

## TEMA 2. LENGUAJE DE PROGRAMACIÓN EN DISEÑO DE INTERFACES

# LINQ

## Funcionamiento básico

Las expresiones lambda llegaron con la versión 3 del lenguaje C# y han transformado la rutina diaria de los programadores. Una expresión lambda, también llamada una lambda, es una manera de escribir una función directamente dentro de una línea sin tener que definirla explícitamente dentro de la clase. La ventaja de este planteamiento es que, a partir de ese momento, esta función se puede usar como variable y ser ejecutada por otra función, e incluso pasarse directamente a una función. Esta manera nueva de escribir una función (también llamada a veces función flecha) es un método abreviado muy apreciado y ampliamente utilizado. Además, a partir de la versión 3 del framework .NET, Microsoft incluyó un sistema nuevo de gestión de las colecciones: LINQ (Language INtegrated Query).

Es frecuente que los programadores se tengan que enfrentar a una base de datos durante la creación de un proyecto. Muchos conocen el lenguaje SQL, norma extendida para los sistemas de gestión de bases de datos relacionales.

Desde una perspectiva de mutualización de las competencias, Microsoft introdujo LINQ dentro del framework .NET para que los desarrolladores .NET pudieran considerar a todas las colecciones como una base de datos y tuvieran un lenguaje de consulta dentro del código C# similar al proporcionado por SQL.

LINQ se usa en muchas variantes; las más difundidas son las siguientes:

- **LINQ-To-Objects:** permite trabajar con colecciones en memoria (vamos a verlo en este capítulo).
- **LINQ-To-SQL:** permite traducir las consultas LINQ a su equivalente SQL. Este planteamiento se usa especialmente para los sistemas de acceso a los datos, como Entity Framework Core, por ejemplo, que se ocupa de traducir la consulta LINQ en SQL, siempre y cuando el operador sea traducible..

Esta aportación permite hacer consultas en las colecciones de nuestra aplicación con facilidad y obtener de ellas otra colección. Todas las colecciones que se pueden enumerar implementan (entre otras) la interfaz `IEnumerable<T>`, como aquí la definición de la clase `List<T>` del framework:

```

namespace System.Collections.Generic;

//
// Resumen:
// Represents a strongly typed list of objects that can be accessed by index. Provides
// methods to search, sort and manipulate lists.
//
// Parámetros de tipo:
// T:
// The type of elements in the list.
[DefaultMember("Item")]
public class List<T> : ICollection<T>, IEnumerable<T>, IEnumerable, IList<T>, IReadOnlyCollection<T>, IReadOnlyList<T>, ICollection, IList
{
    //
    // Resumen:

```

### Detalle de las interfaces de la clase List<T>

Partiendo de esta sencilla constante, el planteamiento propuesto por el equipo de Microsoft es crear un conjunto de métodos de extensión, es decir, métodos que no están incluidos en el tipo básico, pero que se añaden a él. Entonces, esto significa que cada clase que implementa `IEnumerable<T>` puede usar los métodos disponibles en LINQ.

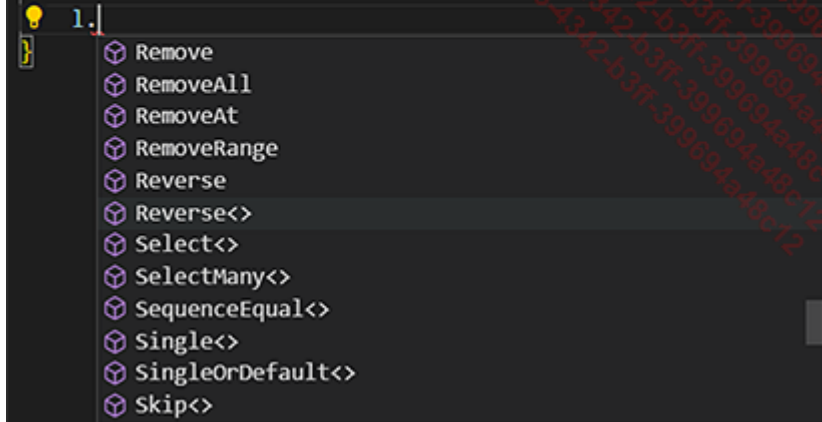
Todos los métodos están guardados en el espacio de nombres `System.Linq`, que hay que importar mediante una instrucción `using` escrita en el encabezado del archivo. Una vez realizada esta operación, se puede observar una lista considerable de métodos adicionales disponibles: //No es necesario en .NET 6, porque es de los namespaces que están implícitos.

```

static void Main(string[] args)
{
    List<int> l = new()
    {
        1, 2, 3, 4, 5, 6, 7, 8, 9
    };

    1.

```



### Lista de los métodos accesibles en una clase de tipo List<T>

Como se puede observar, nos encontramos con una terminología bastante similar a la de las consultas SQL: `SELECT`, `WHERE`, `ORDER BY`, etc., son los operadores.

Pero antes de estudiar los operadores, es necesario comprender un tipo de objeto un poco especial en C#: las variables anónimas.

# Variables anónimas

C# es un lenguaje fuertemente tipado, lo que hace que un objeto de un tipo dado no pueda cambiar de tipo durante su existencia. Sin embargo, con la llegada de LINQ se quiere recuperar un conjunto de valores que tienen una agrupación lógica, pero sin hacer una clase o una estructura definida.

Con la versión 3 de C# se ha introducido el concepto de variable anónima. Se trata de crear un objeto, fuertemente tipado, pero que no pertenece a un tipo conocido ni declarado. Esta flexibilidad permite crear un objeto a medida, que siempre respeta todas las limitaciones. Por ejemplo, si queremos crear un coche sin tener que definir una clase para alojar los datos relacionados, escribimos el siguiente código:

```
var coche = new { Marca = "Peugeot", Caballos = 120, Funciona = true };
```

Como se puede comprobar, no hay ningún tipo después de la instrucción new. La apertura de las llaves permite definir directamente la lista de las propiedades que contendrá este objeto. Esta flexibilidad es muy útil para generar de inmediato un objeto como retorno de la consulta LINQ, a fin de producir un objeto nuevo a partir de datos de una colección de objetos conocidos.

## Principios de los operadores LINQ

De manera nativa, LINQ proporciona una gran cantidad de operadores. Si la lista propuesta no es suficiente para usted, hay un proyecto comunitario que añade muchos otros: <https://github.com/morelinq/MoreLINQ>. Por nuestra parte, nos conformamos con estudiar en este capítulo los que se proporcionan de manera nativa en el framework.

En primer lugar, hay que comprender que LINQ funciona en un orden inverso al de SQL. En efecto, es frecuente que una consulta de SQL simple se formule de esta manera:

```
SELECT ... FROM Table WHERE ... ORDER BY ...
```

Al traducir esta consulta a español, se obtiene la siguiente lógica: en la tabla Table (FROM), donde se aplica una condición (WHERE), ordenando las líneas según un orden (ORDER BY), recupérame las siguientes columnas (SELECT).

Los operadores LINQ toman como parámetro una lambda que define lo que se espera mediante el método. La mayoría de las lambdas para LINQ están formadas de la siguiente manera: la lambda toma como parámetro la instancia actual que corresponde al elemento actual de las colecciones y la usa en el cuerpo de la función. Se puede pasar por una función clásica para obtener el mismo resultado. Así, las dos maneras siguientes de escribir son similares:

```
public void TestLinq()
{
    List<int> l = new()
```

```

{
    1, 2, 3, 4, 5, 6, 7, 8, 9
};
var enteros = l.Where(i => i < 3);
var enteros2 = l.Where(Filtro);
}

public bool Filtro(int i)
{
    return i < 3;
}

```

El segundo caso es menos frecuente porque el nombre de la función que debe ser llamada para el filtro se pasa como parámetro al operador LINQ. Esto solo funciona si la función tiene el número, tipos de parámetros y tipo de devolución correctos; por eso se recomienda usar las funciones flecha.

Algunos operadores trabajan con un valor de la colección y producen un retorno bajo la forma de un booleano, ya que es una prueba que se evalúa para cada uno de los elementos (como el Where, por ejemplo):

```

List<int> l = new()
{
    1, 2, 3, 4, 5, 6, 7, 8, 9
};
var enterosMenoresDeCinco = l.Where(i => i < 5);

```

Esta consulta itera en todos los elementos de la lista de enteros y evalúa, para cada uno de ellos, si responde o no a la prueba de lambda. En el caso actual, lambda es una función que toma un parámetro de tipo int (aquí se ha llamado i) y devuelve un booleano.

Si la prueba devuelve verdadero, el valor está disponible en la colección nueva; en caso contrario, el valor se ignora. Podemos observar que la colección inicial permanece sin cambios porque se produce una colección nueva (aquí, se guarda en la variable llamada enterosMenoresDeCinco).

Otros operadores LINQ no ofrecen devolver un booleano, sino seleccionar el valor que se desea tratar dentro de la colección. Tomemos un ejemplo un poco más complejo para ilustrar esto. Consideremos una lista de coches:

```

public class Coche
{
    public string Marca { get; set; }
    public int Caballos { get; set; }
}

public class Cap5Linq
{
    static void Main(string[] args)
    {

```

```

var coches = new List<Coche>
{
    new Coche{ Marca = "Peugeot", Caballos = 120 },
    new Coche{ Marca = "Renault", Caballos = 110 },
    new Coche{ Marca = "Citroën", Caballos = 90 },
    new Coche{ Marca = "Ferrari", Caballos = 250 },
};
}
}

```

Ya hemos visto cómo filtrar una lista. Por ejemplo, si queremos obtener la lista de los coches que tienen 120 caballos o menos, se usa la siguiente consulta para recuperar una lista nueva de coches filtrados:

```
var cochesPocosCaballos = coches.Where(c => c.Caballos <= 120);
```

Tras esta instrucción, nos encontramos con una colección nueva que solo contiene los coches de la primera colección cuya cantidad de caballos es inferior o igual a 120.

Considerando que Coche es una clase, y por lo tanto un tipo de referencia, nuestras dos colecciones comparten los mismos datos en memoria. Esto significa que, si se usa una u otra colección para modificar un valor de un elemento (por ejemplo, actualizando la cantidad de caballos), las dos colecciones verán que el objeto se modifica porque las dos listan el mismo puntero de memoria y, por lo tanto, el mismo objeto.

Sin embargo, si solo se quiere recuperar la marca de los coches que tienen pocos caballos, estamos obligados a usar otro operador: Select. Gracias a este operador es posible definir el(los) valor(es) que queremos recuperar. Se observa que el operador LINQ Where también devuelve una colección que implementa la interfaz IEnumerable; por eso se pueden encadenar las llamadas:

```
var cochesPocosCaballos = coches.Where(c => c.Caballos
<= 120).Select(c => c.Marca);
```

Al final de la evaluación de esta consulta LINQ, no obtenemos una colección de coches nueva, sino una colección nueva de string, que contiene las marcas de coches que tienen 120 caballos o menos.

Antes de continuar, es necesario comprender algo sobre el tipo IEnumerable<T>. Este tipo, que es el básico cuando se trabaja con LINQ, indica que tenemos un puntero en una colección. La colección a la que apunta quizás no ha sido evaluada (más tarde veremos, en esta sección, operadores LINQ de evaluación). Esto quiere decir que, hasta que no se ha recorrido la colección, no ha sido evaluada. Este concepto es importante porque implica que la consulta LINQ construida sigue siendo completamente virtual mientras no se recorra. Para ilustrar esto, vamos a retomar el método anterior, pero añadiendo una instrucción dentro del filtro:

```
bool seHaRecorrido = false;
var cochesPocosCaballos = coches
```

```
.Where(c => { seHaRecorrido = true; return c.Caballos <= 120; })  
.Select(c => c.Marca);  
Console.WriteLine("¿Colección recorrida? " + seHaRecorrido);
```

Si se ejecuta la aplicación que contiene el código anterior, se puede ver que el booleano `seHaRecorrido` contiene `false`, lo que quiere decir que no se ha ejecutado el código dado a la función `Where`. Sin embargo, si se decide hacer un `foreach` en la variable `cochePocosCaballos`, podemos constatar que el booleano ha pasado a `true` porque `Where` ha sido evaluado durante el recorrido. Esto significa que, mientras no haya decidido explotar la variable `IEnumerable<T>`, es libre de construirla sin riesgo.

Ahora que hemos precisado este concepto, es el momento de ver los operadores que ofrece el framework.

Estos operadores se dividen en tres categorías:

- Los operadores de producción de colección: partiendo de una colección, se obtiene otra colección más específica o genérica.
- Los operadores de selección: partiendo de una colección, se recupera un valor escalar o un elemento específico.
- Los operadores de generación: partiendo de nada, se obtiene una colección.

## 1. Operadores de producción

La mayoría de los operadores se encuentran en esta categoría. Sin embargo, esta categoría se divide en subcategorías. Así, tenemos los siguientes elementos:

- Los operadores de filtrado. De una colección de elementos de tipo `T`, se obtiene una colección nueva de elementos de tipo `T`.
- `Where`: filtra una colección para obtener otra colección que responde a una condición enunciada:

```
var lista = new List<int> { 1, 2, 3, 4, 5 };  
var filtro = lista.Where(i => i >= 2); // Contendrá 2, 3, 4 y 5
```

- `Take`: recupera un número finito de elementos (valor entero) desde el inicio de la colección:

```
var lista = new List<int> { 1, 2, 3, 4, 5 };  
var take = lista.Take(3); // Contendrá 1, 2 y 3
```

- Skip: ignora un número finito de elementos (valor entero) desde el inicio de la colección y recupera el resto:

```
var lista = new List<int> { 1, 2, 3, 4, 5 };
var skip = lista.Skip(2); // Contendrá 3, 4 y 5
```

- TakeWhile: recupera elementos si la evaluación de lambda devuelve verdadero:

```
var lista = new List<int> { 1, 2, 3, 4, 5 };
var takeWhile = lista.TakeWhile(i => i < 3); // Contendrá 1 y 2
```

- SkipWhile: ignora los elementos si la evaluación de lambda devuelve verdadero:

```
var lista = new List<int> { 1, 2, 3, 4, 5 };
var skipWhile = lista.SkipWhile(i => i < 3); // Contendrá 3, 4 y 5
```

- Distinct: devuelve los elementos comparándolos unos con otros para recuperar solo los elementos únicos:

```
var lista = new List<int> { 1, 1, 2, 3, 3, 4, 5, 5, 5 };
var skipWhile = lista.Distinct(); // Contendrá 1, 2, 3, 4 y 5,
donde cada valor está presente una sola vez
```

A excepción de TakeWhile y SkipWhile, los otros operadores tienen un equivalente nativo en SQL.

Para que el operador Distinct pueda funcionar, la comparación debe ser posible.

- Los operadores de proyección. De una colección de elementos de tipo T, se obtiene una colección de elementos nueva de tipo T2, donde T2 es una transformación obtenida a partir de los valores contenidos en T.
- Select: selección de un valor en la colección de elementos:

```
var coches = new List<Coche>
{
    new Coche{ Marca = "Peugeot", Caballos = 120 },
    new Coche{ Marca = "Renault", Caballos = 110 },
    new Coche{ Marca = "Citroën", Caballos = 90 },
    new Coche{ Marca = "Ferrari", Caballos = 250 },
};
var marcas = coches.Select(c => c.Marca); // recupera la
colección de las marcas de coches, que es una colección de string
```

Select es un operador muy flexible, se puede recuperar cualquier objeto cuando se evalúa, y no solo un valor dado.

- **SelectMany**: examen minucioso de colecciones insertadas en una colección:

```
var cochesPorAño = new Dictionary<int, List<Coche>>
{
    [1990] = new List<Coche> { new Coche { Marca = "Peugeot 1990" } },
    [1995] = new List<Coche> { new Coche { Marca = "Renault Clio" } },
    [2000] = new List<Coche> { new Coche { Marca = "Peugeot 307" },
    new Coche { Marca = "Citroën Saxo" } },
    [2007] = new List<Coche> { new Coche { Marca = "Peugeot 308" },
    new Coche { Marca = "Renault Megane" } },
};
var todosLosCoches = cochesPorAño.SelectMany(v => v.Value);
```

- Los operadores de unión. De una colección de elementos de tipo T y de una colección de elementos de tipo T2, se obtiene una colección de elementos de tipo T3 resultante, poniendo en común T y T2 a partir de criterio(s) compartido(s). Observe que el operador también funciona con dos colecciones del mismo tipo.
- **Join**: recuperación de una colección nueva aplicando una lógica de unión entre dos colecciones sobre la base de un criterio común. Este operador parte de una colección para efectuar la unión con otra, después define en primer lugar el selector del criterio común dentro del objeto de la primera colección, luego la sección del criterio común dentro del objeto de la segunda colección y, por último, la producción de un resultado tomando la instancia de la primera colección con la instancia de la segunda colección:

### Con el mismo tipo de datos

```
var lista1 = new List<int> { 1, 2, 3, 4, 5 };
var lista2 = new List<int> { 3, 4, 5, 6, 7 };
var join = lista1.Join(lista2, l1 => l1, l2 => l2, (l1, l2) => l1);
// contendrá 3, 4 y 5
```

### Con un tipo de datos distinto, sobre la base de un criterio definido

```
var coches2 = new List<Coche>
{
    new Coche{ Marca = "Peugeot", Caballos = 120 },
    new Coche{ Marca = "Renault", Caballos = 110 },
    new Coche{ Marca = "Citroën", Caballos = 90 },
};
var camiones = new List<Camion>
{
    new Camion{ Marca = "Mercedes", Caballos= 250},
}
```



```

    new Camion{ Marca = "John Deer", Caballos= 110},
    new Camion{ Marca = "Renault", Caballos= 120},
};
var joinClases = coches2.Join(camiones, coche =>
coche.Caballos, camion => camion.Caballos, (coche, camion) =>
new { Caballos = coche.Caballos, MarcaCoche = coche.Marca,
MarcaCamion = camion.Marca }); // Contendrá { Caballos = 120,
MarcaCoche = Peugeot, MarcaCamion = Renault } y { Caballos = 110,
MarcaCoche = Renault, MarcaCamion = John Deer }

```

- GroupJoin: recuperación de una colección nueva jerarquizada (y no horizontal, como el operador Join), es decir, que se recupera un grupo de datos conectados a una clave, que habrá sido seleccionada durante la llamada de la función. Este operador no existe en SQL:

// Modelos

```

public class Estudiante
{
    public string Nombre { get; set; }
    public int Cursold { get; set; }
}
public class Curso
{
    public int Id { get; set; }
    public string Nombre { get; set; }
}

```

// Consulta LINQ

```

var curso = new List<Curso>
{
    new Curso { Id = 1, Nombre = "Las bases de C#" },
    new Curso { Id = 2, Nombre = "ASP.NET Core" },
    new Curso { Id = 3, Nombre = "EF Core" },
};
var estudiantes = new List<Estudiante>
{
    new Estudiante { Nombre = "Juan Puente", Cursold = 1 },
    new Estudiante { Nombre = "Jaime Duende", Cursold = 1 },
    new Estudiante { Nombre = "Steve Johnson", Cursold = 2 },
    new Estudiante { Nombre = "Scott Hanselman", Cursold = 2 },
    new Estudiante { Nombre = "Gerard Gers", Cursold = 3 }
};
var estudiantesPorCurso = curso.GroupJoin(estudiantes, curso =>
curso.Id, estudiante => estudiante.Cursold, (curso, grupoEstudiantes)
=> new { Estudiantes = grupoEstudiantes, Curso = curso.Nombre });

foreach (var item in estudiantesPorCurso)

```

```

{
    Console.WriteLine("Curso " + item.Curso);
    foreach (var estudiantes in item.Estudiantes)
    {
        System.Console.WriteLine("\t " + estudiante.Nombre);
    }
}

```

- Zip: produce una colección nueva a partir de dos colecciones y de una función de tratamiento de los dos elementos en curso en cada colección. Si una colección es mayor que otra, los elementos adicionales se ignoran:

```

var enteros = new List<int> { 1, 2, 3 };
var enterosEnLetras = new List<string> { "uno", "dos", "tres", "cuatro" };
var resultado = enteros.Zip(enterosEnLetras, (entero, enteroEnLetra) =>
    entero + " = " + enteroEnLetra); //contendrá "1 = uno", "2 = dos",
    "3 = tres"

```

El operador Join es el único que tiene un equivalente SQL, lo que da lugar a la traducción de la consulta en INNER JOIN.

- Los operadores de orden. De una colección de elementos de tipo T, se obtiene una colección de elementos de tipo T que están ordenados de distinta manera.
- OrderBy/OrderByDescending: produce una colección ordenada nueva a partir de una colección dada y de un selector. OrderBy produce una colección ordenada de manera ascendente, mientras que con OrderByDescending la colección producida se ordena de manera descendente:

```

var estudiantes = new List<Estudiante>
{
    new Estudiante { Nombre = "Juan Puente", Cursold = 1 },
    new Estudiante { Nombre = "Jaime Duende", Cursold = 1 },
    new Estudiante { Nombre = "Steve Johnson", Cursold = 2 },
    new Estudiante { Nombre = "Scott Hanselman", Cursold = 2 },
    new Estudiante { Nombre = "Gerard Gers", Cursold = 3 }
};
var estudiantesPorOrdenAlfabetico = estudiantes.OrderBy(e => e.Nombre);

```

```

var enteros = new List<int> { 1, 2, 3 };
var enterosDescendientes = enteros.OrderByDescending(e => e);

```

- ThenBy/ThenByDescending: ejecuta un orden nuevo de una colección ya ordenada, conservando el primer orden activo, pero ordenando según un segundo criterio. ThenBy y ThenByDescending solo pueden llamarse en una colección que ha sido ordenada con un OrderBy o con otro ThenBy:

```
var nombres = new List<string> { "Tom", "Bob", "Jim",
"Jack", "John", "Tim", "Alice" };
var nombresOrdenadosPorTamanoLuegoPorOrdenAlfabetico =
nombres.OrderBy(s => s.Length).ThenBy(s => s);
```

- Reverse: invierte el orden de los elementos dentro de una colección:

```
var enteros = new List<int> { 1, 2, 3 };
enteros.Reverse(); // ahora enteros será 3, 2, 1
```

El encadenamiento OrderBy/ThenBy (con o sin Descending) se gestiona en SQL separando las columnas con comas en la instrucción ORDER BY. Reverse no tiene equivalente.

- Los operadores de agrupación. De una colección de elementos de tipo T, se obtiene una colección nueva de elementos de tipo Group<Key, IEnumerable<T>>.
- GroupBy: agrupa los elementos de una colección según un criterio definido, produciendo una colección nueva que contiene la asociación de una clave con los valores guardados bajo esta clave:

```
public class Estudiante
{
    public string Nombre { get; set; }
    public int MediaDe20 { get; set; }
}
```

```
var estudiantesMedia = new List<Estudiante>
{
    new Estudiante{ Nombre = "Bob", MediaDe20 = 14},
    new Estudiante{ Nombre = "Jim", MediaDe20 = 14},
    new Estudiante{ Nombre = "Jack", MediaDe20 = 12},
    new Estudiante{ Nombre = "John", MediaDe20 = 13},
    new Estudiante{ Nombre = "Tim", MediaDe20 = 13},
};
```

```
var estudianteGruposPorMedia = estudiantesMedia.GroupBy(e =>
e.MediaDe20); //producción de una colección de grupo donde
la media es la clave. Entonces tendremos el grupo con la clave 14,
que contendrá Bob y Jim, por ejemplo
```

Atención: GroupBy es un falso amigo. En efecto, existe un GROUP BY en SQL, pero se refiere a una lógica de teoría de los conjuntos. Por lo tanto, la lógica del GroupBy y la del GROUP BY SQL son bastante diferentes. Así, con frecuencia, la traducción directa de uno a otro es imposible.

- Los operadores de agrupación matemáticos. De una colección de elementos de tipo T y de otra colección de elementos de tipo T, se obtiene una colección nueva de

elementos de tipo T que es el resultado de la aplicación de un operador de la teoría matemática de los conjuntos.

- Concat: produce una colección nueva que es la concatenación de otras dos colecciones:

```
var enteros1 = new List<int> { 1, 2, 3, 4, 5 };  
var enteros2 = new List<int> { 3, 4, 5, 6, 7 };  
var enterosTotales = enteros1.Concat(enteros2); // Se obtendrá  
una colección que contiene 1, 2, 3, 4, 5, 3, 4, 5, 6, 7
```

- Union: produce una colección nueva que es el resultado de la unión de otras dos colecciones. La diferencia principal con Concat es que la colección nueva no contiene duplicados:

```
var enteros1 = new List<int> { 1, 2, 3, 4, 5 };  
var enteros2 = new List<int> { 3, 4, 5, 6, 7 };  
var enterosTotales = enteros1.Union(enteros2); // Se obtendrá  
una colección que contiene 1, 2, 3, 4, 5, 6, 7
```

- Intersect: produce una colección nueva que es la intersección entre dos colecciones, es decir, solo sus elementos comunes:

```
var enteros1 = new List<int> { 1, 2, 3, 4, 5 };  
var enteros2 = new List<int> { 3, 4, 5, 6, 7 };  
var enterosTotales = enteros1.Intersect(enteros2); // Se obtendrá  
una colección que contiene 3, 4, 5
```

- Except: produce una colección nueva que solo contiene los elementos distintos de las dos colecciones:

```
var enteros1 = new List<int> { 1, 2, 3, 4, 5 };  
var enteros2 = new List<int> { 3, 4, 5, 6, 7 };  
var enterosTotales = enteros1.Except(enteros2); // Se obtendrá  
una colección que contiene 1, 2, 6, 7
```

Estos cuatro operadores nacen de la teoría matemática de los conjuntos y, por lo tanto, todos tienen un equivalente en SQL. Concat y Union se traducen mediante la palabra clave UNION (con la adición de ALL para el Concat), mientras que Intersect produce un WHERE ... IN. Al final, Except hace la operación inversa, es decir, WHERE... NOT IN.

- Los operadores de conversión. De una colección de tipo T, se obtiene una colección proyectada y evaluada o transformada. Ninguno de estos operadores tiene un equivalente SQL.

- OfType: toma una colección no genérica para producir una colección genérica que solo contiene elementos del tipo buscado; los otros se ignoran:

```
ArrayList array = new ArrayList();
array.Add(3);
array.Add(4);
array.Add("lolo");
var enterosOfType = array.OfType<int>(); // colección nueva
que contendrá 3 y 4
```

- Cast: toma una colección no genérica para producir una colección genérica y devuelve una excepción en el primer elemento no convertible al tipo buscado. Esta excepción solo se devuelve si se intenta iterar en la colección con el operador Cast, y no con la llamada de la función:

```
ArrayList array = new ArrayList();
array.Add(3);
array.Add(4);
array.Add("lolo");

var exception = array.Cast<int>(); // devolverá una excepción
```

El tipo ArrayList se define dentro del espacio de nombres System.Collection. Se trata de una estructura que permite tener una tabla de objetos con redimensionamiento automático antes de que existiera el tipo List<T>. En la actualidad, no se recomienda usar este tipo, que solo está presente aquí para la explicación y la demostración.

- ToArray: transforma una colección en una tabla, evaluando la consulta LINQ:

```
var enteros = new List<int> { 1, 2, 3, 4, 5 };
var enterosSuperiorATres = enteros.Where(i => i > 3).ToArray();
//aquí tendremos una tabla de 2 enteros que contendrá 4 y 5
```

- ToList: transforma una colección en lista, evaluando la consulta LINQ:

```
var enteros = new List<int> { 1, 2, 3, 4, 5 };
var enterosSuperiorATres = enteros.Where(i => i > 3).ToList();
//aquí tendremos una lista de 2 enteros que contendrá 4 y 5
```

- ToDictionary: transforma una colección en Dictionary, evaluando la consulta LINQ y pidiendo un selector de clave, así como un selector de valor:

```
public class Estudiante
{
```

```

    public string Nombre { get; set; }
    public int MediaDe20 { get; set; }
}

```

```

var estudiantesMedia = new List<Estudiante>
{
    new Estudiante{ Nombre = "Bob", MediaDe20 = 14},
    new Estudiante{ Nombre = "Jim", MediaDe20 = 14},
    new Estudiante{ Nombre = "Jack", MediaDe20 = 12},
    new Estudiante{ Nombre = "John", MediaDe20 = 13},
    new Estudiante{ Nombre = "Tim", MediaDe20 = 13},
};

```

var dicoEstudiantes = estudiantesMedia.**ToDictionary**(e => e.Nombre, e => e.MediaDe20); //producción de un diccionario, donde la clave es un string que contiene el nombre y el valor es el entero que contiene la media

- ToLookup: transforma una colección en Lookup, evaluando la consulta LINQ y pidiendo un selector de clave, así como un selector de valor:

```

public class Estudiante
{
    public string Nombre { get; set; }
    public int MediaDe20 { get; set; }
}

```

```

var estudiantesMedia = new List<Estudiante>
{
    new Estudiante{ Nombre = "Bob", MediaDe20 = 14},
    new Estudiante{ Nombre = "Jim", MediaDe20 = 14},
    new Estudiante{ Nombre = "Jack", MediaDe20 = 12},
    new Estudiante{ Nombre = "John", MediaDe20 = 13},
    new Estudiante{ Nombre = "Tim", MediaDe20 = 13},
};

```

var dicoEstudiantes = estudiantesMedia.**ToLookup**(e => e.Nombre, e => e.MediaDe20); //producción de un lookup, donde la clave es un string que contiene el nombre, y el valor es el entero que contiene la media

La clase Lookup expone la misma lógica que la clase Dictionary, es decir, un vínculo entre una clave y un valor. La distinción procede del hecho de que la clase Lookup es inmutable, es decir, que ya no se puede modificar la colección una vez creada.

- AsEnumerable: transforma una consulta LINQ en IEnumerable:

```
var enteros = new List<int> {1, 2, 3, 4, 5 };
var enterosDecrecientes = enteros.OrderByDescending(e => e)
.AsEnumerable(); // "retransformación" de un IOrderedEnumerable
en IEnumerable gracias al método AsEnumerable
```

- AsQueryable: transforma una consulta LINQ en IQueryable:

```
var enteros = new List<int> {1, 2, 3, 4, 5 };
var enterosQueryable = enteros.AsQueryable();
```

El tipo IQueryable no se ha visto hasta ahora. Este último es más completo que el tipo IEnumerable porque permite guardar los operadores de la consulta LINQ y sus valores bajo la forma de un árbol (dentro de la clase Expression). Este dato permite a diversas herramientas trabajar con la expresión (por ejemplo: ORM Entity Framework, que generará SQL a partir de la consulta LINQ). Es un tema avanzado que no trataremos en este libro.

## 2. Operadores de selección

Los operadores de selección permiten extraer un elemento particular de una colección. Esta colección puede haber sido tratada previamente por uno o varios operadores de producción. Se distinguen varios tipos de operadores:

- Los operadores de elementos, que permiten extraer un elemento exacto.
- First/FirstOrDefault: recupera el primer elemento de una colección. La versión FirstOrDefault devuelve el valor predeterminado en caso de que no se haya podido recuperar el primer elemento, mientras que First devuelve una excepción si no se puede recuperar el primer elemento. Este operador también puede tomar directamente un predicado:

```
var enteros = new List<int> { 1, 2, 3, 4, 5 };
var primero = enteros.First(); // Aquí, se guardará el valor 1
var primeroException = enteros.First(i => i > 5); // devolverá
una excepción
var primeroPredeterminado = enteros.FirstOrDefault(i => i > 5);
// devolverá el valor predeterminado del tipo int, por lo tanto 0
```

- Last/LastOrDefault: recupera el último elemento de una colección. El funcionamiento de la versión OrDefault es idéntico al de FirstOrDefault:

```
var enteros = new List<int> { 1, 2, 3, 4, 5 };
var primero = enteros.Last(); // Aquí, se guardará el valor 5
var primeroException = enteros.Last(i => i > 5); // devolverá
una excepción
var primeroPredeterminado = enteros.LastOrDefault(i => i > 5);
```

// devolverá el valor predeterminado del tipo int, por lo tanto 0

First y Last se evalúan en SQL como si fueran un SELECT TOP 1 con una cláusula ORDER BY. Esta última es ASC en el caso de First o DESC en el caso de Last.

- Single/SingleOrDefault: recupera el único elemento de una colección. En el caso de que haya varios elementos que se puedan recuperar, Single y SingleOrDefault devolverán ambos una excepción. En el caso de que no se pueda recuperar ningún elemento, el funcionamiento de las dos versiones será similar a First y FirstOrDefault:

```
var enteros = new List<int> { 1, 2, 3, 4, 5 };
var primero = enteros.Single(); // Devolverá una excepción
var tres = enteros.Where(i => i == 3).Single(); //Aquí tendremos 3
var primeroExcepcion = enteros.Where(i => i > 5).Single();
// devolverá una excepción
var primeroPredeterminado = enteros.Where(i => i > 5).SingleOrDefault();
// devolverá el valor predeterminado del tipo int, por lo tanto 0
```

- ElementAt/ElementAtOrDefault: recupera un elemento en una posición específica (mismo funcionamiento que el uso de un índice, con una base en 0). La versión OrDefault funciona de manera similar a lo que hemos visto antes:

```
var enteros = new List<int> { 1, 2, 3, 4, 5 };
var primero = enteros.ElementAt(1); // aquí tendremos el valor 2
var primeraExcepcion = enteros.Where(i => i > 5).ElementAt(0);
// devolverá una excepción
var primeroPredeterminado = enteros.Where(i => i > 5). ElementAt(0);
// devolverá el valor predeterminado del tipo int, por lo tanto 0
```

Single y ElementAt no tienen equivalente en SQL.

- DefaultIfEmpty: devuelve el valor predeterminado si la colección está vacía. Si la colección no está vacía, simplemente se recupera tal cual:

```
var enteros = new List<int> { 1, 2, 3, 4, 5 };
var enterosVacios = new List<int>();
var enterosIfEmpty = enteros.DefaultIfEmpty(); // aquí, tendremos
la colección inicial
var enterosEmpty = enterosVacios.DefaultIfEmpty(); // aquí tendremos
una colección solo con el valor 0 (equivalente de new
List<int> { 0 })
```

DefaultIfEmpty se puede traducir a SQL gracias al concepto de unión con un OUTER JOIN.



- Los operadores de agrupación, que ejecutan una operación de agrupación de información para producir un valor escalar.
- Count/LongCount: recuperación del valor de la cantidad de elementos dentro de una colección contándolos explícitamente. LongCount se debe usar en una colección grande (superior a dos mil millones de elementos) porque el valor recuperado es de tipo long, y no int. También se puede pasar una lambda a Count para contar solo los elementos que le responden:

```
var enteros = new List<int> { 1, 2, 3, 4, 5 };
var count = enteros.Count(); // aquí, tendremos el valor 5
var count = enteros.Count(i => i > 2); // aquí, tendremos el valor 3
porque hay 3 elementos que son superiores al valor 2
```

El uso del operador Count sin lambda permite contar la cantidad de elementos, pero esta manera de proceder es más lenta y costosa que acceder directamente a la propiedad que tiene el tamaño actual. Así, si la colección es una instancia de la clase List<T>, se recomienda usar la propiedad Count en lugar del operador LINQ. De la misma manera, si se trata de una tabla, es preferible acceder a la propiedad Length.

- Min/Max: recuperación del valor mínimo o máximo dentro de una colección. Para que la comparación sea posible, el tipo debe ser comparable, es decir, implementar la interfaz IComparable<T>. También es posible pasar una lambda para obtener un valor calculado o para seleccionar el dato afectado en caso de uso de un tipo complejo:

```
var enteros = new List<int> { 1, 2, 3, 4, 5 };
var max = enteros.Max(); // aquí, tendremos el valor 5
var min = enteros.Min(); // aquí, tendremos el valor 1
var maxModulo2 = enteros.Max(i => i % 2); // aquí, tendremos el valor 1,
porque el max es 5, y modulo 2, el resultado es 1
```

```
public class Estudiante
{
    public string Nombre { get; set; }
    public int MediaDe20 { get; set; }
}
```

```
var estudiantesMedia = new List<Estudiante>
{
    new Estudiante{ Nombre = "Bob", MediaDe20 = 14},
    new Estudiante{ Nombre = "Jim", MediaDe20 = 15},
    new Estudiante{ Nombre = "Jack", MediaDe20 = 12},
    new Estudiante{ Nombre = "John", MediaDe20 = 13},
    new Estudiante{ Nombre = "Tim", MediaDe20 = 13},
};
```

```
var mediaMax = estudiantesMedia.Max(e => e.MediaDe20);  
// aquí, tendremos 15, que es la media más alta
```

- Sum: recuperación de la suma de todos los elementos si estos soportan la suma o de un dato de un tipo complejo. El operador es bastante restrictivo y solo soporta tipos numéricos:

```
var enteros = new List<int> { 1, 2, 3, 4, 5 };  
var sum = enteros.Sum(); // aquí, tendremos 15
```

```
public class Estudiante  
{  
    public string Nombre { get; set; }  
    public int MediaDe20 { get; set; }  
}
```

```
var estudiantesMedia = new List<Estudiante>  
{  
    new Estudiante{ Nombre = "Bob", MediaDe20 = 14},  
    new Estudiante{ Nombre = "Jim", MediaDe20 = 15},  
    new Estudiante{ Nombre = "Jack", MediaDe20 = 12},  
    new Estudiante{ Nombre = "John", MediaDe20 = 13},  
    new Estudiante{ Nombre = "Tim", MediaDe20 = 13},  
};
```

```
var mediaMax = estudiantesMedia.Sum(e => e.MediaDe20);  
// aquí, tendremos 67, la suma de todas las medias
```

- Average: recuperación de la media de todos los elementos o de un dato de un tipo complejo. Por definición, una media es un número real; según el dato seleccionado, el operador devuelve un resultado de tipo decimal, float o double (este último es el más habitual):

```
var enteros = new List<int> { 1, 2, 3, 4, 5 };  
var avg = enteros.Average(); // aquí, tendremos 3
```

```
public class Estudiante  
{  
    public string Nom { get; set; }  
    public int MediaDe20 { get; set; }  
}
```

```
var estudiantesMedia = new List<Estudiante>  
{  
    new Estudiante{ Nombre = "Bob", MediaDe20 = 14},
```

```

new Estudiante{ Nombre = "Jim", MediaDe20 = 15},
new Estudiante{ Nombre = "Jack", MediaDe20 = 12},
new Estudiante{ Nombre = "John", MediaDe20 = 13},
new Estudiante{ Nombre = "Tim", MediaDe20 = 13},
};

```

```

var mediaDeLasMedias = estudiantesMedia.Average(e =>
e.MediaDe20); // aquí, tendremos 13.2

```

Todos los operadores presentados con anterioridad tienen un equivalente en SQL.

- Los operadores de cuantificación, que evalúan un método para obtener un valor booleano basado en el resultado de esta evaluación.
- **Contains**: define si un elemento particular está contenido dentro de una colección de destino:

```

var enteros = new List<int> { 1, 2, 3, 4, 5 };
var estaAlli = enteros.Contains(3); // true
var noEstaAlli = enteros.Contains(6); // false

```

- **Any**: define si al menos un elemento que responde a la condición expresada bajo la forma de lambda está contenido dentro de una colección de destino:

```

var enteros = new List<int> { 1, 2, 3, 4, 5 };
var estaAlli = enteros.Any(i => i > 3); // true
var noEstaAlli = enteros.Any(i => i >= 6); // false

```

**Any** y **Contains**, ambas se traducen a SQL con **WHERE ... IN (...)**.

- **All**: define si todos los elementos de la colección responden a la condición expresada bajo la forma de lambda:

```

var enteros = new List<int> { 1, 2, 3, 4, 5 };
var todos = enteros.All(i => i >= 1); // true
var noTodos = enteros.All(i => i >= 2); // false

```

- **SequenceEqual**: define si todos los elementos de una colección son idénticos a los de otra colección. El orden de los elementos es primordial dentro de la comparación:

```

var enteros = new List<int> { 1, 2, 3, 4, 5 };
var enterosEq = new List<int> { 1, 2, 3, 4, 5 };
var enterosNotEq = new List<int> { 1, 3, 4, 2, 5 };

```

```

var equal = enteros1.SequenceEqual(enterosEq); // true

```

```
var notEqual = enteros1.SequenceEqual(enterosNotEq); // false
```

### 3. Operadores de generación

Usados con menos frecuencia, estos operadores permiten generar un valor o un conjunto de valores a partir de una sencilla llamada de método:

- **Empty**: producción de una colección nueva vacía:

```
var empty = Enumerable.Empty<string>(); // colección vacía string
```

- **Range**: producción de una colección nueva que contiene una cantidad de valores definida a partir de un valor inicial:

```
var enteros = Enumerable.Range(1, 5); // Contiene 1, 2, 3, 4 y 5
```

- **Repeat**: producción de una colección nueva que repite un valor fijo una cantidad de veces definida:

```
var soloUnos = Enumerable.Repeat(1, 5); // contiene 1, 1, 1, 1, 1
```

## Expresión de consulta LINQ

Hasta ahora, hemos hablado de LINQ desde el punto de vista de los métodos de extensión en la interfaz `IEnumerable<T>`. Sin embargo, LINQ ofrece otra sintaxis, llamada expresión de consulta (o query expression en inglés). Esta última se acerca a un formalismo que se parece más a lo que se puede encontrar en SQL, incluso si el orden lógico está invertido.

En efecto, cuando se usa este enfoque, hay que empezar por describir el nombre de la variable con la que se quiere trabajar dentro de una colección dada, usando el formalismo `from ... in ...`

Después de esta extracción, se pueden añadir uno o varios operadores:

- **where**.
- **orderby/orderby ... descending**. De forma predeterminada, el operador `order by` funciona de manera ascendente sin que sea necesario especificarlo. Sin embargo, para que sea explícito se puede añadir la palabra clave `ascending`.
- **thenby/thenby ... descending**. De forma predeterminada, `then by` funciona de manera ascendente sin que sea necesario especificarlo. Sin embargo, para que sea explícito se puede añadir la palabra clave `ascending`.

- group ... by ....
- join.

La consulta termina con un select.

Por ejemplo:

```
var enteros = new List<int> { 1, 2, 3, 4, 5 };
var clasificacion = from i in enteros where i > 2 orderby
i descending select
i.ToString(); // tendremos una colección de cadenas, que contiene
los enteros superiores a 2, clasificados en orden decreciente
```

Como se puede comprobar, esta manera de escribir una consulta LINQ se acerca al SQL, a excepción de la inversión de la lógica de selección (el operador select está el último en LINQ, pero llega el primero en SQL).

## 1. La palabra clave into

Esta palabra clave (que solo existe en las expresiones de consultas LINQ) permite hacer una agrupación nueva de datos dentro de una variable intermedia nueva.

Esto permite crear una colección nueva a partir de los datos de la variable inicial, que de hecho se convierte en inaccesible. Escribiendo la consulta anterior con la palabra clave into, se obtiene:

```
var enteros = new List<int> { 1, 2, 3, 4, 5 };
var clasificacion = from i in enteros where i > 2 select i into
enteroSupADos
orderby enteroSupADos descending select enteroSupADos.ToString();
```

La nueva variable intermedia, enteroSupADos, ya ha sido prefiltrada con lo que se ha hecho antes, gracias a la palabra clave into. Esto también significa que la variable inicial, i, se convierte en inaccesible:

```
var clasificacion = from i in enteros where i > 2 select i into
enteroSupADos
orderby enteroSupADos descending select i.ToString(); // ilegal
usar i aquí, el into la ha hecho "desaparecer"
```

El operador join también usa la palabra clave into, que de hecho se convierte en un GroupJoin:

```
var curso = new List<Curso>
{
    new Curso { Id = 1, Nombre = "Las bases de C#" },
    new Curso { Id = 2, Nombre = "ASP.NET Core" },
```

```

    new Curso { Id = 3, Nombre = "EF Core"},
};
var estudiantes = new List<Estudiante>
{
    new Estudiante { Nombre = "Juan Puente", Cursold = 1 },
    new Estudiante { Nombre = "Jaime Duende", Cursold = 1 },
    new Estudiante { Nombre = "Steve Johnson", Cursold = 2 },
    new Estudiante { Nombre = "Scott Hanselman", Cursold = 2 },
    new Estudiante { Nombre = "Gerard Gers", Cursold = 3 }
};
var estudiantesPorCurso2 = from c in curso
    join e in estudiantes on c.Id equals e.Cursold
    into estudiantesCurso
    select new { Curso = c, Estudiantes =
estudiantesCurso };
foreach (var item in estudiantesPorCurso2)
{
    Console.WriteLine("Curso " + item.Curso);
    foreach (var estudiante in item.Estudiantes)
    {
        System.Console.WriteLine("\t " + estudiante.Nombre);
    }
}

```

También es útil para el operador group ... by para obtener una colección nueva:

```

var estudiantesMedia = new List<Estudiante>
{
    new Estudiante{ Nombre = "Bob", MediaDe20 = 14},
    new Estudiante{ Nombre = "Jim", MediaDe20 = 14},
    new Estudiante{ Nombre = "Jack", MediaDe20 = 12},
    new Estudiante{ Nombre = "John", MediaDe20 = 13},
    new Estudiante{ Nombre = "Tim", MediaDe20 = 13},
};
var estudianteGruposPorMedia = from e in etudiantesMedia group
e.Nombre by e.MediaDe20 into nombresPorMedia order by
nombresPorMedia.Key select nombresPorMedia;

```

La variable nombresPorMedia representa el mismo tipo de objeto que el que se puede obtener usando el método de extensión LINQ GroupBy, a saber: un IGrouping<int, IEnumerable<string>>, que pone en correlación la media (la clave, un int) y la lista de los nombres de los estudiantes que tienen esta media.

## 2. La palabra clave let

La sintaxis de expresión de consulta permite usar la palabra clave let, que da la posibilidad de asignar una variable nueva durante la creación de la consulta. La primera variable que se ha creado es la declarada durante el from, pero la palabra clave let permite definir una

variable adicional para las necesidades de la consulta. Una vez creada la variable, se puede usar dentro del conjunto de la consulta para perfeccionar el resultado:

```
var listaEnteros = new List<int> { 1, 2, 3, 4, 5, 6, 7, 8, 9 };  
var enterosPares = from e in listaEnteros  
    let pares = e % 2  
    where pares == 0  
    select e;
```

La ventaja de este enfoque es que se puede definir una expresión o una prueba bajo la forma de variable reutilizable durante toda la consulta sin tener que duplicarla. La cantidad de declaraciones `let` no está limitada y se puede usar a cualquier nivel de la consulta después de su declaración.