Escuelita Crombie 3er Edición

Documentacion
Fuente

¿Qué es LINQ?

Dicho de manera sencilla, LINQ (**L**anguage **In**tegrated **Q**uery) es un conjunto de extensiones integradas en el lenguaje C#, que nos permite trabajar de manera cómoda y rápida con colecciones de datos, como si de una base de datos se tratase. Es decir, podemos llevar a cabo inserciones, selecciones y borrados, así como operaciones sobre sus elementos.

Todas estas operaciones las vamos a conseguir muy fácilmente gracias a los **métodos de extensión** para colecciones que nos ofrece el espacio de nombres "System.Linq" y a las **expresiones lambda**. Sin ir más lejos, los dos ejemplos anteriores se podrían hacer así:

```
var valores = new List<int> {1,2,3,4,5,6,7,8,9};
var suma = valores.Sum();
var pares = valores.Where(x => x % 2 == 0).ToList();
```

Como se puede comprobar, la lectura es mucho más clara, por lo que ganamos en mantenibilidad del código.

En el ejemplo anterior, vemos que LINQ puede devolvernos valores de operaciones, o devolvernos colecciones.

Seguramente te hayas fijado en el `ToList()` del segundo caso. Esto es porque LINQ siempre nos va a devolver un objeto de tipo `IEnumerable<T>`, el cual debemos iterar. **Hasta que no lo iteremos, la consulta no se ha ejecutado todavía, y solo tenemos una expresión sobre una colección**, por eso invocamos `ToList()` para forzar la ejecución de la consulta.

Select

Nos va a permitir hacer una selección sobre la colección de datos, ya sea seleccionándolos todos, solo una parte o transformándolos:

```
var nombresAlumnos = alumnos.Select(x =>
x.Nombre).ToList();
```

Where

Nos permite seleccionar una colección a partir de otra con los objetos que cumplan las condiciones especificadas:

```
var alumnosAprobados = alumnos.Where(x => x.Nota >=
5).ToList();
```

First/Last

Esta extensión nos va a permitir obtener respectivamente el primer y el último objeto de la colección. Esto es especialmente útil si la colección está ordenada.

```
var primero = alumnos.First(); var ultimo =
alumnos.Last();
```

OrderBy

Gracias a este método, vamos a poder ordenar la colección en base a un criterio de ordenación que le indicamos mediante una expresión lambda. Análogamente, también existe `OrderByDescending`, el cual va a ordenar la colección de manera inversa según el criterio:

```
var ordenadoMenorAMayor = alumnos.OrderBy(x =>
x.Nota).ToList(); var ordenadoMayorAMenos =
alumnos.OrderByDescending(x => x.Nota).ToList();
```

GorupBy

LINQ permite agrupar elementos de una colección según una clave específica, creando grupos donde cada uno contiene los elementos que comparten esa clave. Esto es útil para realizar operaciones de agregación como contar, promediar o sumar sobre cada grupo. Por ejemplo, puedes agrupar una lista de personas por ciudad y contar cuántas personas viven en cada ciudad, calculando también valores como edad promedio o total en cada grupo.

```
var agrupadoPorCiudad = personas
    .GroupBy(p => p.Ciudad)

    .Select(g => new

    {

        Ciudad = g.Key,

       CantidadPersonas = g.Count()
});
```

Respuesta esperada

Ciudad: Madrid, Cantidad de Personas: 2

Ciudad: Barcelona, Cantidad de Personas: 2

Ciudad: Valencia, Cantidad de Personas: 1

GorupBy

LINQ permite agrupar elementos de una colección según una clave específica, creando grupos donde cada uno contiene los elementos que comparten esa clave. Esto es útil para realizar operaciones de agregación como contar, promediar o sumar sobre cada grupo. Por ejemplo, puedes agrupar una lista de personas por ciudad y contar cuántas personas viven en cada ciudad, calculando también valores como edad promedio o total en cada grupo.

```
var agrupadoPorCiudad = personas
    .GroupBy(p => p.Ciudad)

    .Select(g => new

    {

        Ciudad = g.Key,

        CantidadPersonas = g.Count()
});
```

Respuesta esperada

Ciudad: Madrid, Cantidad de Personas: 2

Ciudad: Barcelona, Cantidad de Personas: 2

Ciudad: Valencia, Cantidad de Personas: 1

Sum

Como hemos visto más arriba, nos va a permitir sumar la colección:

```
var sumaNotas = alumnos.Sum(x => x.Nota);
```

Max/Min

Gracias a esta extensión, vamos a poder obtener los valores máximo y mínimo de la colección:

```
var notaMaxima = alumnos.Max(x => x.Nota); var notaMinima = alumnos.Min(x => x.Nota);
```

Average

Este método nos va a devolver la media aritmética de los valores (numéricos) de los elementos que le indiquemos de la colección:

```
var media = alumnos.Average(x => x.Nota);
```

All/Any

Con este último operador, vamos a poder comprobar si todos o alguno de los valores de la colección cumplen el criterio que le indiquemos:

```
var todosAprobados = alumnos.All(x => x.Nota >= 5); var algunAprobado = alumnos.Any(x => x.Nota >= 5);
```

Join

Permite combinar dos colecciones en función de una clave común, similar a un *INNER JOIN* en SQL. Útil para relacionar datos de distintas listas. Se especifica la clave en ambas colecciones y el resultado proyectado:

```
var resultado = personas
    .Join(ordenes,
          persona => persona.Id,
          orden => orden.PersonaId,
          (persona, orden) => new
          {
              PersonaNombre = persona.Nombre,
              OrdenFecha = orden.Fecha,
              OrdenTotal = orden.Total
          });
```

Sintaxis integrada

Aunque en los ejemplos anteriores hemos visto el uso directo de los métodos de extensión, otra de las grandes ventajas que tiene LINQ es que permite crear expresiones directamente en el código, de manera similar a si escribiésemos SQL directamente en C#. Por ejemplo:

```
var resultado = from alumno in alumnos where alumno.Nota
>= 5 orderby alumno.Nota select alumno;
```

nos devolverá la lista de alumnos que tienen una nota superior a o igual a 5, ordenados por nota ascendentemente.

Ventajas y desventajas

Ahora que hemos visto un poco por dónde pisamos, es hora de que hablemos sobre las ventajas y desventajas que nos puede aportar utilizar LINQ en vez de iterar las colecciones.

La principal y única **desventaja** que tiene, es que es un poco más lenta que si utilizásemos bucles `for` o `foreach` para iterar la colección y hacer la operación. Por supuesto esto no es apreciable en prácticamente ninguna situación convencional, pero en entornos donde cada milisegundo cuenta, debes conocer que tiene un impacto.

Por otro lado, las **ventajas** que nos aporta LINQ son principalmente que el código es más legible, ya que utiliza una sintaxis muy declarativa de lo que está haciendo, y sobre todo, nos ofrece una manera unificada de acceder a datos, sean el tipo que sean, y tengan el origen que tengan. Por ejemplo, podemos utilizar LINQ para trabajar con bases de datos, con XML, con Excel, con objetos en memoria, ¡y hasta con Twitter!

Resumiendo

Pese a que en esta entrada solo hemos hecho una pequeña introducción con un resumen reducido de las extensiones más frecuentes que nos aporta LINQ (créeme que muy pequeño... te recomiendo mirar el espacio de nombres y ver todas sus opciones), **es una herramienta muy potente**. Tanto, que otros lenguajes la han implementado también.

Si bien es cierto que existe una merma de rendimiento respecto a iterar el bucle directamente, el rendimiento perdido en el 99,99% de los casos se compensa con el beneficio que aporta tener un código claro, legible y mantenible.

Ejemplo de encadenada

Filtrar, ordenar y seleccionar datos específicos

```
var resultado = personas
    .Where(p => p.Edad >= 18 && p.Edad <= 30)
    .OrderBy(p => p.Apellido)
    .Select(p => new { p.Nombre, p.Edad });
```