellas se puede utilizar.

### **Ejemplo**

#### **Ejemplo de Expresion**

```
var unTomate = from t in ArraydeStrings
                where t == "Tomate"
                select t;
```

```
ArraydeStrings.Where(t=> t == "Tomate")
```

#### **Ejemplo de Metodo de extension**

# Comparativa

## Programación declarativa

- Paradigma de la programación.
- Instrucciones donde especifico lo que quiero y no como lo quiero.
- Contraposición a la programación imperativa.
- Fiable y simple.

## Programación imperativa

- Paradigma de la programación.
- Secuencia paso a paso de instrucciones.
- Contraposición a la programación declarativa.
- Código más extenso pero fácil de interpretar.

## Ejemplo JavaScript

```
// Declarativo
const array1 = [1, 2, 3, 4, 5];
array1.forEach(element => console.log(element));

// Imperativo
const array1 = [1, 2, 3, 4, 5];
for (let i = 0; i < array1.length; i++) {
  console.log(array1[i]);
}
```

## Ejemplo C#

```
// Declarativo
var listOfNumbers = new int[] {1,2,3,4,5};
var item1 = listOfNumbers.FirstOrDefault(p=> p==1);
Console.WriteLine(item1);

// Imperativo
var listOfNumbers = new int[] {1,2,3,4,5};
for (int i = 0; i < listOfNumbers.Length; i++) {
  if(listOfNumbers[i] == 1) Console.WriteLine(listOfNumbers[i] );
}
```

NOTA, las expresiones de Linq suelen devolver un IEnumerable este tipo de listas no tienen expresiones Linq por lo que hay que convertirlo en una lista con ToList()

Las funciones imperativas ahorran código de puesto a que las funciones están predefinidas lo que puedes hacer con una línea lo que haces con 4 líneas

## Operadores de Linq

Los podemos agrupar en 3 grandes grupos:

1. Los operadores básicos: Que filtran o hacen modificaciones específicas sobre la colección.
2. Los operadores de agregación: que hacen operaciones sobre todos los datos de la colección.
3. Los operadores de agrupamiento: que nos permiten agrupar los datos de acuerdo con algunos criterios.

Usando el operador Where recibe una condición y filtra la colección igual que en SQL

### Reto operador Where 1

Utilizando el operador Where retorna los libros que fueron publicados después del año 2000.

```
17     }
18     1 reference
19     public IEnumerable<Book> LibrosDespuesde2000()
20     {
21         return librosCollection.Where(p => p.PublishedDate.Year > 2000);
22     }
```

```
8     1 reference
9     public IEnumerable<Book> LibrosDespuesde2000()
10    {
11        //Extension Method
12        //return librosCollection.Where(p => p.PublishedDate.Year > 2000);
13        //Query expesion
14        return from libro in librosCollection where libro.PublishedDate.Year > 2000 select libro;
15    }
16 }
```

TERMINAL   PROBLEMS   OUTPUT   DEBUG CONSOLE   AZURE

|  |     |            |
|--|-----|------------|
| am Foundation Server 2008 in Action      | 344 | 1/12/2008  |
| ownfield Application Development in .NET | 550 | 16/04/2010 |

## Reto operador where 2

Utilizando el operador Where retorna los libros que tengan más de 250 páginas y el título contenga las palabras in Action.

```
public IEnumerable<Book> LibrosMayorA250()
{
    //Extension Method
    //return librosCollection.Where(p => p.PageCount > 250 && p.Title.Contains("in Action"));
    //Query expresion
    return from libro in librosCollection
           where libro.PageCount > 250 && libro.Title.Contains("in Action")
           select libro;
}
```

Veremos 2 nuevos operadores:

**All:** Verifica si **UNA O MAS CONDICIONES** se cumplan en **TODOS LOS ELEMENTOS** de la Colección.

**Any:** Verifica si **UNA O MAS CONDICIONES** se cumplan en **SE CUMPLA EN AL MENOS UN ELEMENTO** de la Colección.

## Reto operador All

Utilizando el operador All verifica que todos los elementos de la colección tenga un valor en el campo Status.

```
public bool TodosLosLibrosTienenStatus()
{
    return librosCollection.All(p => p.Status != string.Empty);
}
```

## Reto operador Any

Utilizando el operador Any verifica si alguno de los libros fue publicado en 2005.

```
public bool SiAlgunLibroFuePublicado2005()  
{  
    return librosCollection.Any(p => p.PublishedDate.Year == 2005);  
}
```

Contains para saber si el string se encuentra en el texto

## Reto operador Contains

Utilizando el operador Contains retorna los elementos que pertenezcan a la categoría de Python.

```
public IEnumerable<Book> LibrosDePython(){  
    //return librosCollection.Where(p => p.Categories.Contains("python"));  
    return from libro in librosCollection  
           where libro.Categories.Contains("Python")  
           select libro;  
}
```

**OrderBy** o **OrderByDescending** : Son lo mismo, uno de manera ascendente y otro de manera descendente.

## Reto operador OrderBy

Utilizando el operador OrderBy retorna todos los elementos que sean de la categoría de Java ordenados por nombre.

```
public IEnumerable<Book> LibrosDeJavaAsc(){  
    return librosCollection  
        .Where(p=> p.Categories.Contains("Java"))  
        .OrderBy(p => p.Title);  
}
```

## Reto operador OrderByDescending

Utilizando el operador OrderByDescending retorna los libros que tengan más de 450 páginas, ordenados por número de páginas en forma descendente.

```
public IEnumerable<Book> LibrosMayor450Desc(){  
    return librosCollection  
        .Where(p=> p.PageCount > 450)  
        .OrderByDescending(p => p.PageCount);  
}
```

### Take y Skip:

**Take:** Te permite tomar una cantidad de elementos que queremos tomar sea filtrada o ordenada

**Skip:** Omitir cierta cantidad de registros y luego seleccionar de allí en adelante todos los registros que continúen.

**TakeLast:** Toma los últimos de la colección

**TakeWhile:** Devuelve los elementos de una secuencia siempre que el valor de una condición especificada sea true y luego omite los elementos restantes.

**SkipWhile:** Omite los elementos de una secuencia en tanto que el valor de una condición especificada sea true y luego devuelve los elementos restantes.

## Reto operador Take

Utilizando el operador Take selecciona los primeros 3 libros con fecha de publicación más reciente que estén categorizados en Java.

```
public IEnumerable<Book> LibrosDeJavaTake(){  
    return librosCollection  
        .Where(p=> p.Categories.Contains("Java"))  
        .OrderByDescending(p => p.PublishedDate)  
        .Take(3);  
}
```

## Selección Dinámica de Elementos

Este método permite seleccionar las columnas que se desea devolver haciendo la consulta más eficiente y/o limpia.

### Reto operador selección dinámica

Utilizando el operador Select selecciona el título y el número de páginas de los primeros 3 libros de la colección.

```
0 references
public void TresPrimerosBooks(){
    librosCollection
        .Take(3)
        .Select(p => new{Title = p.Title, PCount = p.PageCount});
}
```

Como vemos le estamos asignando nuevos nombres a Title y PageCount, pero también podríamos tomar los nombres originales:

```
public void TresPrimerosBooks(){
    librosCollection
        .Take(3)
        .Select(p => new{p.Title, p.PageCount});
}
```

Pero esto devolverá un objeto dinámico, por ello mejor crearemos un objeto book. Así:

```
public IEnumerable<Book> TresPrimerosBooks(){
    return librosCollection
        .Take(3)
        .Select(p => new Book{Title = p.Title, PageCount = p.PageCount});
}
```

Esto devolverá un objeto book con solo estos campos llenos

```
11 //ImprimirValores(queries.LibrosDeMayor400Skip());
12 ImprimirValores(queries.TresPrimerosBooks());
13
```

| Title                             | N. Page | Publicacion |
|-----------------------------------|---------|-------------|
| Unlocking Android                 | 416     | 1/01/0001   |
| Android in Action, Second Edition | 592     | 1/01/0001   |
| Specification by Example          | 0       | 1/01/0001   |

```
PS C:\platzi\consoleLinq> dotnet run
PS C:\platzi\consoleLinq>
```

Al ejecutarlo podrás comprobar que por ejemplo las fechas de publicación al estar vacía aparece como “1/01/0001” en nuestra función de impresión.



## Operadores de Agregación

Los operadores de agregación permiten realizar cálculos sobre toda la colección y devolver un dato en específico que se calcula de acuerdo a todos los elementos que tiene la colección.

**Operadores LongCount y Count:** Ambos son muy parecidos, de hecho, hacen lo mismo. La diferencia es que **count** soporta 32 bits u **Longcount** soporta 64 bits.

Eso quiere decir que LongCount permite manejar enteros mucho más grandes.

Al final el objetivo es realizar la cuenta de los elementos de la Query, devuelve cuantos elementos hay, podemos utilizarla para consultar cuantos elementos tiene nuestra consulta.

### Reto operador Count

Utilizando el operador Count, retorna el número de libros que tengan entre 200 y 500 páginas.

```
public int LibrosPgCountEntre200y500(){  
    return librosCollection  
        .Where(p=> p.PageCount >= 200 && p.PageCount <= 500)  
        .Count();  
}
```

```
13 Console.WriteLine($"Cantidad de libros con entre 200 y 500 paginas: {queries.LibrosPgCountEntre200y500()}");  
14
```

TERMINAL PROBLEMS 1 OUTPUT DEBUG CONSOLE AZURE powershell

```
PS C:\platzi\consoleLinq> dotnet run  
Cantidad de libros con entre 200 y 500 paginas: 11
```

Si usáramos LongCount tendríamos que utilizar **long** en vez de devolver **int** así:

```
public long CantidadDeLibrosEntre200y500Pag()  
{  
    return librosCollection.Where(p=> p.PageCount>=200 && p.PageCount<=500).LongCount();  
}
```

**ATENCIÓN:** Tanto LongCount como count pueden recibir una condición dentro así lo cual es mas eficiente y nos devuelve más rápido el resultado:

```
public int LibrosPgCountEntre200y500(){  
    return librosCollection  
        .Count(p=> p.PageCount >= 200 && p.PageCount <= 500);  
}
```



## Operadores Min y Max:

Min: Nos devuelve el menor valor, ejemplo:

|  |   |
|--|---|
| <b>Reto operador Min</b>   | <pre>public DateTime FechaPublicacionMenor(){<br/>    return librosCollection.Min(p =&gt; p.PublishedDate);<br/>}</pre> |
| Utilizando el operador Min, retorna la menor fecha de publicación de la lista de libros. |   |

```
14 Console.WriteLine($"Menor Fecha de publicación de la base: {queries.FechaPublicacionMenor()}");  
15  
TERMINAL PROBLEMS 1 OUTPUT DEBUG CONSOLE AZURE  
PS C:\platzi\consoleLinq> dotnet run  
Menor Fecha de publicación de la base: 1/07/1997 00:00:00
```

Max: Nos devuelve el máximo valor, ejemplo:

|   |  |
|---|--|
| <b>Reto operador Max</b>  | <pre>public int MaximaCantidadPaginas(){<br/>    return librosCollection.Max(p =&gt; p.PageCount);<br/>}</pre> |
| Utilizando el operador Max, retorna la cantidad de páginas del libro con mayor número de páginas en la colección. |  |

```
15 Console.WriteLine($"Menor Fecha de publicación de la base: {queries.MaximaCantidadPaginas()}");  
16  
TERMINAL PROBLEMS 1 OUTPUT DEBUG CONSOLE AZURE  
PS C:\platzi\consoleLinq> dotnet run  
Menor Fecha de publicación de la base: 706
```

**Operadores MinBy y MaxBy:** En el apartado anterior vimos como usar min y max para ver el valor máximo o mínimo de un campo de una colección, pero no podemos extraer el registro que tiene este valor, con los operadores que veremos a continuación podremos obtener este registro.

**MinBy:** Este nos da el registro entero de donde halla el valor mínimo. Ejemplo:

|  |   |
|--|---|
| <b>Reto operador MinBy</b>   | <pre>public Book MenorPagMayorCero(){<br/>    return librosCollection<br/>        .Where(p =&gt; p.PageCount &gt; 0)<br/>        .MinBy(p =&gt; p.PageCount);<br/>}</pre> |
| Retorna el libro que tenga la menor cantidad de páginas mayor a 0. |   |

```
16 var libroP = queries.MenorPagMayorCero();  
17 Console.WriteLine($"title:{libroP.Title} n° Pag.: {libroP.PageCount}");  
18  
TERMINAL PROBLEMS 2 OUTPUT DEBUG CONSOLE AZURE  
title:C#: The Ultimate Advanced Guide To Master C# Programming n° Pag.: 123  
PS C:\platzi\consoleLinq>
```

**MaxBy:** Este nos da el registro entero de donde halla el valor máximo. Ejemplo:

#### Reto operador MaxBy

Retorna el libro con la fecha de publicación más reciente.

```
public Book MayorFechaP(){  
    return librosCollection.MaxBy(p => p.PublishedDate);  
}
```

```
18 var libroPMax = queries.MayorFechaP();  
19 Console.WriteLine($"title:{libroPMax.Title}\nDateP: {libroPMax.PublishedDate.ToShortDateString()}\nPag.:  
20  
TERMINAL PROBLEMS 3 OUTPUT DEBUG CONSOLE AZURE  
PS C:\platzi\consoleLinq> dotnet run  
title:C#: The Ultimate Advanced Guide To Master C# Programming  
DateP: 22/04/2022
```

## Operadores Sum y Aggregate

**Sum:** Lo que hace es sumar la propiedad de las celdas que le indiquemos:

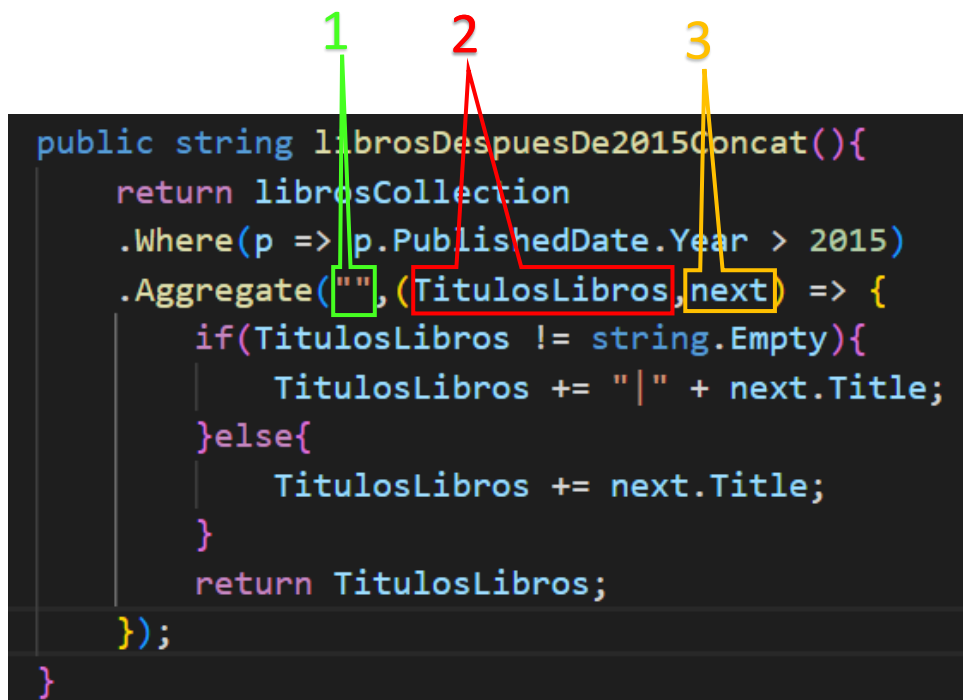
#### Reto operador Sum

Retorna la suma de la cantidad de páginas, de todos los libros que tengan entre 0 y 500.

```
public int SumaLibros0a500Pag(){  
    return librosCollection  
        .Where(p=> p.PageCount >=0 && p.PageCount<= 500)  
        .Sum(p => p.PageCount);  
}
```

```
20 Console.WriteLine($"Suma de paginas de libros 0-500: {queries.SumaLibros0a500Pag()}");  
21  
TERMINAL PROBLEMS 3 OUTPUT DEBUG CONSOLE AZURE  
Suma de paginas de libros 0-500: 4162
```

**Aggregate:** Es un operador muy versátil parecido al operador “reduce” de java script esto permite hacer un total personalizado, utilizar este operador por primera vez puede parecer algo complicado, pero iremos paso a paso.



```
public string librosDespuesDe2015Concat(){
    return librosCollection
        .Where(p => p.PublishedDate.Year > 2015)
        .Aggregate("", (TitulosLibros, next) => {
            if(TitulosLibros != string.Empty){
                TitulosLibros += "|" + next.Title;
            }else{
                TitulosLibros += next.Title;
            }
            return TitulosLibros;
        });
}
```

1. Primero Necesita una semilla, como es un string lo iniciamos con "" si fuese un integer será 0, luego una coma.
2. Luego necesita una variable, que se definirá con el tipo de la semilla en este caso string, luego una coma.
3. Finalmente el “next” siempre representa el objeto que esta recorriendo por eso se puede ver que se le saca el atributo title

```
next.Title;
```

#### Reto operador Aggregate

Retorna el título de los libros que tienen fecha de publicación posterior a 2015.

```
public string librosDespuesDe2015Concat(){
    return librosCollection
        .Where(p => p.PublishedDate.Year > 2015)
        .Aggregate("", (TitulosLibros, next) => {
            if(TitulosLibros != string.Empty){
                TitulosLibros += " - " + next.Title;
            }else{
                TitulosLibros += next.Title;
            }
            return TitulosLibros;
        });
}
```

```
21 Console.WriteLine($"Libros Mayores 2015 Concat: {queries.librosDespuesDe2015Concat()}");
22
TERMINAL PROBLEMS 3 OUTPUT DEBUG CONSOLE AZURE powers
Libros Mayores 2015 Concat: JavaScript: The Definitive Guide - C#: The Ultimate Advanced Guide To Master C# Programming
```

**Operador Average:** Nos permite sacar un promedio de alguna propiedad numérica que tengamos dentro de la colección.

#### Consideraciones:

1. Average devuelve siempre un elemento de tipo Double.
2. La columna no necesariamente tiene que ser numérica, por ejemplo si usamos length estaremos extrayendo el número de caracteres, al ser número el average funciona.

#### Reto operador Average

Utilizando el operador Average, retorna el promedio de caracteres que tienen los títulos de la colección.

```
public double averageCaractTitle(){  
    return librosCollection.Average(p=> p.Title.Length);  
}
```

```
22 Console.WriteLine($"El promedio de numero de caracteres title es: {queries.averageCaractTitle()}");  
23  
24 void TerminaVista() { FormularioDeBusca.Dispose(); }  
TERMINAL PROBLEMS 3 OUTPUT DEBUG CONSOLE AZURE  
El promedio de numero de caracteres title es: 29.24137931034483
```

**Ejm.:** Promedio de numero de paginas de libros con numero de pagina mayor a cero.

```
public double averagePagMayora0(){  
    return librosCollection  
        .Where(p=> p.PageCount > 0 )  
        .Average(p=>p.PageCount);  
}  
23 Console.WriteLine($"El promedio de numero de pag mayores a cero es: {queries.averagePagMayora0()}");  
24  
TERMINAL PROBLEMS 3 OUTPUT DEBUG CONSOLE AZURE  
El promedio de numero de pag mayores a cero es: 461
```

## Operadores de Agrupamiento

Son bastante útiles para agrupar diferentes datos de diferentes tablas y estamos utilizando Entity Framework, podemos hacer un paquete de esos datos de una manera mucho más organizada y retornarla al cliente o a cualquier otra aplicación que lo necesite.

### GroupBy:

#### Reto operador GroupBy

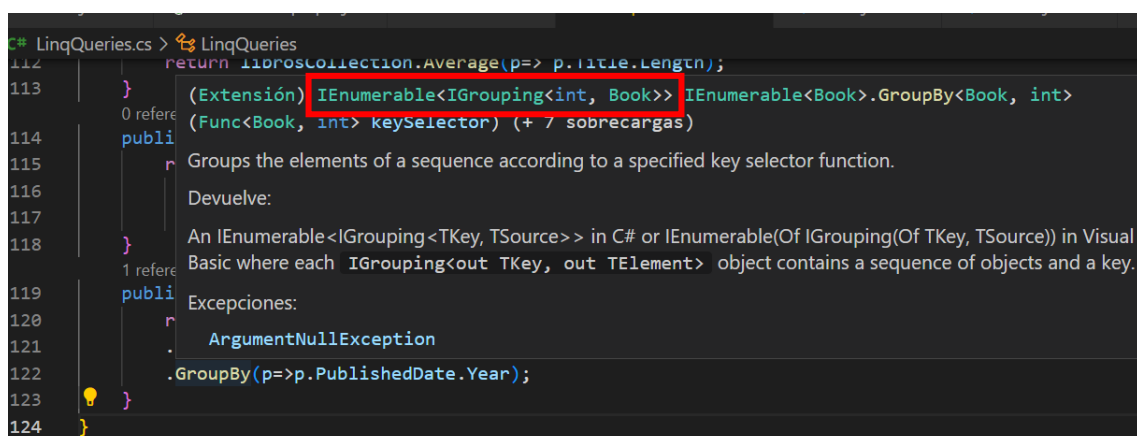
Retorna todos los libros que fueron publicados a partir del 2000, agrupados por año.

```
public IEnumerable<IGrouping<int,Book>> LibrosAgrupadosdespues2000(){
    return librosCollection
        .Where(p=> p.PublishedDate.Year >= 2000)
        .GroupBy(p=>p.PublishedDate.Year);
}
```

Como nos daremos cuenta esta función no devuelve un tipo book, devuelve un tipo distinto por lo cual le generamos una función para poderlo imprimir

```
void ImprimirGruposA(IEnumerable<IGrouping<int,Book>> ListaDeGrupos)
{
    foreach (var ListaDeLibros in ListaDeGrupos)
    {
        Console.WriteLine("");
        Console.WriteLine($"Grupo: {ListaDeLibros.Key}");
        Console.WriteLine("{0,-60} {1,7} {2,11}\n","Title","N. Page","Publicacion");
        foreach (var item in ListaDeLibros)
        {
            Console.WriteLine("{0,-60} {1,7} {2,11}",item.Title,item.PageCount,item.PublishedDate.ToShortDateString());
        }
    }
}
```

En todo momento podemos ver que devuelven las funciones cuando pasamos el mouse por la función



### Resultado

| TERMINAL                              |  |                     |            |
|---------------------------------------|--|---------------------|------------|
| PROBLEMS 3 OUTPUT DEBUG CONSOLE AZURE |  |                     |            |
| Grails in Action                      |  | 520                 | 1/05/2009  |
| Grupo: 2011                           |  |                     |            |
| Title                                 |  | N. Page Publicacion |            |
| Android in Action, Second Edition     |  | 592                 | 14/01/2011 |
| Specification by Example              |  | 0                   | 3/06/2011  |
| OSGi in Depth                         |  | 325                 | 12/12/2011 |
| MongoDB in Action                     |  | 0                   | 12/12/2011 |
| OSGi in Action                        |  | 576                 | 6/04/2011  |

**Operador Lookup:** Este es un operador de agrupación por el nombre puede resultar confuso puesto a que podrías confundirlo con una función de búsqueda de Excel. Consta de varias partes lo veremos con un ejemplo:

En esta primera parte se le dice el elemento por el cual lo querés agrupar, en este caso la posición Cero del title o sea la primera letra

```
public ILookup<char, Book> DiccionarioDeLibrosXLetra(){  
    return librosCollection.ToLookup(p => p.Title[0], p => p);  
}
```

En esta parte el elemento que se quiere devolver del grupo en este caso todo el libro pero podría ser solo el número de páginas ejm p.pageCount

### Reto operador Lookup

Retorna un diccionario usando Lookup que permita consultar los libros de acuerdo a la letra con la que inicia el título del libro

```
public ILookup<char, Book> DiccionarioDeLibrosXLetra(){  
    return librosCollection.ToLookup(p => p.Title[0], p => p);  
}
```

Como nos daremos cuenta esta función no devuelve un tipo book, devuelve un tipo distinto por lo cual le generamos una función para poderlo imprimir

```
void ImprimirDicLetter(ILookup<char, Book> Bloque, char Letra){  
    Console.WriteLine($"Letra: {Letra}");  
    Console.WriteLine("{0,-60} {1,7} {2,11}\n", "Title", "N. Page", "Publicacion");  
    foreach (var item in Bloque[Letra])  
    {  
        Console.WriteLine("{0,-60} {1,7} {2,11}", item.Title, item.PageCount, item.PublishedDate.ToShortDateString());  
    }  
}
```

Le damos la letra

```
var ListaL = queries.DiccionarioDeLibrosXLetra();  
ImprimirDicLetter(ListaL, 'J');
```

Este es resultado:

| Letra: J                             |         |             |
|--------------------------------------|---------|-------------|
| Title                                | N. Page | Publicacion |
| JXPath and Betwixt: Working with XML | 0       | 1/03/2005   |
| JavaScript: The Definitive Guide     | 706     | 6/06/2020   |

En conclusión, es bastante útil pues permite desarrollar nuevas colecciones agrupadas con una colección que ya estemos utilizando, y luego podemos consultar esta luego podemos consultar rápidamente los elementos que son parte de esa colección haciendo uso del agrupamiento.

Al igual que en SQL tenemos en Linq un operador que nos permite interceptar dos colecciones y devolver los elementos que se encuentran en ambas colecciones. Así como hacemos un innerJoin o un join dentro de SQL.

### Reto operador Join

Obtén una colección que tenga todos los libros con más de 500 páginas y otra que contenga los libros publicados después del 2005. Utilizando la cláusula Join, retorna los libros que estén en ambas colecciones.

Veamos el reto por partes, primero creamos una colección que tenga la fecha de publicación mayor a la del 2005 luego creamos otra colección de libros que contengan más de 500 páginas o sea son dos colecciones

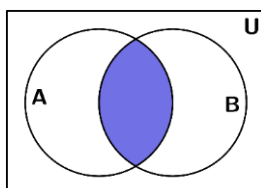
```
var colec1 = librosCollection
    .Where(p=> p.PublishedDate.Year > 2005);
var colec2 = librosCollection
    .Where(p=>p.PageCount > 500);
```

A continuación detallaré cómo funciona el join

```
return colec1.Join(colec2, p=> p.Title, x=> x.Title, (p,x)=>p);
```

1 2 3 4 5 6

1. Primero damos la colección con la que vamos a realizar la operación .Join
2. Dentro del operador la segunda colección con la que debemos realizar la operación.
3. Luego le damos la propiedad con la que se va a relacionar la primera colección con la segunda.
4. ahora le damos la propiedad con la que la segunda colección se relacionará con la primera eso genera una relación. se recomienda utilizar siempre identificadores ID
5. como quinto entre esos paréntesis se pone la representación de las dos variables de la colección en este caso la representación de la primera colección es la letra P y el de la segunda colección es la X.
6. luego en el arrow function ponemos lo que queremos devolver en este caso queremos devolver elementos de la primera colección que también estén en la segunda colección básicamente la intersección de conjuntos.



Gracias, si te ha servido dame +10 y directo al cielo ajajaja