

Programa Académico Desarrollo de Software Nivel II

C# II

Prof. Gabriel Roa


Tema III – Funciones Lambda

Contenido

I – FUNCIONES LAMBDA	2
I.1 – DEFINICIONES BÁSICAS	3
I.2 – SINTAXIS	4
I.3 – USOS COMUNES	8
I.3.1 – ENUMERABLE	8
I.4 – EJERCICIOS PROPUESTOS	18
II – REFERENCIAS BIBLIOGRÁFICAS	21

I – Funciones Lambda

Hasta ahora hemos estudiado la forma de crear funciones tradicionales a través del uso de la estructura habitual, es decir, *public void nombreFuncion(parámetros)...* Sin embargo, esta no es la única manera de escribir funciones que ofrece C#. A veces, hay funciones que por sus características resultan ser muy pequeñas, de tal manera que escribir una función tradicional anexa muchísimo más código del necesario. Consideremos, por ejemplo, una función `ElevarAlCuadrado(int numero)`, que recibe un número entero y devuelve su cuadrado:

A screenshot of a code editor with a dark background and a purple border. The code is written in C# and defines a public method named ElevarAlCuadrado that takes an integer parameter and returns its square. The code is as follows:

```
public int ElevarAlCuadrado(int numero){  
    return numero * numero;  
}
```

Fig. 105 – Función para elevar un número al cuadrado

Esta función utiliza *tres* líneas de código para efectuar una operación sumamente sencilla como lo puede ser elevar un número entero al cuadrado. C# ofrece, por otro lado, una sintaxis mucho más corta, legible y fácil de utilizar para definir funciones: la sintaxis de las funciones o expresiones Lambda.

I.1 – Definiciones Básicas

Una función o expresión lambda es una función definida a través del uso de la siguiente sintaxis:

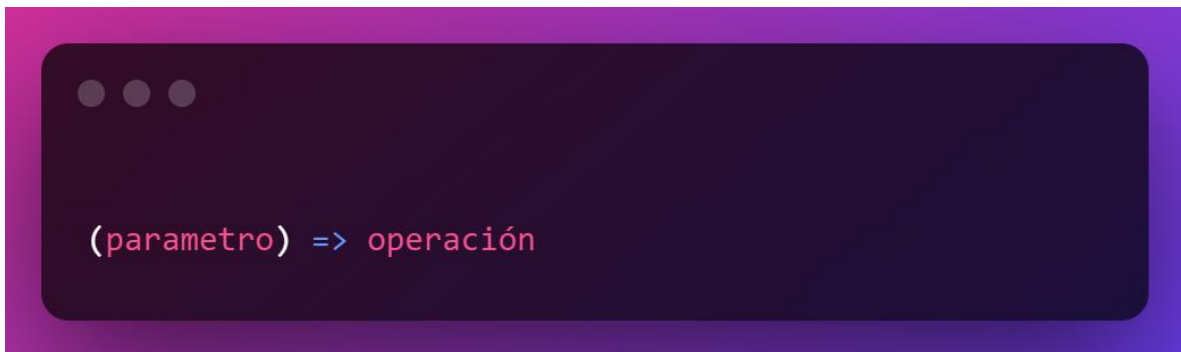


Fig. 106 – Función Lambda

Estas funciones lambda tienen la característica de que pueden o no ser anónimas, entendiendo una función anónima como una función que se define y declara explícitamente sin necesidad de indicar un nombre para la misma. Las expresiones lambda son aquellas que definen una sola operación o una sola línea de código, mientras que una función lambda define un conjunto de expresiones, pero nunca más de dos o tres al mismo tiempo; por lo tanto, se puede decir que la diferencia entre una expresión lambda y una función lambda radica en la cantidad de líneas de código que esta ejecuta en su interior.

Finalmente, entendemos como colección a cualquier conjunto de elementos agrupados bajo una misma variable. Una colección bien puede ser

un arreglo primitivo, un List, Queue, Stack, e incluso pueden ser resultados de efectuar operaciones con una base de datos – para efectos prácticos de C#, entendemos que todas estas colecciones implementan la interfaz IEnumerable, lo que permite que sobre ellas se utilicen los métodos de System.Linq.

I.2 – Sintaxis

Una expresión lambda anónima luce como se describe en la figura 106. El parámetro se indica a la izquierda de la expresión, seguido de una flecha (=>) y, a continuación, la operación a efectuar dentro de la función. El valor resultante de esta operación es el mismo valor que será retornado al ejecutarla. Así, una expresión lambda anónima análoga a la función definida en la figura 105 sería la siguiente:



```
(numero) => numero * numero;
```

Fig. 107 – Expresión Lambda para elevar un número al cuadrado

Una expresión lambda puede recibir tantos parámetros como sean necesarios, y para ello, sólo hace falta separarlos con una coma del lado

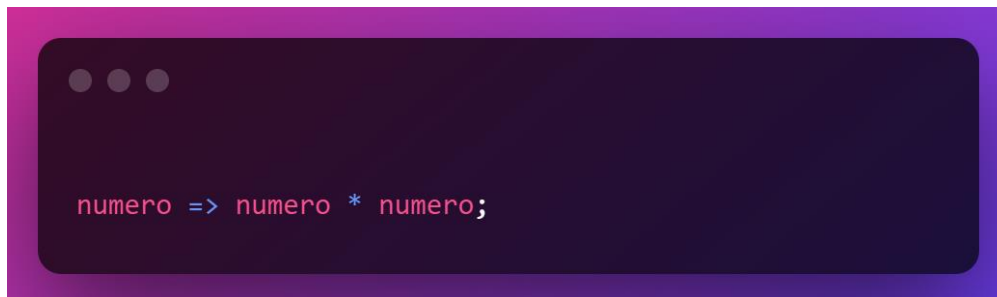
izquierdo de la flecha. Por ejemplo, esta expresión lambda anónima multiplica dos números:



```
(nro1, nro2) => nro1 * nro2;
```

Fig. 108 – Expresión Lambda para multiplicar dos números

Más aún, en caso de tratarse de una expresión con un solo parámetro, se puede prescindir de los paréntesis del lado izquierdo de la flecha. Así, la expresión de la figura 107 podría reescribirse de la siguiente manera:



```
numero => numero * numero;
```

Fig. 109 – Expresión Lambda anónima de un solo parámetro sin paréntesis

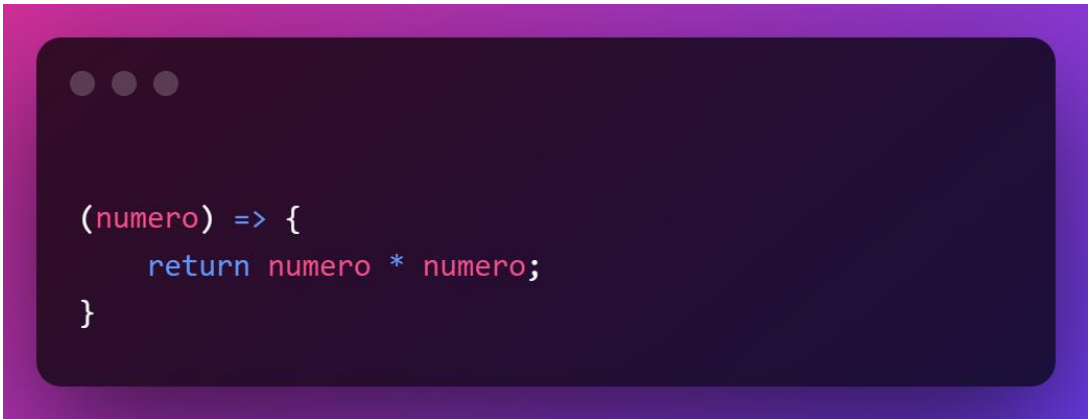
Tanto una función como una expresión Lambda pueden definirse sin necesidad de parámetros. Considerando la siguiente expresión Lambda que imprime un salto de línea en consola, vemos cómo se utilizan unos paréntesis vacíos para denotar que la expresión no recibe parámetros:



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

Fig. 110 – Salto de línea en una Expresión Lambda

Una función Lambda define un conjunto de instrucciones u operaciones a ejecutar, y para ello, encierra estas operaciones entre llaves después de la flecha, de manera similar a esto:



```
(numero) => {  
    return numero * numero;  
}
```

Fig. 111 – Función Lambda

Una función lambda, a diferencia de una expresión, debe especificar explícitamente cuál será el valor de retorno, o en caso contrario, la función no retornará valor alguno.

Ahora bien, las funciones Lambda no necesariamente deben ser anónimas. Podemos nombrarlas a través del uso de la palabra reservada `Func<>`. Esta palabra reservada recibirá, entre los `<>`, los tipos de datos de los parámetros y el tipo de dato del valor de retorno de la función, en ese orden estricto. Por ejemplo, aquí tenemos la expresión lambda de la figura 109 nombrada `ElevarAlCuadrado`:

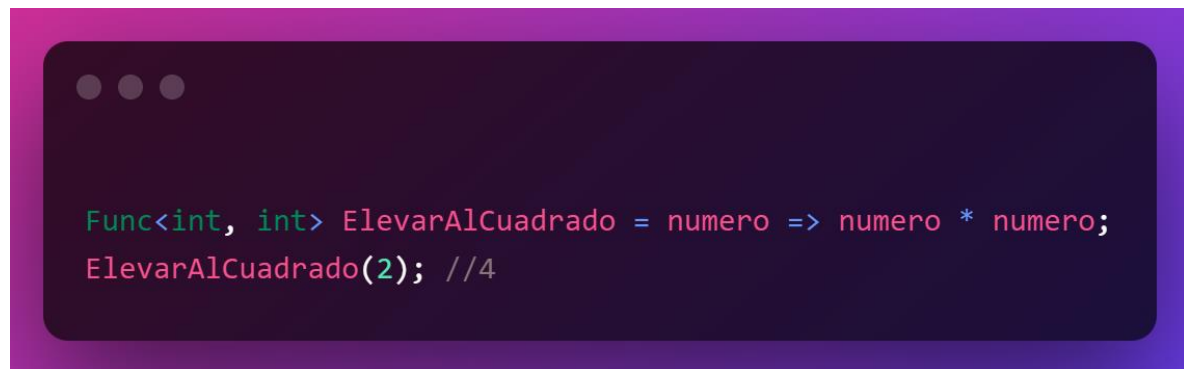
A screenshot of a code editor with a dark background and a purple border. It shows a named lambda function in Kotlin. The code is: `Func<int, int> ElevarAlCuadrado = numero => numero * numero;` followed by a call `ElevarAlCuadrado(2); //4` on the next line. The text is color-coded: `Func` is blue, `<int, int>` is green, `ElevarAlCuadrado` is red, `=` is blue, `numero` is pink, `=>` is blue, `numero` is pink, `*` is blue, `numero` is pink, `;` is blue, `ElevarAlCuadrado` is red, `(2)` is green, `;` is blue, and `//4` is light blue.

Fig. 112 – Expresión Lambda con nombre y cómo hacer llamado a la misma.

Del mismo modo, es posible utilizar la palabra reservada `var` para que el compilador detecte en tiempo de compilación el tipo de función a utilizar, lo que facilita mucho más la definición de la función:

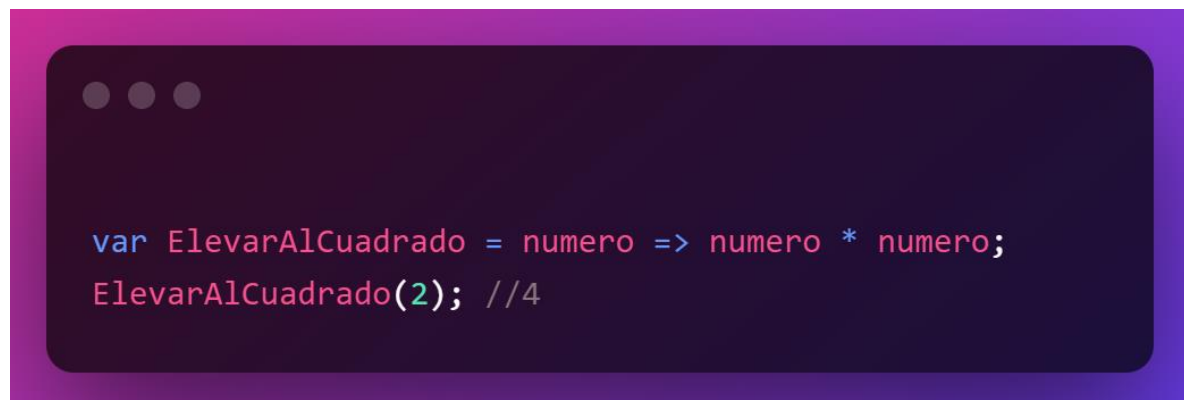
A screenshot of a code editor with a dark background and a purple border. It shows a lambda function declared with the `var` keyword in Kotlin. The code is: `var ElevarAlCuadrado = numero => numero * numero;` followed by a call `ElevarAlCuadrado(2); //4` on the next line. The text is color-coded: `var` is blue, `ElevarAlCuadrado` is red, `=` is blue, `numero` is pink, `=>` is blue, `numero` is pink, `*` is blue, `numero` is pink, `;` is blue, `ElevarAlCuadrado` is red, `(2)` is green, `;` is blue, and `//4` is light blue.

Fig. 113 – Expresión Lambda declarada con la palabra reservada var.

I.3 – Usos comunes

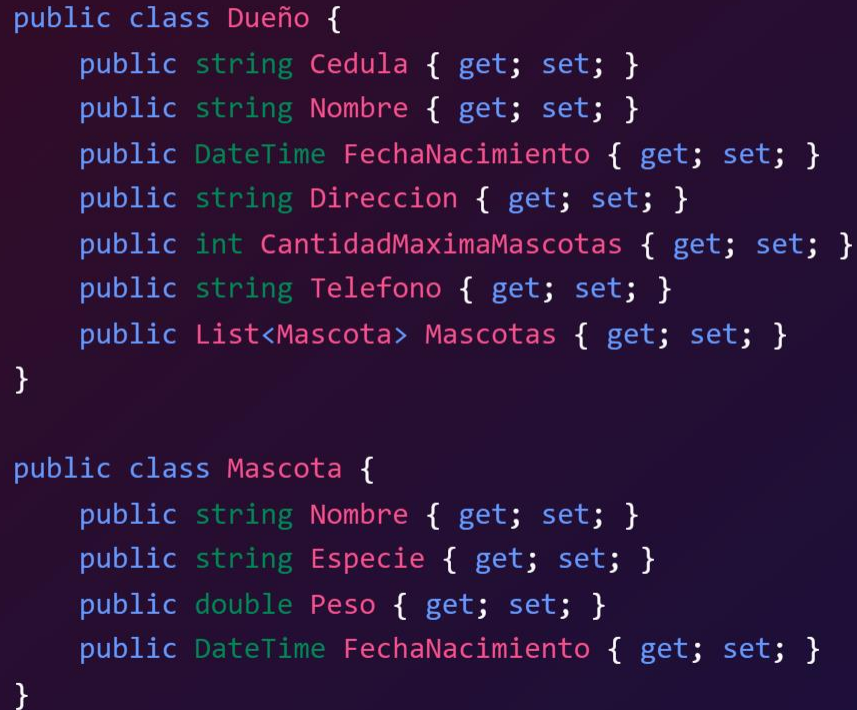
Las funciones Lambda son ampliamente utilizadas para definir funciones cortas, sencillas y de alcance local, cuando se está operando dentro de un método. No obstante, su uso principal es con el conjunto de clases que implementan la interfaz `IEnumerable`, que nos ofrece una sintaxis poderosísima basada en métodos para ejecutar sentencias sobre conjuntos de datos bien definidos.

I.3.1 – Enumerable

Como interfaz, `IEnumerable` nos define una serie de métodos que, en caso de ser aplicadas, nos permiten hacer uso de la clase `Enumerable`, perteneciente a `System.Linq`. Muchas clases implementan esta interfaz, y entre ellas, están `Queue` (Colas), `Stack` (Pilas), `List` (Listas) y muchas más que estudiaremos más adelante.

Algunos de los métodos que ofrece y son los más comúnmente utilizados son los métodos `Average`, `Contains`, `Count`, `First`, `GroupBy`, `Join`, `Last`, `Max`, `Min`, `OrderBy`, `Single`, `Select`, `Sum`, `TakeWhile`, `Where`, entre muchos otros. Una lista completa puede conseguirse en la documentación oficial de C#, en la cual se explican uno a uno estos métodos. Para efectos de este curso, explicaremos a través del uso de `List`, los métodos `Count`, `First`, `Single`, `OrderBy`, `Where`, `Sum` y `Average`, además de explicar el uso de los métodos `OrDefault`.

En todo caso, asumiremos las siguiente clases Dueño y Mascota para ilustrar los métodos:



```
public class Dueño {  
    public string Cedula { get; set; }  
    public string Nombre { get; set; }  
    public DateTime FechaNacimiento { get; set; }  
    public string Direccion { get; set; }  
    public int CantidadMaximaMascotas { get; set; }  
    public string Telefono { get; set; }  
    public List<Mascota> Mascotas { get; set; }  
}  
  
public class Mascota {  
    public string Nombre { get; set; }  
    public string Especie { get; set; }  
    public double Peso { get; set; }  
    public DateTime FechaNacimiento { get; set; }  
}
```

Fig. 114 – Clases Dueño y Mascota

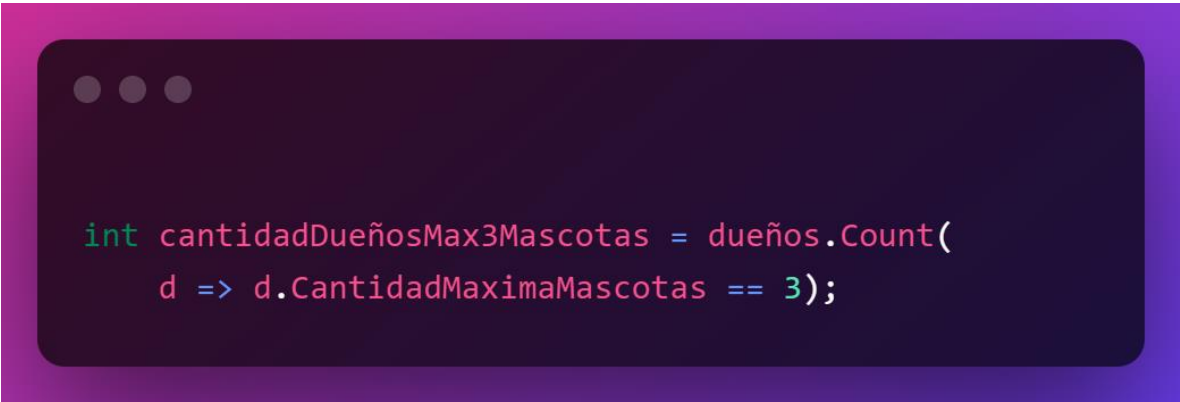
Y asumiremos la existencia de un List<Dueño> llamado dueños, el cual contendrá la información de un conjunto de dueños y sus respectivas mascotas, además de importar el paquete Linq en nuestro código a través de using System.Linq.

I.3.1.1 – Count

El método Count funciona de dos maneras diferentes: o bien puede devolver el conteo de todos los elementos contenidos en nuestra colección, o bien puede devolver el conteo únicamente de algunos elementos que cumplan una cierta condición. Para ello, debemos realizar el llamado al método Count con un parámetro de función Lambda, la cual recibirá como parámetro un dueño y retornará un valor booleano que responda la siguiente pregunta:

¿Cuáles elementos deseo contar? ¿Qué característica debe cumplir un elemento para que este pertenezca a mi conteo?

Consideremos un ejemplo de conteo, donde queremos contar todos los dueños cuya cantidad máxima de mascotas sea exactamente de 3.

The image shows a screenshot of a code editor with a dark background and a purple border. It contains a single line of C# code that uses the Count method on a collection named 'dueños'. The lambda expression 'd => d.CantidadMaximaMascotas == 3' is used as the argument for Count, indicating that the method will return the number of elements in 'dueños' where the 'CantidadMaximaMascotas' property is equal to 3.

```
int cantidadDueñosMax3Mascotas = dueños.Count(  
    d => d.CantidadMaximaMascotas == 3);
```

Fig. 115 – Método Count.

La expresión Lambda dicta que, en este caso, deseo contar los dueños *d*, siempre y cuando su cantidad máxima de mascotas sea igual a 3. Esta expresión devolverá un valor booleano que se ejecutará para cada uno de los dueños pertenecientes a la colección, y en caso de que su valor sea true, se contará el mismo.

I.3.1.2 – First

Este método nos devuelve el *primer* elemento de una colección que cumpla con un criterio dado. Este criterio se establece a través de una función Lambda. Por ejemplo, si queremos obtener el primer dueño que tenga dos mascotas, podemos utilizar el siguiente código:

A screenshot of a code editor with a dark background and a purple border. It shows a single line of C# code: `Dueño dosMascotas = dueños.First(d => d.Mascotas.Count == 2);`. The code is color-coded: 'Dueño' is pink, 'dosMascotas' is blue, 'dueños' is light blue, 'First' is light blue, 'd' is light blue, 'd.Mascotas' is light blue, 'Count' is light blue, '== 2' is green, and the semicolon is light blue.

```
Dueño dosMascotas = dueños.First(d => d.Mascotas.Count == 2);
```

Fig. 116 – Método First

Este método funciona análogo al método Last, por lo que si quisiéramos obtener *el último* dueño que tenga sólo dos mascotas, podemos utilizar el siguiente código:



```
Dueño dosMascotas = dueños.Last(d => d.Mascotas.Count == 2);
```

Fig. 117 – Método Last

I.3.1.3 – Single

El método Single nos devuelve un único elemento de una colección que cumpla una condición dada. Es análogo al First, exceptuando por dos condiciones que debemos evaluar para ver si vamos a utilizar First o Single:

- 1) ¿Hay múltiples elementos que pueden cumplir mi condición? Si la respuesta es sí, debo usar el First. En caso contrario, usar Single.
- 2) ¿Necesito garantizar tener un solo elemento como valor de retorno después de la condición? Si la respuesta es sí, debo usar Single. En caso contrario, debo usar el First.

Al final ambos funcionan de manera muy similar, así que la decisión de usar un método contra el otro radicará principalmente en un razonamiento estético de legibilidad, si yo necesito tener un solo elemento, debo usar Single, y si puedo tener varios elementos y quiero sólo el primero, debo usar First.

Como tal, si quisiera buscar a un dueño cuya cédula de identidad sea 1.234.567, debería utilizar el siguiente código:

```
Dueño dosMascotas = dueños.Single(d => d.Cedula == "1234567");
```

Fig. 118 – Método Single

I.3.1.4 – OrderBy

Este método realiza el ordenamiento de una colección de elementos de forma ascendente dada una propiedad en particular que haya elegido, y la elección de la propiedad se realiza a través de una función lambda que lo único que debe hacer es retornar una propiedad de la clase en particular.

Si quisiéramos ordenar los dueños por fecha de nacimiento, tendríamos que usar el siguiente fragmento de código:

```
var dueñosPorFechaNacimiento = dueños.OrderBy(d => d.FechaNacimiento);
```

Fig. 119 – Método OrderBy

Del mismo modo se nos ofrece el método `OrderByDescending`, que como su nombre lo dice, realiza el ordenamiento de forma descendiente y es análogo al `OrderByAscending`.

I.3.1.5 – Where

Similar a como funciona el `Where` en sentencias SQL, en este caso utilizar `Where` nos devuelve **todos** los elementos de una colección que cumplan con una cierta condición, lo que hace que este método funcione como un filtrado. La condición se establece a través de una función lambda.

Si quisiéramos obtener todos los dueños cuyo nombre comienza por J, podríamos hacer el siguiente método:



```
var dueñosPorJ = dueños.Where(d => d.Nombre.StartsWith("J"));
```

Fig. 120 – Método Where

I.3.1.6 – Sum

Este método devuelve el acumulado de una propiedad en una colección de datos. La propiedad se selecciona a través del uso de una función Lambda, y el valor de retorno depende del tipo de dato que estamos acumulando.

Por ejemplo, si quisiéramos ver cuántas mascotas en total pueden tener los dueños que tenemos registrados, podríamos hacer Sum de la siguiente manera:

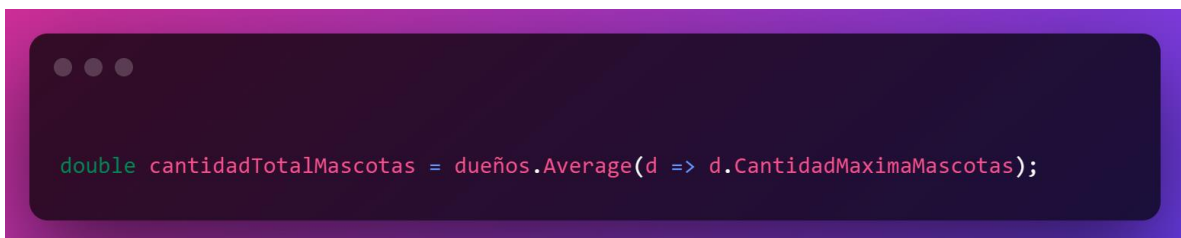
A code editor window with a dark background and a purple border. It contains a single line of C# code: `int cantidadTotalMascotas = dueños.Sum(d => d.CantidadMaximaMascotas);`. The code is color-coded: `int` is green, `cantidadTotalMascotas` is red, `=` is blue, `dueños` is red, `.Sum` is blue, `(d =>` is green, `d.CantidadMaximaMascotas)` is red, and `;` is blue. There are three small grey circles in the top left corner of the editor.

```
int cantidadTotalMascotas = dueños.Sum(d => d.CantidadMaximaMascotas);
```

Fig. 121 – Método Sum

I.3.1.7 – Average

De forma análoga a como podemos utilizar el Sum, podemos usar el método Average para determinar el promedio de un valor dado en nuestro conjunto de datos. Si quisiéramos saber, por ejemplo, cuántas mascotas máximo pueden tener los dueños en promedio, podemos hacer Average de la siguiente manera:

A code editor window with a dark background and a purple border. It contains a single line of C# code: `double cantidadTotalMascotas = dueños.Average(d => d.CantidadMaximaMascotas);`. The code is color-coded: `double` is green, `cantidadTotalMascotas` is red, `=` is blue, `dueños` is red, `.Average` is blue, `(d =>` is green, `d.CantidadMaximaMascotas)` is red, and `;` is blue. There are three small grey circles in the top left corner of the editor.

```
double cantidadTotalMascotas = dueños.Average(d => d.CantidadMaximaMascotas);
```

Fig. 122 – Método Average

I.3.1.8 – OrDefault

Hay algunos métodos de los que hemos estudiado hasta los momentos que ofrecen una alternativa llamada OrDefault. Más precisamente, los métodos Single, First y Last ofrecen unas versiones llamadas SingleOrDefault, FirstOrDefault y LastOrDefault.

Su principal ventaja es que, a diferencia de sus análogos sin el OrDefault, en caso de que ocurra algún tipo de excepción en su ejecución, estos devolverán el valor *default* en lugar de arrojar la excepción en sí, lo que hace que una búsqueda sin tener garantía de que existan elementos o una búsqueda que puede devolver más de un elemento no pueda arrojar excepciones, además de brindarnos el valor default con el que es mucho más sencillo validar en caso de inconvenientes.

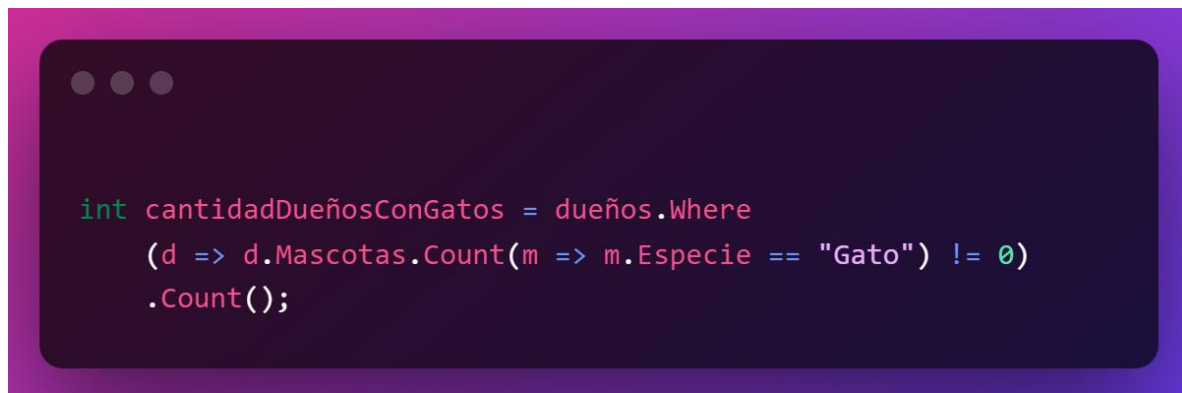
I.3.1.9 – Combinación de métodos

Todos los métodos que hemos listado hasta ahora (y la mayoría de los métodos que ofrece Enumerable en general) pueden mezclarse y combinarse para escribir sentencias similares a SQL sin ningún tipo de problema, además que pueden recibir funciones lambda tan complejas o tan sencillas como sea necesario para resolver los problemas dados.

Más aún, la lógica es similar para cualquier tipo de colección de datos que nos ofrezca C#, por lo que haber aprendido todo esto a través del uso de

Listas también nos ayudará muchísimo para escribir sentencias que manipulen datos de una Base de Datos con SQL Server, como lo haremos a continuación.

Por ejemplo, si quisiéramos saber cuántos dueños tienen gatos, podríamos hacer la siguiente sentencia:

A screenshot of a code editor with a dark background and a purple border. The code is written in C# and uses LINQ to filter owners based on the number of cats they have. The code is as follows:

```
int cantidadDueñosConGatos = dueños.Where
    (d => d.Mascotas.Count(m => m.Especie == "Gato") != 0)
    .Count();
```

Fig. 123 – Cuántos dueños tienen gatos

En este caso estamos combinando dos funciones LINQ en simultáneo: estamos buscando a todos los dueños, cuyas mascotas cumplan con una condición: que al contar las mascotas de especie Gato, estas tengan una cantidad distinta de cero, y tras esto, procedemos a contar los dueños que cumplan con esta condición.

Si quisiéramos, por ejemplo, obtener un listado ordenado por fecha de nacimiento (de más viejo a más joven), en el cual estén todos los dueños cuyo nombre sea José, podríamos hacer esta sentencia:

```
var dueñosJoseOrdenados = dueños.Where(d => d.Nombre.Contains("José")).  
    OrderBy(d => d.FechaNacimiento);
```

Fig. 124 – Todos los José ordenados por fecha de nacimiento

I.4 – Ejercicios propuestos

1) Escriba funciones Lambda que resuelvan los siguientes problemas:

- a. Encuentre los coeficientes de una ecuación cuadrática.
- b. Dada una lista de números, determine cuál es el menor.
- c. Devuelva el primer caracter de una cadena de caracteres.
- d. Sume dos matrices.
- e. Sume dos números imaginarios.
- f. Encuentre el enésimo término de la sucesión de Fibonacci.
- g. Calcule el factorial de un número.
- h. Concatene una cadena de caracteres con un número entero.
- i. Imprima un mensaje en consola.
- j. Imprima una cuadrícula de nxn en consola, dado un número n.
- k. Calcule el área de una circunferencia.
- l. Calcule el área de un triángulo rectángulo.
- m. Encuentre el promedio de una lista de números enteros.
- n. Realice un log de error utilizando la clase ErrorLog.

2) Considerando las clases Dueño y Mascota anteriormente definidas, escriba combinaciones de los métodos ofrecidos por Enumerable para obtener:

- a. Cuántos dueños hay con 3 mascotas exactamente.
- b. Mascotas cuyos dueños poseen líneas telefónicas Movilnet.
- c. Dueños que tienen más de 45 años y menos de 3 perros, ordenados por edad de más jóvenes a más viejos.
- d. Todos los Gatos registrados.
- e. Todos los Perros de menos de 5 kilos de peso, cuyos dueños viven en Barquisimeto.
- f. Dueños que poseen más mascotas de las que realmente deberían tener.
- g. Mascotas de más de 5 kilos de peso, cuyos nombres comiencen por la letra J, y cuyos tengan nombres que comiencen por la letra J también.
- h. El primer dueño que haya nacido el 13 de marzo del 1984.
- i. Cuántas mascotas existen en total.
- j. Cuántos dueños tienen gatos.
- k. Cuántos gatos pesan más de 3 kilos y son de dueños menores de edad.
- l. Todos los dueños, ordenados por su cantidad de mascotas y luego por su fecha de nacimiento.
- m. Todas las mascotas, ordenadas por su peso y luego por su fecha de nacimiento.

- n. La mascota más joven de todas.
- o. El dueño más viejo de todos.
- p. Cuántas mascotas en total tienen los dueños de menos de 18 años.
- q. El promedio de edad de los dueños.
- r. El promedio de edad de las mascotas.
- s. El peso total combinado de todas las mascotas cuyos dueños tienen un nombre que termina con una vocal.

II – Referencias Bibliográficas

Aguinaga, Á. (07 de 01 de 2022). *¿Qué es un ORM y cuándo emplearlo?*
Obtenido de Cipsa: <https://cipsa.net/que-es-un-orm-y-cuando-emplearlo/>

Avast. (01 de 07 de 2022). *¿Qué es la inyección de SQL? | Explicación y protección | Avast*. Obtenido de Avast.com: <https://www.avast.com/es-es/c-sql-injection>

ConnectionStrings. (07 de 01 de 2022). *SQL Server 2012 Data Types Reference - ConnectionStrings.com*. Obtenido de ConnectionStrings: <https://www.connectionstrings.com/sql-server-2012-data-types-reference/>

Deloitte. (07 de 01 de 2022). *¿Qué es un ORM?* Obtenido de Deloitte: <https://www2.deloitte.com/es/es/pages/technology/articles/que-es-orm.html>

EntityFrameworkTutorial. (08 de 01 de 2022). *Entity Framework Tutorial*. Obtenido de Entity Framework Tutorial: <https://www.entityframeworktutorial.net/>

esic. (07 de 01 de 2022). *El ORM como herramienta eficiente de trabajo*. Obtenido de esic: <https://www.esic.edu/rethink/tecnologia/el-orm-como-herramienta-eficiente-de-trabajo>

Farrell, J. (2018). *Microsoft Visual C# 2017: An Introduction to Object-Oriented Programming*. Boston: Cengage Learning.

- Griffiths, I. (2019). *Programming C# 8.0*. Sebastopol: O'Reilly Media, Inc.
- Microsoft. (28 de 01 de 2021). *C# documentation*. Obtenido de Microsoft Docs: <https://docs.microsoft.com/en-us/dotnet/csharp/>
- Microsoft. (08 de 01 de 2022). *Migrations Overview - EF Core | Microsoft Docs*. Obtenido de Microsoft Docs: <https://docs.microsoft.com/en-us/ef/core/managing-schemas/migrations/>
- Microsoft. (08 de 01 de 2022). *Overview of Entity Framework Core - EF Core | Microsoft Docs*. Obtenido de Microsoft Docs: <https://docs.microsoft.com/en-us/ef/core/>
- Pérez, S. D. (07 de 01 de 2022). *¿Qué es Microsoft SQL Server y para qué sirve?* Obtenido de Intelequia: <https://intelequia.com/blog/post/2948/qu%C3%A9-es-microsoft-sql-server-y-para-qu%C3%A9-sirve>
- Siahaan, V. (2020). *VISUAL C# .NET FOR STUDENTS: A Project-Based Approach to Develop Desktop Applications*. Balige: BALIGE PUBLISHING.
- Siahaan, V., & Sianipar, R. (2020). *VISUAL C# .NET: A Step By Step, Project-Based Guide to Develop Desktop Applications*. Balige: BALIGE PUBLISHING.
- Tutorials Point. (12 de 01 de 2022). *Entity Framework - Fluent API*. Obtenido de Tutorials Point: https://www.tutorialspoint.com/entity_framework/entity_framework_fluent_api.htm

Villalobos, J., & Casallas, R. (s.f.). *Fundamentos de Programación. Aprendizaje activo basado en casos*. Bogotá: Universidad de los Andes - Facultad de Ingeniería.