

# Tema 11.4

Descargar estos apuntes en [pdf](#) o [html](#)

## Índice

- ▼ Introducción a la Programación Funcional
  - ▼ Cálculo Lambda
    - Definición de Expresión Lambda
    - Expresiones Lambda en C#
  - ▼ Clausuras en CSharp
    - Variables Externas Capturadas
  - ▼ Patrón Map-Filter-Fold
    - Map (o Select)
    - Filter (o Where)
    - Fold (o Aggregate)
    - Combinando las tres operaciones
  - ▼ Otras operaciones funcionales declarativas en C#
    - OrderBy/OrderByDescending
    - Distinct
    - ▼ GroupBy
      - Resultado del GroupBy en un nuevo tipo de datos
      - Resultado del GrupBy en un diccionario
    - Zip
    - FlatMap o Aplanado

# Introducción a la Programación Funcional

La **programación funcional** es un **paradigma de programación declarativa** basado en el uso de **funciones matemáticas**, en contraste con la programación **imperativa**, que enfatiza los cambios de estado mediante la mutación de variables.

Existen lenguaje de programación funcionales puros como por ejemplo **Haskell** y **Clojure**. Sin embargo, el paradigma funcional ha ido cogiendo cada vez más peso en la programación moderna. Tanto es así, que lenguajes imperativos y orientados a objetos tradicionales como **C++**, **Java**, **C#** o **PHP** han incluido características de la programación funcional en su sintáxis o potenciando las que ya tenían lenguajes como **JavaScript**, **Python**. Además, los lenguajes multi-paradigma de nueva creación como **Scala**, **Go**, **F#** o **Kotlin** (este último en menor medida) se han diseñado para que tenga más peso la programación funcional que la orientada a objetos e imperativa.

Simplificando y a grandes rasgos podemos decir que **se basa en los siguientes pilares matemáticos**:

- **Cálculo Lambda**.
- **Clausuras**.
- **Teoría de categorías** y más en concreto el concepto de **functor** y su aplicación a las ciencias de la computación a través de:
  - **Mónadas**.
  - **Covarianza y Contravarianza**.
- **Recursión o Recursividad** (Formaría parte del Cálculo Lambda).

Cómo se puede observar, su bases son bastante amplias y llevaría bastante ahondar todos estos conceptos. Por esta razón, este tema pretende ser meramente introductorio y por tanto en él solo vamos a abordar algunos conceptos a través del lenguaje C#. Estos conceptos serán extrapolables a otros lenguajes ya que como hemos comentados todos utilizan, en mayor o medida, dicho paradigma.

## Cálculo Lambda

Es uno de los pilares de la programación funcional. Esté fue formulado por el matemático-lógico norteamericano **Alonzo Church** y consiste en '*...un sistema formal diseñado para investigar la definición de función, la noción de aplicación de funciones y la recursión...*'.

Se basa en una definición de función alternativa a la tradicional que dentro del contexto de las ciencias de computación denominaremos: '**Expresión Lambda**'.

## Definición de Expresión Lambda

Para entender lo que es, vamos ver el concepto de función dependiendo de ámbito de conocimiento...

1. En **matemática tradicional** las funciones se representan con un nombre  $f$  o  $g$

Definición	Explicación
$f(x) = x + 2$	Un parámetro.
$pow(x, y) = x^y$	Dos parámetros.

2. En la **teoría matemática del cálculo lambda** ( $\lambda$ -calculus) esta misma función **se expresaría sin nombre**, solo los parámetros que entran y la expresión que la representa.

Definición	Explicación
$\lambda x. x + 2$	Un parámetro (definición) y equivale a $f(x) = x + 2$ .
$(\lambda x. x + 2)10 = 12$	Un parámetro (aplicación de una valor) y equivale a $f(10) = 10 + 2 = 12$ .
$\lambda(x, y). x^y$	Dos parámetros.

3. 🖐 En **ciencias de computación** se expresará a través de una **función anónima o expresión lambda**.

Definición	Explicación
$x \rightarrow x + 2$	Un parámetro (definición) y equivale a $f(x) = x + 2$ .
$(x \rightarrow x + 2)(10) = 12$	Un parámetro (aplicación de un valor) y equivale a $f(10) = 10 + 2 = 12$ .
$(x, y) \rightarrow x^y$	Dos parámetros.

Esta última, forma de definir funciones de forma anónima y como expresiones, es la que usaremos el la gran mayoría de lenguajes de programación y aunque vamos a ver el caso concreto de C#, la forma de hacelos será muy similar en otros lenguajes. Por ejemplo, si ya las hemos usado en JavaScript, su sintaxis nos será muy familiar.

# Expresiones Lambda en C#

Una **λ-expresión** en C# es un **método anónimo** (sin nombre) que se ha de declarar/definir y asignar inmediatamente, sobre una instancia de un **delegado**.

Las λ-expresión puede ser usadas en alguno estos escenarios en C#:

1. Definir cuerpos de expresión dentro de una clase (**Esto ya lo hemos usado**).

✦ **Nota:** Por ejemplo, un cuerpo de expresión es una expresión lambda.

2. Pasar como argumento a otra función.
3. Como tipo de retorno de una función, esto es, *'Una función que retorna otra función'*.
4. Asignación a instancias de delegados parametrizados.

La representación general de una expresión lambda sigue el siguiente formato general...

**(parámetros) => sentencia o {bloque de sentencias con return si función}**

Vamos a describir que es cada cosa:

1. **(parámetros)** : Lista de parámetros separada por comas.

```
(x, y) => Math.Pow(x, y);
```

Se debe dejar los paréntesis vacíos **()** si no hay parámetros.

```
() => Console.WriteLine("Hola");
```

Si solo hay **un parámetro**, los paréntesis se pueden obviar.

```
// Suponiendo que s es de tipo string la expresión lambda siguiente
// se evaluará a tipo delegado Func<string, int> esto es, una función anónima
// que recibe un string y retorna un int.
s => s.Length;
```

2. **=>** : Este es el **operador** que se encarga de mapear los parámetros a la expresión o al conjunto de sentencias del bloque.
3. La parte derecha, tras el operador **=>**, puede ser una única sentencia en forma de expresión que se evalúe a lo que retorna la función como en lo ejemplo anteriores o de forma menos habitual, un bloque de instrucciones. En este último caso la sintáxis será equivalente a la de las funciones tradicionales y tratando de sangrar el código y escribirla en varias líneas para mayor claridad.

```
(x, y) =>
{
    Console.WriteLine($"Calculando {x} elevado a {y}");
    4 return Math.Pow(x, y); // Fíjate que ahora sí debemos usar return.
}
```

Las expresiones lambda anteriores anteriores de forma análoga a lo que sucedería con la expresión `int x = 3 + 4`, deberemos asignarla a algo. Puesto que los tipos que referencian a funciones son los objetos delegado, deberemos asignarlo a algún identificador de objeto delegado parametrizado o no. Es más, **será completamente necesario para deducir los tipos de los parámetros de entrada y de retorno.**

```
// x es double y retorna double
Func<double, double> cubo = (x) => x * x * x;

// x , y son enteros y retorna entero
Func<int, int, int> suma = (o1, o2) => o1 + o2;

// ERROR no sé deducir los tipos de forma implícita.
// Esto solo sería válido en un lenguaje débilmente tipado como JavaScript.
var suma = (o1, o2) => o1 + o2;
```

”

*In software, when something is painful, the way to reduce the pain is to do it more frequently, not less..*

”

- David Farley

# Clausuras en CSharp

Para entender el concepto de **clausura**, vamos a ver la siguiente definición del lenguaje C#.

## Variables Externas Capturadas

En el cuerpo de expresión de una expresión lambda, podemos incluir referencias a variables locales y a los parámetros de un método. A estas referencias se les conoce como **variables externas capturadas**.

```
static void A()
{
    // Variable local factor, su alcance o scope es la el método A
    // fuera del mismo la variable dejará de existir.
    int factor = 3;

    // factor es una variable externa capturada en la expresión lambda pues
    // está siendo usada en un ámbito más cercano.
    Func<int, int> producto = n => n * factor;

    // Como hago una invocación del delegado dentro del scope de A
    // factor sigue existiendo y valiendo 3 por lo que al ejecutarse
    // el cuerpo de expresión de la función lambda a la que referencia
    // se mostrará 5 * 3 = 15
    Console.WriteLine(producto(5));
}
```

A las **expresiones lambda** que están integradas por variables externas capturadas se les llama **clausuras**.

Este tipo de variables, se evalúan en el momento en que se llama a la expresión Lambda de tal manera que si reescribiéramos el código anterior así ...

```

static void A()
{
    int factor = 3;
    Func<int, int> producto = n => n * factor;
    5 factor = 10;

    // Ahora mostrará 5 * 10 = 50 porque evalúa el valor de factor en
    // el momento de la llama a la función anónima.
    Console.WriteLine(producto(5)); // Mostrará 50
}

```

La referencia y el valor a una variable capturada, **está disponible mientras el ámbito de la expresión lambda** esté al alcance de su ejecución. Esto es, a pesar de encontrarnos fuera el ámbito de definición de la variable externa capturada, se guarda una referencia y siempre estará accesible para la expresión lambda.

Pero, en nuestro ejemplo `producto` que es la referencia a nuestra expresión lambda y `factor` que es la variable externa capturada, tienen el mismo ámbito de ejecución que es el método `A`, esto es, ambas referencias existen en el heap mientras estoy ejecutando `A` y al finalizar el método desaparecen. Entonces... **¿Cómo puede extenderse el ámbito de ejecución de `producto` más allá del ámbito de existencia de `factor`?**

La respuesta es, mediante **funciones de orden superior** o **HOF** ([High-Order Functions](#)). Las HOF en el  $\lambda$ -cálculo podemos resumir que **son funciones que reciben como parámetro o retornan otra función**.

¿Cómo pasamos por parámetro o devolvemos una función en C#?. La respuesta sería a través de un tipo delegado. Entonces, cuando vimos los delegados ... ¿Aquellos métodos que recibían un delegado eran HOF?. La respuesta es **sí**.

¿Cómo redactaremos el código anterior para que el método `A` sea una HOF y la función lambda (clausura) tenga un alcance de ejecución más allá de su ámbito de definición?. Una posible propuesta sería la siguiente ...

```

// Ahora el método A es HOF porque retorna una función
static Func<int, int> A()
{
    int factor = 3;
    return n => n * factor;
}

```

```
static void Main()
{
    var producto = A();
    // A() ha devuelto el control a Main y se supone que la variable local factor
    // ya ha desaparecido. Sin embargo, la clausura que la capturaba aún está
    // referenciada por el delegado producto y por tanto aún la podemos ejecutar.
    // Al no desaparecer el alcance de ejecución la variable capturada factor aún
    // no ha desaparecido de la memoria y tendrá el último valor que se le asignó
    // que es 3.
    Console.WriteLine(producto(5)); // Mostrará 15
}
```

## Resumen:

Es momento de realizar una breve recapitulación de conceptos...

En programación funcional ( $\lambda$ -cálculo) se denominan funciones de orden superior (HOF), aquellas que:

1. Reciben como parámetro otra función  $(r_1 \rightarrow r_2) \rightarrow r_3$
2. Devuelven como parámetro otra función  $r_1 \rightarrow (r_2 \rightarrow r_3)$
3. O ambas cosas  $(r_1 \rightarrow r_2) \rightarrow (r_3 \rightarrow r_4)$ .

En C# lo serán a aquellas funciones que reciban o devuelvan un delegado, o ambas cosas.

1. Reciben un delegado...

```
Func<int, int> func1 = d => d + 1;
Func<Func<int, int>, int, int> func2 = (func, d) => func(d) + 2;
// func2 es una HOF
int a = func2(func1, 3); // a = func1(3) + 2 = (3 + 1) + 2 = 6
```

2. Devuelven un delegado...

```
Func<int, Func<int, int>> suma = op1 => op2 => op1 + op2;
int a = suma(3)(2); // a = (op2 => 3 + op2)(2) = 3 + 2 = 5
```



Veamos un par de ejemplo de uso de este concepto de clausura con una utilidad más concreta pero simplemente para que terminemos de entender el concepto y la terminología ...

**Ejemplo 1:** Definiremos una HOF `Contador()` que me devuelva una clausura que al llamarla me devuelva un contador capturado incrementado.

```
static Func<int> Contador()
{
    int i = 0;
    return () => ++i; // Devuelvo la clausura que captura el contador i..
}

static void Main()
{
    // Cada cuenta captura una variable i local diferente por lo que
    // llevarán cuentas independientes
    var cuenta1 = Contador(); // Creo un contador
    var cuenta2 = Contador(); // Creo un segundo contador contador

    Console.WriteLine(cuenta1()); // Muestra 1
    Console.WriteLine(cuenta2()); // Muestra 1 por ser otro contador
    Console.WriteLine(cuenta2()); // Muestra 2
    Console.WriteLine(cuenta2()); // Muestra 3
    Console.WriteLine(cuenta1()); // Muestra 2
}
```

**Ejemplo 2:** Definiremos una HOF `Potencia(double y)` que me permita definir otras funciones que calculen potencias con un

```
class Ejemplo
{
    // HOF que devuelve una clausura que captura el parámetro de entrada 'y'.
    // Fíjate que estamos definiendo un cuerpo de expresión que se evalúa una
    // clausura que dada una base 'x' calcula 'x' elevado a la variable capturada 'y'.
    static Func<double, double> Potencia(double y) => x => Math.Pow(x, y);

    // Definimos una función que calcula cuadrados pasándole el exponente a capturar.
    static readonly Func<double, double> Cuadrado = Potencia(2);
    // Definición análoga para capturar el cubo.
    static readonly Func<double, double> Cubo = Potencia(3);

    static void Main()
    {
        Console.WriteLine(Cuadrado(10)); // x = 10 elevado a y = 2 mostrará 100
        Console.WriteLine(Cubo(10));     // x = 10 elevado a y = 3 mostrará 1000
    }
}
```

Si vamos al principio del tema donde definíamos lo que era la programación funcional, decíamos que:

*'La **programación funcional** es un **paradigma de programación declarativa** basado' ...*

Para entender un poco mejor a que se refiere con '*programación declarativa*', vamos a comparar y resumir de forma sencilla la diferencia entre la programación imperativa y declarativa. Diremos que, en la **imperativa** le decimos nosotros al ordenador '**cómo tiene que hacer algo**' mediante una secuencia de instrucciones que sería más o menos lo que hemos hecho hasta ahora. Sin embargo, en la **declarativa** le decimos al ordenador '**lo que tiene que hacer**', proporcionándole únicamente las estrategias para hacerlo. Por poner un ejemplo, el lenguaje SQL sería claramente declarativo ya que en él, le decimos al SGBD por ejemplo que seleccione n-tuplas que cumplan un criterio y les aplique una estrategia de agrupación, pero '**no le indicamos cómo tiene que hacerlo**', de hecho no sabemos cómo lo hace internamente.

Sin entrar en aspectos más teóricos o formales matemáticos como quizá hemos hecho al hablar del  $\lambda$ -cálculo, diremos que la gran mayoría de lenguajes modernos proporcionan una serie de métodos para trabajar sobre secuencias de datos de forma similar a cómo lo haríamos en SQL aplicando programación funcional.

En C# estos métodos para trabajar con secuencias ( `IEnumerable<T>` ) están definidos como [métodos de extensión](#) en `System.Linq` como ya comentamos al hablar de este tipo de métodos.

Entre ellos podremos destacar: `Select` , `Where` , `Agregate` , `GroupBy` , `Distinct` , `ElementAt` , `Join` , `GroupJoin` , `OrderBy` , `Reverse` , `SelectMany` , etc.

# Patrón Map-Filter-Fold

Es un patrón funcional utilizado en numerosos lenguajes tan populares como JavaScript(>ES6), Java(>8), C#, Python, Scala, Swift, Kotlin, etc y que permite hacer operaciones declarativas sobre secuencias de elementos de forma análoga a SQL.

📌 **Nota 1:** Detrás de este patrón hay una gran teoría matemática aunque nosotros la vamos a obviar y vamos a tratar de explicar las operaciones de forma más didáctica. Si quieres profundizar en el tema puedes seguir los enlaces en el tema.

📌 **Nota 2:** Durante el resto del tema vamos a utilizar la siguiente notación para representar secuencias (colecciones) y objetos...

- **Corchetes** para secuencias [ e1, e2, e3, ..., en ] o **Sec**[ e1, e2 ]
- **Llaves** para objetos { Propiedad1=valor1, Propiedad2=valor2 }

## Map (o Select)

**Map** es una HOF que aplica una función única de '*mapeo*' o **transformación** a cada elemento de la secuencia y devuelve una nueva secuencia que contiene los resultados de la transformación en el mismo orden de la secuencia de entrada.

A lo largo de este tema vamos a ejemplificar los conceptos de forma genérica con emojis, porque realmente es un concepto abstracto y estos nos ayudarán a entender la abstracción de forma didáctica.

Supongamos que representamos **map** con la expresión **map = (A → B) × Sec[A] → Sec[B]** donde:

- **A** son productos en crudo o sin procesar.  
Ej. el maíz 🌽 .
- **B** son productos cocinados o procesados.  
Ej. las palomitas de maíz 🍿 .
- **A → B** sería la función que cocina o **transforma** un producto en crudo en un producto cocinado y la podemos llamar por ejemplo **cook**. Esta función se pasará como parámetro a la función **map** (por eso **map** es una HOF).  
Ej. **cook**( 🌽 ) → 🍿 .
- **Sec[A]** sería la secuencia de **entrada** de la función **map** de productos en crudo.  
Ej. [ 🐼, 🐟, 🍓, 🐔, 🌽 ]
- **Sec[B]** sería la secuencia de **salida** de la función **map** de productos ya cocinados o procesados por la función **cook**.

Ej. [🍔, 🌀, 🍷, 🍗, 🍿]

Con lo cual podríamos decir que, **map = cook × ProductosCrudos → ProductosCocinados**.  
Esto es, map dada una secuencia de productos en crudo le aplica a cada producto la **función de transformación** cocinar y me dará como resultado una secuencia de productos cocinados.

Ej. [🍷, 🍷, 🍷, 🍷, 🍷].map(cook) → [🍔, 🌀, 🍷, 🍗, 🍿]

En C# lo aplicaremos a través del método extensor **Select** ...

```
// Se aplica a secuencia de entrada IEnumerable<A>
// A cada elemento de la secuencia de entrada se le aplica el 'mapeo' A => B
// Retorna una secuencia de salida IEnumerable<B> con los
// elementos de la secuencia de entrada mapeados.
IEnumerable<B> Select<A, B>(this IEnumerable<A> source, Func<A, B> mapeoDeAenB);
```

Veamos su uso a través de un ejemplo sencillo donde transformamos una secuencia de números reales en su equivalente entero. '*Mapeando*' de real a entero cada uno de elementos de la secuencia de entrada.

```
// Dada la siguiente secuencia de números reales
double[] reales = { 1.3, 3.4, 4.5, 5.6, 8.7 };

// Aplicamos la función ToInt32 que cumple con el tipo delegado Func<double, int>
// para transformar la secuencia a enteros.
IEnumerable<int> enteros = reales.Select(Convert.ToInt32);

Console.Write(string.Join(", ", enteros));
```

## Filter (o Where)

**Filter** es una HOF que aplica un predicado a cada elemento de la secuencia de entrada y devuelve una secuencia de salida con los elementos en la secuencia que cumplen el predicado.

Supongamos que representamos **filter** con la expresión **filter = (A → bool) × Sec[A] → Sec[A]** donde:

- **A** son productos cocinados o preparados.  
Ej. una hamburguesa 🍔.
- **A → bool** sería el predicado que evalúa si los elementos de **A** cumplen un determinado criterio o no. Este predicado se pasará como parámetro a la función **filter**.  
Ej. **isVegetarian(🍔) → false**.

- **Sec[A]** sería la secuencia de **entrada** de la función **filter** de productos ya cocinados.  
Ej. [ 🍔, 🌀, 🍟, 🍷, 🍿 ]
- **Sec[A']** sería la secuencia de **salida** de la función **filter** de productos ya cocinados que cumplen el predicado **isVegetarian**, esto es,  $A' \subseteq A$ .  
Ej. [ 🍟, 🍿 ]

Con lo cual podríamos decir que, **filter = isVegetarian × ProductosCocinados** → **ProductosCocinadosVegetarianos**. Esto es, filter dada una secuencia de productos ya cocinados me devolverá aquellos que sean vegetarianos.

Ej. [ 🍔, 🌀, 🍟, 🍷, 🍿 ].filter(isVegetarian) → [ 🍟, 🍿 ]

En C# lo aplicaremos a través del método extensor **Where** ...

```
// Dada la siguiente secuencia de enteros.
int[] enteros = { 1, 3, 4, 5, 8 };

// Filtramos aquellos que sean pares...
// 1. Definimos el predicado para ver si un número es par.
Func<int, bool> esPar = n => n % 2 == 0;
// 2. Aplicamos el predicado a la función where
IEnumerable<int> enterosPares = enteros.Where(esPar).ToList();
Console.WriteLine(string.Join(", ", enterosPares));
```

## Fold (o Aggregate)

**Fold** es una HOF también conocida como **reduce**, **aggregate**, **accumulate**, etc. que '*combina*' los elementos de la secuencia de entrada reduciéndolos a un único elemento de retorno. Para obtener este elemento de retorno necesitaremos una función binaria que combine los elementos de la secuencia en dicho elemento de retorno que normalmente tendrá un estado inicial.

Supongamos que representamos **fold** con la expresión **fold = (B × A → B) × Sec[A] × B → B** donde:

- **B** es el elemento de retorno el cual tomará un valor inicial.  
Ej. Pepe que **inicialmente** está hambriento y cansado 🥱 pero puede tener otros estados como satisfecho 😊
- **A** son los productos cocinados o preparados.  
Ej. la hamburguesa de vacuno 🍔
- **Sec[A]** obviamente es una secuencia de productos cocinados o preparados.  
Ej. [ 🍔, 🌀, 🍟, 🍷, 🍿 ]

- **B × A → B** función binaria que combina **B** con **A** y se evalúa a **B**.

Ej. En nuestro ejemplo, esta función de combinación puede ser **eat**(😞, 🍔) → 😊 donde el resultado de combinar a 😞 (Pepe hambriento) con la 🍔 (hamburguesa) sería 😊 (Pepe satisfecho). Ojo!, que también puede pasar que al comer (combinar), el estado de pepe no cambie y siga hambriento **eat**(😞, 🍔) → 😞

💡 **Tip:** En otras palabras, **eat** es una función de plegado (**fold**), reducción (**reduce**), agregación (**aggregate**), etc. de la hamburguesa sobre Pepe y que me da como resultado al propio Pepe con menos hambre o saciado...

Con lo cual podríamos decir que, **fold = come × ProductosCocinados × PepeHambriento → PepeSatisfecho**. Esto es, fold dado un Pepe hambriento y una secuencia de productos cocinados, me devolverá un Pepe satisfecho.

Ej. [🍔, 🍔].fold(😞, eat) → 😊

En C# lo aplicaremos a través de las sobrecargas del método extensor **Aggregate** ...

🔴 **Nota:** **System.Linq** nos proporciona muchas funciones de agregación ya implementadas como métodos de extension sobre secuencias tales como: **Sum**, **Max**, **Min**, **Average**, o **Count**.

```
// Dada la siguiente secuencia de notas enteras
int[] notas = { 10, 3, 4, 5, 8, 2 };

// Cuenta las notas mayores o iguales a 5.
// 1. Definimos la función binaria de agregación que vamos apasar a Aggregate....
// c -> Es el contador.
// n -> Es un elemento de la secuencia (una nota).
// Se evalúa a la cuenta incrementada o sin incrementar dependiendo de
// si nota >= 5 o no.
Func<int, int, int> cuentaAprobados = (c, n) => n >= 5 ? c + 1 : c;

// 2. Aplicamos la HOF de agregación, pasándole el valor inicial de la cuenta y
// la función agregadora.
int aprobados = notas.Aggregate(0, cuentaAprobados);
Console.WriteLine($"Total aprobados: {aprobados}");
```

**Aggregate** por defecto recorrerá la secuencia de izquierda a derecha y para el caso anterior de contar aprobados nos daría igual. Sin embargo, muchos lenguajes funcionales ofrecen ambas posibilidades:

- **foldl** : (fold left) Recorre como **Aggregate** de izquierda a derecha.

- **foldr** : (fold right) Recorre la secuencia de derecha a izquierda.

Veamos un ejemplo de como hacerlo en C#. Para ello, vamos a suponer que queremos implementar el método de utilidad definido en la clase **String** y que hemos usado muchas veces a lo largo del curso. **string.Join(string? separador, IEnumerable<string?> textos)**.

Si nos fijamos cumple con  $(B \times A \rightarrow B) \times \text{Sec}[A] \times B \rightarrow B$  pero ahora no obtendremos el mismo resultado si hacemos **foldl** o **foldr** ...

- **A** Serán una palabra ( **String** ) a concatenar.
- **Sec[A]** Será la secuencia de palabras a concatenar.
- **B** Ahora el tipo 'agregador' sería un **StringBuilder** donde vamos a concatenar todas las palabras con el separador adecuado.
  - Valor inicial será **new StringBuilder(palabras.Length > 0 ? palabras.First() : "")**, esto es, un **StringBuilder** vacío si la secuencia esta vacía o uno que contenga la primera palabra de la misma si contiene alguna.
  - Valor final será el **StringBuilder** inicial **B** al que le he concatenado todas las palabras por la derecha.
- $(B \times A \rightarrow B)$  La función binaria de agregación será de tipo **Func<StringBuilder, string, StringBuilder>** y recibirá el **StringBuilder B** con lo que llevo de la cadena concatenada, la siguiente palabra **A** a concatenar con el separador y retornará el **StringBuilder B** con la palabra concatenada por la derecha.

Podemos resumir de forma simple que **Join** se comporta como **foldl** porque entra una secuencia de cadenas y devuelve un único tipo resultado de concatenar las cadenas con un separador determinado.

La única diferencia es que **Join** devuelve un **string** y nosotros vamos a devolver un **StringBuilder** por eficiencia.

Veamos pues nuestra propuesta de implementación de **Join** con un **Aggregate** ( **foldr** ) en C# ...



```
// Definimos la secuencia
string[] palabras = { "Once", "upon", "a", "time" };

// Definimos el separador que será una variable capturada en la función lambda.
const string separador = " ";

// Función lambda agregadora concatena...
// Si la palabra a concatenar no es nula y no coincide con el valor inicial de la
// concatenación o en otras palabras es la primera palabra de la secuencia la
// concatenamos con el valor del separador capturado.
Func<StringBuilder, string, StringBuilder> concatena =
(f, p) => f.Append(p != null && f.ToString() != p ? $"{separador}{p}" : "");

StringBuilder frase = palabras.Aggregate(
    new StringBuilder(palabras.Length > 0 ? palabras.First() : ""),
    concatena);

Console.WriteLine(frase); // Mostrará "Once upon a time"
```

pero...¿y si quisiéramos hacer un `foldr` en C#?. Sería *'tan simple'* como **invertir la secuencia de entrada** con `Reverse()` y el lugar de tomar la primera ocurrencia ( `First` ) de la secuencia para inicializar nuestro `StringBuilder` , tomaríamos la última `Last` .

```
StringBuilder frase = palabras
    .Reverse()
    .Aggregate(
        new StringBuilder(palabras.Length > 0 ? palabras.Last() : ""),
        concatena);

Console.WriteLine(frase); // Mostrará "time a upon Once"
```

**Aggregate** además tiene varias sobrecargas y vamos a comentar una de ellas que nos puede ser interesante en este caso.

Si vamos a ver la definición de **Aggregate** para el uso anterior tendremos el siguiente interfaz que ya deberíamos entender...

```
// TSource es la parametrización de A
// TAccumulate es la parametrización de B
public static TResult Aggregate<TSource, TAccumulate>(
    this IEnumerable<TSource> source,           // Seq[A]
    TAccumulate seed,                          // B (inicial)
    Func<TAccumulate, TSource, TAccumulate> func); // B × A → B
```

Sin embargo, tenemos también la sobrecarga ...

```
public static TResult Aggregate<TSource, TAccumulate, TResult>(
    this IEnumerable<TSource> source,
    TAccumulate seed,
    Func<TAccumulate, TSource, TAccumulate> func,
    Func<TAccumulate, TResult> resultSelector);
```

donde aparece otro nuevo tipo parametrizado `TResult` y función de entrada

`Func<TAccumulate, TResult> resultSelector` que una vez realizado todo el proceso de **'fold'** me transformará (mapeará) mi `TAccumulate` final en un `TResult`.

En nuestro ejemplo, al definir el `TAccumulate` como un `StringBuilder` por optimización, el resultado del `Aggregate` será un `StringBuilder` y no un `string` como nos proporcionaría el `string.Join(...)`. Pero la función de mapeo `Func<TAccumulate, TResult> resultSelector` nos permitirá convertir el `StringBuilder` a un `string` con `sb => sb.ToString()` y será lo que terminará devolviendo `Aggregate`.

Nuestra expresión equivalente al `Join` haciendo un `Aggregate` (`foldl`) finalmente quedaría ...

```
string[] palabras = { "Once", "upon", "a", "time" };
const string separador = " ";

// Con Join
string frase = string.Join(separador, palabras);

// Con Aggregate
string frase = palabras.Aggregate(
    new StringBuilder(palabras.Length > 0 ? palabras.First() : ""),
    (f, p) => f.Append(f.ToString() != p && p != null ? $"{separador}{p}" : ""),
    sb => sb.ToString());

Console.WriteLine(frase);
```

👉 **Importante:** Fíjate que en esta ocasión hemos pasado las funciones de agregación y mapeo directamente a **Aggregate** en lugar de asignarlas previamente a un delegado. Esto es posible porque **C# es capaz de deducir los tipos** de  $(f, p) \Rightarrow$  y de  $sb \Rightarrow$  a partir del tipo de la secuencia **palabras** y el tipo del objeto que le pasamos como acumulador.

📌 **Nota:** Obviamente en este caso, es muchísimo más simple utilizar la función ya implementada **Join**. Pero hemos puesto este ejemplo, además de para ver el concepto de **foldl** y **foldr**, para entender la potencia de esta función si entendemos bien lo potente que es esta operación conceptualmente.

De momento la hemos usado para cosas tan diferentes como: que Pepe coma 😊, contar aprobados o concatenar palabras con un separador.

## Combinando las tres operaciones

Podemos combinar dichas operaciones para realizar nuestro proceso en una única función de forma declarativa.

Imaginemos nuestro ejemplo en que queremos alimentar a Pepe, que recordemos que es vegetariano y partimos de un montón de productor en crudo. Podríamos hacer...

`[🐷, 🐟, 🍅, 🐔, 🌿].map(cook).filter(isVegetarian).fold(😞, eat) → 😊`

Esto equivaldría a las operaciones anteriores de forma encadenada. Donde la secuencia de salida de una función pasa a ser la de entrada de la siguiente  $[🐷, 🐟, 🍅, 🐔, 🌿] \rightarrow [🍔, 🍷, 🍟, 🍌, 🍩] \rightarrow [🍟, 🍩] \rightarrow 😊$

Sin embargo es importante preguntarnos si el orden de aplicación es eficiente. Fíjate que en este caso, hemos tenido que 'cocinar' todos los alimentos aunque realmente Pepe solo se ha comido dos de ellos. Este proceso hubiera sido muchísimo más eficiente si hubieramos cocinado solo aquellos que 'son vegetales'  $[🐷, 🐟, 🍅, 🐔, 🌿] \rightarrow [🍅, 🌿] \rightarrow [🍟, 🍩] \rightarrow 😊$ , esto es, haciendo el filtrado antes que la transformación ...

`[🐷, 🐟, 🍅, 🐔, 🌿].filter(isVegetarian).map(cook).fold(😞, eat) → 😊`

📌 **Nota:** Esto último sería posible solo si la función de filtrado ahora tuviese como conjunto **A** a los productos crudos o una **abstracción** de ambos, es decir, si se pudiera saber si un producto 'es vegetal' independientemente de si está cocinado o no **isVegetarian(🌿) → true** y **isVegetarian(🍟) → true**

**Ejemplo:** Imaginemos que queremos saber el total de aprobados de una lista de notas con decimales.

```
// Select (map):      Transforma de double a double sin decimales.
//                  Puesto que Math.Round tiene varias sobrecargas, deberemos indicar
//                  mediante un cast que queremos una función del tipo A → B
//
// Where (filter):    Filtramos aquellas notas que que cumplen el predicado n >= 5d
//
// Aggregate (fold):  Vamos a llevar una cuenta en c empezando en 0d y al final del proceso
//                  convertimos la cuenta real en un valor entero. Esto último no sería
//                  necesario, si el valor inicial de la cuenta hubiera sido un literal
//                  entero, pero lo hacemos así para ves un ejemplo de uso de resultSelecto

double[] notas = {1.3, 3.4, 4.6, 5.6, 6.7, 8.7};
int aprobados = notas.Select((Func<double, double>)Math.Round)
    .Where(n => n >= 5d)
    .Aggregate(0d, (c, n) => c + 1, c => Convert.ToInt32(c));
Console.WriteLine($"Aprobados: {aprobados}");
```

Puesto que cosas como contar elementos en una secuencia, acumular en una suma, buscar un máximo, etc. son casos típicos de fold, ya vienen predefinidos en C# y no tendríamos que implementarlos a través de Aggregate

```
int aprobados = notas.Select((Func<double, double>)Math.Round).Where(n => n >= 5d).Count();
```

Fíjate que es bastante similar a SQL, una posible consulta para hacer lo mismo en SQL.

```
SELECT COUNT(ROUND(nota)) AS aprobados FROM notas where ROUND(nota) >= 5;
```

En este caso no podemos filtrar antes de hacer el mapeo puesto que el resultado sería diferente. Piensa que **4.6** después del **Round** será **5** y contará como aprobado.

## Ampliación:

Cómo hemos dicho este patrón existirá de forma similar en otros lenguajes y seguramente sin saber la sintaxis de otros lenguajes seguramente podemos entender el código equivalente en lenguajes tan populares como:

### JavaScript:

```
let notas = [1.3, 3.4, 4.6, 5.6, 6.7, 8.7];
let aprobados = notas.map(Math.round)
                      .filter(n => n >= 5)
                      .reduce((c, n) => c + 1, 0);
document.write(`Aprobados: ${aprobados}`);
```

### Kotlin:

```
val notas = doubleArrayOf(1.3, 3.4, 4.6, 5.6, 6.7, 8.7)
val aprobados = notas.map { Math.round(it) }
                  .filter { it >= 5 }
                  .fold(0) { acc, _ -> acc + 1 }
println("Aprobados: $aprobados")
```

### Python:

```
from functools import reduce
notas = [1.3, 3.4, 4.6, 5.6, 6.7, 8.7]
aprobados = reduce(lambda c, n: c + 1,
                  filter(lambda n: n >= 5,
                          map(round, notas)),
                  0)
print(f'Aprobados: {aprobados}')
```

# Otras operaciones funcionales declarativas en C#

Ya hemos visto que `Select`, `Where`, `Aggregate`, `Count` y sin duda estas operaciones como hemos mencionado nos deben 'sonar' de SQL. Pero..., ¿Existen otras operaciones equivalentes a las que podemos encontrar en SQL?

la respuesta es **sí** y vamos ver algunas de ellas ...

## OrderBy/OrderByDescending

Ordena de manera ascendente/descendente los elementos de una secuencia en función de una clave.

✦ **Nota:** Los tipos deben ser **comparables**. Si no lo son, necesitará de un objeto que implemente el interfaz `IComparer<T>` entre dos elementos del tipo de la clave por la que hemos decidido ordenar.

## Distinct

Elimina valores repetidos en una secuencia.

✦ **Nota:** El tipo de datos de la secuencia deberá implementar `IEquatable` para poder comparar en igualdad los elemento de la secuencia o deberemos pasarle un objeto que implemente un interfaz de comparación.

**Ejemplo:** Imaginemos que queremos saber la mayor nota redondeada de la siguiente secuencia...

```
double[] notas = { 1, 3.4, 4.3, 4.6, 4.3, 7.2, 7.6, 5.6, 8.7 };
```

Podíamos usar un `Aggregate` para obtener el máximo o usar la función `Max` que hace la operación específica de la siguiente forma.

```
int notaMayor = notas.Aggregate(
    double.MinValue,
    (m, n) => n > m ? n : m,
    n => Convert.ToInt32(Math.Round(n)));

// o también
int notaMayor = notas.Max(n => Convert.ToInt32(Math.Round(n)));

Console.WriteLine($"La nota redondeada mayor es {notaMayor}");
```

En este caso también sería bastante similar a un posible consulta en SQL.

```
SELECT MAX(ROUND(nota)) AS notaMayor FROM notas;
```

Aunque más ineficiente, también podríamos usar `Distinct` y `OrderBy` de la siguiente manera ...

```
double[] notas = { 1, 3.4, 4.3, 4.6, 4.3, 7.2, 7.6, 5.6, 8.7 };
int notaMayor = notas.Select(n => Convert.ToInt32(Math.Round(n)))
    // 1. Redondeamos las notas de la secuencia a su valor entero.
    .Distinct()
    // 2. Eliminamos de la secuencia las notas repetidas.
    .OrderByDescending(n => n)
    // 3. Ordenamos la secuencia de en orden descendiente, esto es,
    // la nota mayor es la primera.
    .First();
    // 4. Obtenemos el primer elemento de la secuencia.
Console.WriteLine($"La nota redondeada mayor es {notaMayor}");
```

Fíjate que aunque es un poco más enrevesada, también tenemos una equivalencia en SQL.

```
SELECT DISTINCT ROUND(nota) AS notaMayor FROM notas ORDER BY notaMayor DESC LIMIT 1;
```

## GroupBy

Agrupar los elementos de una secuencia según una función del selector de claves especificada y crea un valor de resultado a partir de cada grupo y su clave. Los elementos de cada grupo se proyectan utilizando una función determinada.

Existen diferentes sobrecargas posibles para usar la agrupación. Nosotros vamos a usar esta pues es la más *'intuitiva'*...

```
IEnumerable<TResult> GroupBy(  
    this IEnumerable<TSource> source,  
    Func<TSource, TKey> keySelector,  
    Func<TKey, IEnumerable<TSource>, TResult> resultSelector);
```

Veamos como interpretarla:

- **keySelector** será la función del selector de claves por las que agruparé la secuencia:
  - **Recibe:** `TSource` que es el tipo de la secuencia a procesar.
  - **Devuelve:** `TKey` tipo la clave resultado del mapeo de `TSource` por que voy a agrupar.
- **resultSelector** será la función que proyecta los elemento de cada grupo con su clave:
  - **Recibe:** `Tkey` con la clave por la que agrupo y la agrupación de los objetos `TSource` para esa clave.
  - **Devuelve:** `TResult` con lo que quiero producir en la nueva secuencia resultado de la agrupación.

Bueno, hasta ahora ha sido una definición muy formal, pero vamos ha hacerlo con un ejemplo más sencillo para entenderlo. Supongamos nuestro array de notas anterior...

```
double[] notas = { 1, 3.4, 4.3, 4.6, 4.3, 7.2, 7.6, 5.6, 8.7 };
```

Queremos agrupar el valor de las notas sin decimales y contabilizar el número de apariciones de las mismas.

Si estuviéramos en MySQL una posible consulta para resolverlo sería...

```
SELECT FLOOR(nota) AS notaRedondeada, count(*) AS veces  
FROM notas GROUP BY notaRedondeada ORDER BY notaRedondeada;
```

El equivalente utilizando la programación funcional y el **GroupBy** de C# sería ...



```

string salida = "notaRedondeada veces\n" +
notas.GroupBy(
    nota => Math.Floor(nota),
    (nota, elementosConMismaNota) =>
        new
        {
            NotaAgrupada = nota,
            Veces = elementosConMismaNota.Count()
        })
.OrderBy(datoAgrupado => datoAgrupado.NotaAgrupada)
.Aggregate("",
    (texto, datoAgrupado) =>
        texto += $"{datoAgrupado.NotaAgrupada, -14} {datoAgrupado.Veces, -5}\n");
Console.WriteLine(salida);

```

👉 **Importante:** Fíjate que aunque `string salida = ...` es una única expresión, hemos ido introduciendo saltos de línea e indentado el código para que sea legible y modificable nuestra expresión funcional. De otra manera el código sería prácticamente ilegible y difícil de modificar para un programador humano.

Si comentamos el código anterior podríamos decir que:

1. Nuestra función selectora ( `Func<TSource, TKey> keySelector` ) es:

```

nota => Math.Floor(nota)

```

Siendo `TSource` los **doubles** de la secuencia de entrada con el identificador `nota` .

Siendo `TKey` el **double** resultado de quedarnos con la parte entera de la nota

`Math.Floor(nota)` .

Todos los elementos de la lista de entrada que produzcan la misma clave `TKey` , estarán en el mismo grupo y por tanto se agruparán en una secuencia. En nuestro caso las claves generadas serán ...

```

[ 1, 3.4, 4.3, 4.6, 4.3, 7.2, 7.6, 5.6, 8.7 ] → [ 1, 3, 4, 5, 7, 8 ]

```

2. Nuestra función de proyección ( `Func<TKey, IEnumerable<TSource>, TResult> resultSelector` ) es:

```

(nota, elementosConMismaNota) => new
{
    NotaAgrupada = nota,
    Veces = elementosConMismaNota.Count()
}

```

Siendo `TKey` el **double** con la clave que recordemos es el resultado de aplicar `Floor` a la nota.

Siendo `IEnumerable<TSource>` una nueva secuencia con los elementos de la secuencia original, esto es, las notas con decimales **agrupadas a esa clave**. Esto es, los elementos que generaron esa clave.

Así pues, tendré las siguientes parejas de **(clave, agrupación)** en

`(TKey, IEnumerable<TSource>)` en nuestro caso `(nota, elementosConMismaNota)`

```
(1, [ 1 ])  
(3, [ 3.4 ])  
(4, [ 4.3, 4.6, 4.3 ])  
(7, [ 7.2, 7.6 ])  
(5, [ 5.6 ])  
(8, [ 8.7 ])
```

Siendo `TResult` la **proyección** resultante de gestionar los pares (clave, agrupación) anteriores. En nuestro caso objetos de la 'anónimos' con dos propiedades `NotaAgrupada` y `Veces` que tendrá la cuenta de cada secuencia de notas agrupadas con la clave.

Por tanto el resultado final será la secuencia de `TResult` siguiente...

```
[  
  { NotaAgrupada = 1, Veces = 1 }, { NotaAgrupada = 3, Veces = 1 },  
  { NotaAgrupada = 4, Veces = 3 }, { NotaAgrupada = 7, Veces = 2 },  
  { NotaAgrupada = 5, Veces = 1 }, { NotaAgrupada = 8, Veces = 1 }  
]
```

### 3. La función

```
.OrderBy(datoAgrupado => datoAgrupado.NotaAgrupada)
```

aplicada a la secuencia anterior de objetos anónimos producirá la misma secuencia pero ordenada por la propiedad `NotaAgrupada`.

### 4. La función

```
.Aggregate("", (texto, datoAgrupado) =>  
    texto += $"{datoAgrupado.NotaAgrupada, -14} {datoAgrupado.Veces, -5}\n");
```

Genera una cadena con los pares nota sin decimales y veces que aparece alineados y separados por un salto de línea.

**Ejemplo:** Veamos un ejemplo algo más elaborado usando una secuencia con un tipo algo más complejo que un double. Para ello, supongamos la siguiente clase inmutable que define los datos de un empleado de una determinada empresa...

```
public enum Ciudad { Elche, Alicante };

public record Empleado(string Nombre, int Edad, Ciudad Ciudad)
{
    public override string ToString() => $"{Nombre,-9}{Edad,-3}{Ciudad}";
}
```

Ahora definimos una clase de utilidad que me devuelva una secuencia de empleados de forma *'perezosa'*...

```
public static class Empleados
{
    public static IEnumerable<Empleado> DepartamentoDeVentas
    {
        get
        {
            yield return
                new(Nombre: "Xusa", Edad: 45, Ciudad: Ciudad.Alicante);
            yield return
                new(Nombre: "Simon", Edad: 51, Ciudad: Ciudad.Alicante);
            yield return
                new(Nombre: "Carmen", Edad: 27, Ciudad: Ciudad.Elche);
            yield return
                new(Nombre: "Juanjo", Edad: 52, Ciudad: Ciudad.Elche);
            yield return
                new(Nombre: "Emy", Edad: 52, Ciudad: Ciudad.Elche);
        }
    }
}
```

Queremos obtener un array con los nombres de los empleados mayores de 40 años sin repeticiones y ordenados por nombre.

```

string[] nombres = Empleados.DepartamentoDeVentas
    .Where(e => e.Edad > 40) // Filtramos por edad.
    .Select(e => e.Nombre)  // Proyectamos la propiedad Nombre
                           // a una nueva secuencia.
    .OrderBy(n => n)        // Ordenamos por nombre.
    .Distinct()            // Eliminamos repetidos.
    .ToArray();            // Pasamos la secuencia a array.

Console.WriteLine(string.Join(", ", nombres));

```

Supongamos que ahora queremos obtener los empleados mayores de 40 años agrupados por ciudad usando **GroupBy**.

```

var empleadosXCiudad = Empleados.DepartamentoDeVentas
    .Where(e => e.Edad > 40)
    .GroupBy(e => e.Ciudad,
        (c, g) => new {Ciudad = c, Empleados = g});

```

El tipo de **empleadosXCiudad** debe ser implícito ( **var** ) pues la secuencia que estamos creando es de un tipo implícito ( **new {Ciudad = c, Empleados = g}** ) que estamos definiendo en la función de proyección del **GroupBy**.

Ahora queremos mostrar los datos de la siguiente manera con los empleados ordenados por edad ....

```

Alicante:
    Xusa      45 Alicante
    Juanjo    51 Alicante
Elche:
    Juanjo    45 Elche
    Emy       52 Elche

```

Aunque la secuencia sea de un tipo implícito podremos componer una cadena con la salida deseada.

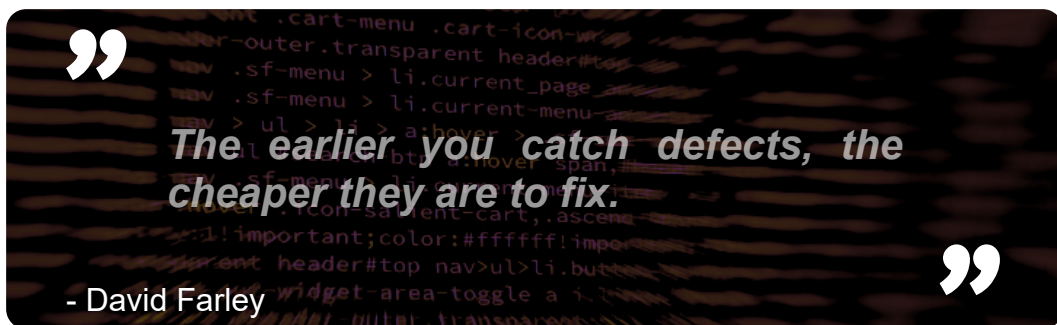
```

StringBuilder salida = new StringBuilder();
foreach (var eXc in empleadosXCiudad)
{
    salida.Append($"{eXc.Ciudad}:\n");
    foreach (Empleado e in eXc.Empleados.OrderBy(e => e.Edad))
        salida.Append($"  \t{e}\n");
}
Console.WriteLine(salida);

```


✦ **Nota:** Al ser la secuencia de un tipo implícito o anónimo, solo podremos recorrer la secuencia en el mismo ámbito de uso de tipo. Si intentáramos encapsular la obtención de la secuencia en una función, no podríamos devolverla como tal, sino que deberíamos encapsularla en un tipo de datos o clase que la contuviera.

Lo mismo sucedería si quisieramos encapsular el código que muestra los datos en pantalla. No sabríamos de que tipo asignarle a la secuencia que pasaríamos como parámetro.



## Resultado del GroupBy en un nuevo tipo de datos

Supongamos que hubiéramos querido encapsular la obtención de `empleadosXCiudad` en una función y hacer el mismo proceso modularizado. Puesto que la secuencia es de un tipo anónimo, nos hubiéramos visto obligados a definir una nueva clase con una propiedad `Ciudad` del tipo enumerado y otra `Empleados` como secuencia de objetos empleados por ejemplo.

 **Importante:** Al este tipo de datos o clase que definimos para obtener el resultado de una consulta y que únicamente tienen como misión transportar los datos de la consulta, se les denomina **DTO Data Transfer Object** y deben ser inmutables y tener la mínima funcionalidad posible. Por eso el modificador `record` es el más adecuado para definirlos.

```
// Definimos el tipo.
record EmpleadosPorCiudadDto(Ciudad Ciudad, IEnumerable<Empleado> Empleados);

// Podemos definir un valor de retorno con un tipo concreto para la secuencia.
// En este caso hemos quitado la restricción de mayores de 40 por simplificar.
private static IEnumerable<EmpleadosPorCiudadDto> EmpleadosVentasPorCiudad() =>
    Empleados.DepartamentoDeVentas
        .GroupBy(e => e.Ciudad,
            (c, g) => new EmpleadosPorCiudadDto(Ciudad: c, Empleados: g));

// Al tener un tipo concreto, también podemos modularizar la composición de la salida.
private static string ATexto(IEnumerable<EmpleadosPorCiudadDto> empleadosXCiudad)
{
    StringBuilder salida = new StringBuilder();
    foreach (var eXc in empleadosXCiudad)
    {
        salida.Append($"{eXc.Ciudad}:\n");
        foreach (Empleado e in eXc.Empleados.OrderBy(e => e.Edad))
            salida.Append($"{e}\n");
    }
    return salida.ToString();
}

// Posteriormente hacer la llamada a las funciones.
Console.WriteLine(ATexto(EmpleadosVentasPorCiudad()));
```

En este caso no hemos filtrado por edad como hacíamos en el caso inicial. Pero si quisiéramos hacerlo a posteriori, podríamos hacerlo de la siguiente manera...

```

var empleadosXCiudad = EmpleadosVentasPorCiudad();

// Mapeamos al mismo DTO pero filtrando la propiedad Empleados por edad.
// Fíjate que al ser inmutable EmpleadosPorCiudadDto,
// para no tener que crear un nuevo objeto copiando propiedad a propiedad.
// Usamos el operador with para crear una nueva instancia
// con la propiedad Empleados filtrada.
var empleadosXCiudadMas40 = empleadosXCiudad.Select(
    eXc => eXc with { Empleados = eXc.Empleados.Where(e => e.Edad > 40) });

Console.WriteLine(ATexto(empleadosXCiudadMas40));

```

## Resultado del GrupBy en un diccionario

Pero, ¿Habría la posibilidad de retornar la agrupación por clave sin tener que definir el tipo `EmpleadosPorCiudadDto`? Podríamos utilizar la posibilidad de transformar cualquier secuencia a las colecciones definidas en el lenguaje.

Por ejemplo vamos a definir la misma función `EmpleadosVentasPorCiudad` pero en lugar de retornar `IEnumerable<EmpleadosPorCiudadDto>` ahora retornará `Dictionary<Ciudad, List<Empleado>>` y así no tendríamos que definir el tipo.

Es fácil e inmediato transformar una secuencia a un array o una lista.

```

IEnumerable<Empleado> secuencia = ...;

Empleado[] arrayEmpleados = secuencia.ToArray();
List<Empleado> listaEmpleados = secuencia.ToList();

```

Pero si queremos crear un diccionario, deberíamos poder indicarle de donde sacamos las claves y los valores asociados infiriendo en ambos casos los tipos parametrizados en el diccionario. Una vez más nos ayudarán la funciones de orden superior (HOF) para hacerlo en este caso podemos usar la siguiente definición de `ToDictionary` ...

```

public static Dictionary<TKey, TElement>
ToDictionary<TSource, TKey, TElement>(
    this IEnumerable<TSource> source,
    Func<TSource, TKey> keySelector,
    Func<TSource, TElement> elementSelector) where TKey : notnull;

```

Donde `Func<TSource, TKey> keySelector` nos ayudará a decidir que propiedad será la clave del tipo anónimo de entrada `TSource` y `Func<TSource, TElement> elementSelector`) hará el proceso análogo para el valor asociado a la clave. Quedando la función anterior.

```
private static Dictionary<Ciudad, List<Empleado>> EmpleadosVentasPorCiudad() =>
Empleados.DepartamentoDeVentas
    .GroupBy(e => e.Ciudad,
        (c, g) => new { Ciudad = c, Empleados = g })
    .ToDictionary(eXc => eXc.Ciudad, eXc => new List<Empleado>(eXc.Empleados));
```

Por último, queremos poder filtrar los empleados de ventas que queremos agrupar por ciudad por cualquier predicado. Una vez más podremos aplicar programación funcional y pasar la estrategia de filtrado como parámetro.

```
// En este caso no necesitamos un tipo concreto, podemos usar un diccionario.
// y mejoramos la funcionalidad anterior para que admita un filtro mediante una HOF.
private static Dictionary<Ciudad, List<Empleado>>
EmpleadosVentasPorCiudad(Func<Empleado, bool> filtroEmpleado) =>
Empleados.DepartamentoDeVentas
    .Where(filtroEmpleado)
    .GroupBy(e => e.Ciudad,
        (c, g) => new { Ciudad = c, Empleados = g })
    .ToDictionary(eXc => eXc.Ciudad, eXc => new List<Empleado>(eXc.Empleados));

private static string ATexto(Dictionary<Ciudad, List<Empleado>> empleadosXCiudad)
{
    StringBuilder salida = new StringBuilder();
    foreach (var (cuidad, empleados) in empleadosXCiudad)
    {
        salida.Append($"{cuidad}:\n");
        foreach (Empleado e in empleados.OrderBy(e => e.Edad))
            salida.Append($"{e}\n");
    }
    return salida.ToString();
}

static void Main()
{
    var empleadosVentasPorCiudadDeMasDe40Años =
    EmpleadosVentasPorCiudad(e => e.Edad > 40);
    Console.WriteLine(ATexto(empleadosVentasPorCiudadDeMasDe40Años));
}
```



# Zip

Se trata de una [operación típica en los lenguajes funcionales](#) al permitirnos manejar el concepto de tupla y que por tanto podemos [encontrar también en C#](#) ya que también nos los permite.

Básicamente realiza una correspondencia unívoca entre los elementos de **dos** o más secuencias iterables, produciendo una **nueva secuencia de tuplas** resultado de dicha correspondencia. Obviamente, su nombre viene de la analogía de cerrar una cremallera.

Supongamos **Sec[A]** = [🦊, 🐟, 🍓, 🐔, 🌽] y **Sec[B]** = [🍔, 🍷, 🍟, 🍌, 🍿]

**zip(Sec[A], Sec[B])** = [(🦊, 🍔), (🐟, 🍷), (🍓, 🍟), (🐔, 🍌), (🌽, 🍿)]

**Ejemplo:** Supongamos que tenemos dos arrays. Uno con nombres de país y otro su población en millones de habitantes. Queremos mostrar una correlación entre ambos arrays por pantalla en forma de texto. Una aproximación funcional a la solución desde C# podría ser...

```
string[] paises = {"China", "India", "Estados Unidos", "Indonesia"};
int[] poblaciones = {1391, 1364, 327, 264};

string salida = string.Join("\n",
    paises.Zip(poblaciones)
        .Select(t => $"Población {t.First} {t.Second} millones."));
Console.WriteLine(salida);
```

Mostrará por pantalla...

```
Población China 1391 millones.
Población India 1364 millones.
Población Estados Unidos 327 millones.
Población Indonesia 264 millones.
```

Fíjate que la función `paises.Zip(poblaciones)` me genera una secuencia de tuplas del tipo `(string, int)` → `[("China", 1391), (India, 1364)], ... ]` y posteriormente en la función de mapeo genera una nueva secuencia de cadenas componiendo los elementos de la tupla `t`.

🚩 **Nota:** La operación inversa **Unzip** no está definida en el lenguaje porque se puede realizar fácilmente con un mapeo...

```
(string, int)[] datos = {  
    ("China", 1391), ("India", 1364), ("Estados Unidos", 327), ("Indonesia", 264)  
};  
string[] paises = datos.Select(t => t.Item1).ToArray();  
int[] poblacion = datos.Select(t => t.Item2).ToArray();
```

Si quisiéramos hacer un **Zip** de tres secuencias o más secuencias, podríamos aplicar la función varias veces. En el siguiente ejemplo vamos a combinar los valores de tres vectores en una tupla de tres elementos correspondientes a coordenadas en el espacio.

```
int[] v1 = { 1, 2, 3, 4, 5 };  
int[] v2 = { 6, 7, 8, 9, 10 };  
int[] v3 = { 11, 12, 13, 14, 15 };  
  
(int x, int y, int z)[] agrupacion = v1  
    // Combinamos v1 con v2  
    .Zip(v2, (d1, d2) => (d1, d2))  
    // Combinamos las tuplas resultantes con v3  
    .Zip(v3, (t12, d3) => (t12.d1, t12.d2, d3))  
    .ToArray();  
  
foreach (var (x, y, z) in agrupacion)  
    Console.WriteLine($"({x}, {y}, {z})");
```

## FlatMap o Aplanado

La operación de aplanado es otra de las funciones típicas de la programación funcional que vamos a encontrar en todos los lenguajes funcionales y otros como JavaScript, Java o C#. A este tipo de HOF se le conoce como **mónada** porque cumple ciertas operaciones matemáticas en la que no vamos a entrar por estar fuera del tema, pero que si has visto la **definición formal** del enlace anterior seguramente no te hayas enterado de nada si no tienes una profunda base matemática de la teoría de categorías.

Pero siguiendo la analogía que hemos usado en el tema, podemos definirla como:

**flatMap = Sec[A] x (A → Sec[B]) → Sec[B]**

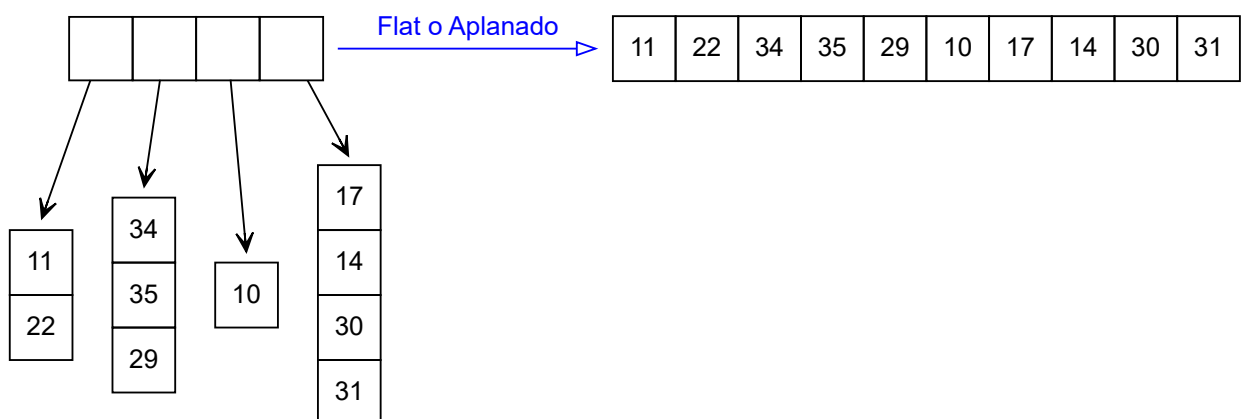
Donde dada una **Sec[A]** le vamos a aplicar una función **A → Sec[B]** donde para cada elemento de **A** produce una secuencia de **B** **Sec[B]** y el resultado será una **Sec[B]** resultado de **unir** todas los **Sec[B]** producidos por cada **A**.

**¿Sigue sin quedar claro?** No pasa nada, vamos a verlo a través de un ejemplo sencillo, pero esta vez en lugar de usar iconos, vamos a usar enteros y programa sencillo en C#.

En primer lugar comentaremos que la función equivalente en C# que más se aproxima a esta definición básica de **FlatMap** es **SelectMany** y en concreto la sobrecarga siguiente:

```
public static IEnumerable<B> SelectMany<A, B> (  
    this IEnumerable<TSource> source,  
    Func<A, IEnumerable<B>> selector);
```

Supongamos la siguiente de representación donde tenemos el típico array de arrays o tabla dentada. En el fondo, podemos considerarlo como una secuencia de secuencias (sub-secuencias) de enteros:



Si te fijas en el diagrama podemos ver que la operación de flat consiste en generar una nueva secuencia con datos de las sub-secuencias de entrada. El efecto es como si estuviéramos '*aplanando*' la tabla dentada.

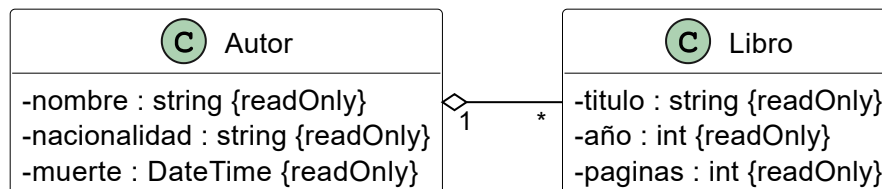
```
int[][] jagged = {
    new int[]{11, 22},
    new int[]{34, 25, 29},
    new int[]{10},
    new int[]{17, 14, 30, 31}
};

// SelectMany = int[][] → (int[] → int[]) → int[]
int[] flat = jagged.SelectMany(v => v).ToArray();

Console.WriteLine(string.Join(", ", flat));
// Mostrará por pantalla: 11, 22, 34, 25, 29, 10, 17, 14, 30, 31
```

Vemos un ejemplo más ilustrativo de este tipo de operación con tipos más elaborados.

Supongamos que la siguiente típica relación de cardinalidad uno a varios entre **Autor** y **Libro**. Donde suponemos que un autor ha escrito uno o varios libros y que modelizaremos de la siguiente manera...



🔴 **Nota:** Para simplificar, puedes descargar el código de las clases y la carga inicial de datos del siguiente código [DatosAutoresTema\\_11\\_4.cs](#).

El programa generará una **secuencia de autores** que mostrará al ejecutarse y que contendrá los siguientes autores:

```
Nombre: William Shakespeare
Nacionalidad: Inglesa
Muerte: 03/05/1616
Libros:
    Titulo: Macbeth                Año: 1623  Páginas: 128
    Titulo: La tempestad           Año: 1611  Páginas: 160
```

Nombre: Miguel de Cervantes

Nacionalidad: Española

Muerte: 22/06/1616

Libros:

Titulo: Don Quijote de la Mancha	Año: 1605	Páginas: 1376
Titulo: La Galatea	Año: 1585	Páginas: 664
Titulo: Los trabajos de Persiles y Sigismunda	Año: 1617	Páginas: 888
Titulo: Novelas ejemplares	Año: 1613	Páginas: 1160

Nombre: Fernando de Rojas

Nacionalidad: Española

Muerte: 07/02/1541

Libros:

Titulo: La Celestina	Año: 1500	Páginas: 160
----------------------	-----------	--------------

Con las funciones que hemos visto hasta el momento podríamos hacer consultas sobre los autores. Por ejemplo, para ver aquellos autores que han publicado algún libro en el siglo XVII podríamos hacer...

```
IEnumerable<Autor> autoresConLibrosPublicadosDuranteSigloXVII =  
Datos.Autores.Where(a => a.Libros.Where(l => l.Año > 1600 && l.Año < 1701).Any());
```

Pero, si quisiéramos obtener los libros publicados en el siglo XVII a partir de la secuencia de autores. Ahora tendríamos un caso de uso de **SelectMany** (Flat Map):

```
// IEnumerable<Autor> x (Autor -> IEnumerable<Libro>) -> IEnumerable<Libro>  
IEnumerable<Libro> librosSigloXVII =  
Datos.Autores.SelectMany(a => a.Libros.Where(l => l.Año > 1600 && l.Año < 1701));
```

Incluso podríamos **generar una correlación** entre **Libros** y **Autor** proyectando cada libro que cumple la condición a una nueva secuencia de objetos anónimos o DTO que tuviese datos del autor del libro.

Por ejemplo, si quisiésemos obtener una vista libro, autor y número de páginas de aquellos libros que tienen menos de mil páginas y crear una secuencia tipada con un DTO podríamos hacer por ejemplo...

```
record LibroDto(string Libro, string Autor, int Paginas)
{
    // IEnumerable<Autor> x (Autor → IEnumerable<LibroDto>) → IEnumerable<LibroDto>
    public static IEnumerable<LibroDto> LibrosDeMenosdeMilPaginas() =>
        Datos.Autores.SelectMany(a => a.Libros.Where(l => l.Paginas < 1000)
            .Select(l => new LibroDto(
                Libro: l.Titulo, Autor: a.Nombre, Paginas: l.Paginas)));
    // Una vez tenemos la secuencia de salida con los objetos anónimos de la proyección.
    // Podemos mostrarla en forma de tabla de la siguiente forma ...
    public static string ToString(IEnumerable<LibroDto> datos, string separadorDato) =>
        datos.Aggregate(separadorDato, (salida, tupla) =>
            salida += $"Libro: {tupla.Libro}\nAutor: {tupla.Autor}\n" +
                $"Paginas: {tupla.Paginas}{separadorDato}");
}
```

Posteriormente si ejecutamos ...

```
Console.WriteLine(LibroDto.ToString(LibroDto.LibrosDeMenosdeMilPaginas(), SeparadorDato));
```

Generando la siguiente salida por consola...

```
Libro: Macbeth
Autor: William Shakespeare
Paginas: 128
-----
Libro: La tempestad
Autor: William Shakespeare
Paginas: 160
-----
Libro: La Galatea
Autor: Miguel de Cervantes
Paginas: 664
-----
Libro: Los trabajos de Persiles y Sigismunda
Autor: Miguel de Cervantes
Paginas: 888
-----
Libro: La Celestina
Autor: Fernando de Rojas
Paginas: 160
```

## 🎓 Caso de estudio:

Vamos a realizar varias consultas sobre el código del ejemplo anterior donde tenemos libros por autor. **Intenta pensarlas sin mirar la propuesta de solución.**

### 1. Autores con **más de un** libro.

```
var snapshot = Datos.Autores.Where(a => a.Libros.Count() > 1);
Console.WriteLine(snapshot.Aggregate(SeparadorDato,
    (salida, a) => salida += $"{a}{SeparadorDato}"));
```

### 2. Total de libros escritos por escritores Españoles.

```
var totalLibros = Datos.Autores.Where(a => a.Nacionalidad == "Española")
    .Aggregate(0, (t, a) => t += a.Libros.Count());
// Otra opción más legible utilizando SelectMany sería...
var totalLibros = Datos.Autores.Where(a => a.Nacionalidad == "Española")
    .SelectMany(a => a.Libros)
    .Count();
Console.WriteLine($"Hay {totalLibros} escritos por españoles");
```

### 3. Muestra el **nombre del autor** y **año de la muerte** agrupando los autores por siglo en el que murieron en orden ascendente de siglo.

```
var snapShot =
    Datos.Autores.Select(a => new { Autor = a.Nombre, AñoMuerte = a.Muerte.Year })
        .OrderBy(a => a.AñoMuerte)
        .GroupBy(a => a.AñoMuerte / 100 + 1,
            (siglo, autores) => new { Siglo = siglo, Autores = autores });

Console.WriteLine(snapshot.Aggregate(SeparadorDato, (salida, aXs) =>
    salida += $"Siglo: {aXs.Siglo}\n"
    + $"{aXs.Autores.Aggregate("",
        (t, a) => t += $"{a.Autor} muerto en {a.AñoMuerte}\n"
    )}" + $"{SeparadorDato}"
    ));
```

🔴 **Nota:** Aunque en este caso hemos utilizado programación funcional para mostrar el resultado. Este sería el típico caso donde **por legibilidad y claridad de código** sería más interesante usar el esquema tradicional de componer un **StringBuilder** con bucles.

#### 4. Numero **total de páginas escritas** por **William Shakespeare**.

```
var totalPaginas =  
Datos.Autores.Where(a => a.Nombre == "William Shakespeare")  
    .SelectMany(a => a.Libros.Select(l => l.Paginas)).Sum();  
  
Console.WriteLine($"William Shakespeare escribió {totalPaginas}");
```