



.Net

Teoría 10

Una introducción rápida a LINQ

LINQ

En la práctica de delegados, se pidió extender al tipo `int[]` con los métodos **Seleccionar** y **Donde**. Por ejemplo si `v` es un vector de enteros, la expresión

```
v.Donde(n => n % 2 == 1).Seleccionar(n => n * n)
```

debía devolver un nuevo vector con todos los elementos impares de `v` elevados al cuadrado.

Los delegados como parámetros aportan muchísima versatilidad.



LINQ

Si en lugar de extender `int[]` extendiésemos `IEnumerable<T>` sería aún más beneficioso porque afectaría a todas las colecciones que implementan esta interfaz. Esto se pidió como ejercicio en la práctica de genéricos.

Afortunadamente no tenemos que hacerlo
`LINQ` ya lo hace por nosotros

Veamos algunos ejemplos ...





Crear una aplicación de consola llamada LINQ



1. Abrir una terminal del sistema operativo
2. Cambiar a la carpeta `proyectosDotnet`
3. Crear la aplicación de consola `LINQ`
4. Abrir code en la carpeta `LINQ`



Codificar Program.cs de la siguiente manera y ejecutar



```
int[] vector = [1, 2, 3, 4, 5];  
IEnumerable<int> secuencia = vector.Select(n => n * 3);  
Mostrar(secuencia);
```

```
void Mostrar<T>(IEnumerable<T> secuencia)  
{  
    foreach (T elemento in secuencia)  
    {  
        Console.Write(elemento + " ");  
    }  
    Console.WriteLine();  
}
```


Copiar el código del archivo
10_RecursosParaLaTeoria.txt

```
int[] vector = [1, 2, 3, 4, 5];  
IEnumerable<int> secuencia = vector.Select(n => n * 3);  
Mostrar(secuencia);
```

`T[]` implementa la interfaz
`IEnumerable<T>`

```
void Mostrar<T>(IEnumerable<T> secuencia)  
{  
    foreach (T elemento in secuencia)  
    {  
        Console.Write(elemento + " ");  
    }  
    Console.WriteLine();  
}
```

Obtenemos los
elementos de `vector`
multiplicados por 3



3 6 9 12 15



Completar la línea que falta para que la salida por consola sea la que se indica



→ `List<string> lista = ["uno", "dos", "tres"];
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Mostrar(secuencia);`

```
void Mostrar<T>(IEnumerable<T> secuencia)
{
    foreach (T elemento in secuencia)
    {
        Console.Write(elemento + " ");
    }
    Console.WriteLine();
}
```

A black computer monitor with a silver stand. The screen displays the text "(UNO) (DOS) (TRES)" in white. A small green light is visible at the bottom right of the monitor frame.

(UNO) (DOS) (TRES)



Posible solución

→ `List<string> lista = ["uno", "dos", "tres"];
IEnumerable<string> secuencia = lista.Select(st => $"({st})".ToUpper());`

`Mostrar(secuencia);`

```
void Mostrar<T>(IEnumerable<T> secuencia)
{
    foreach (T elemento in secuencia)
    {
        Console.Write(elemento + " ");
    }
    Console.WriteLine();
}
```

(UNO) (DOS) (TRES)



Completar la línea que falta para que la salida por consola sea la que se indica



```
List<string> lista = ["uno", "dos", "tres"];
IEnumerable<string> secuencia = lista.Select(st => $"({st})".ToUpper());
Mostrar(secuencia);
→ IEnumerable<int> secuencia2 = xxxxxxxxxxxxxxxxxxxxxxxxxxxx
Mostrar(secuencia2);
```

```
void Mostrar<T>(IEnumerable<T> secuencia)
{
    foreach (T elemento in secuencia)
    {
        Console.Write(elemento + " ");
    }
    Console.WriteLine();
}
```

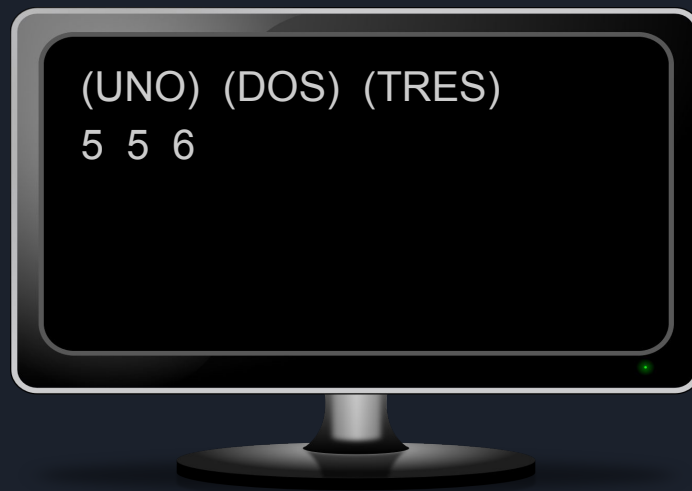
Longitud de los
strings de secuencia

(UNO) (DOS) (TRES)
5 5 6

Posible solución

```
List<string> lista = ["uno", "dos", "tres"];  
IEnumerable<string> secuencia = lista.Select(st => $"({st})".ToUpper());  
Mostrar(secuencia);  
IEnumerable<int> secuencia2 = secuencia.Select(st => st.Length);  
Mostrar(secuencia2);
```


```
void Mostrar<T>(IEnumerable<T> secuencia)  
{  
    foreach (T elemento in secuencia)  
    {  
        Console.Write(elemento + " ");  
    }  
    Console.WriteLine();  
}
```



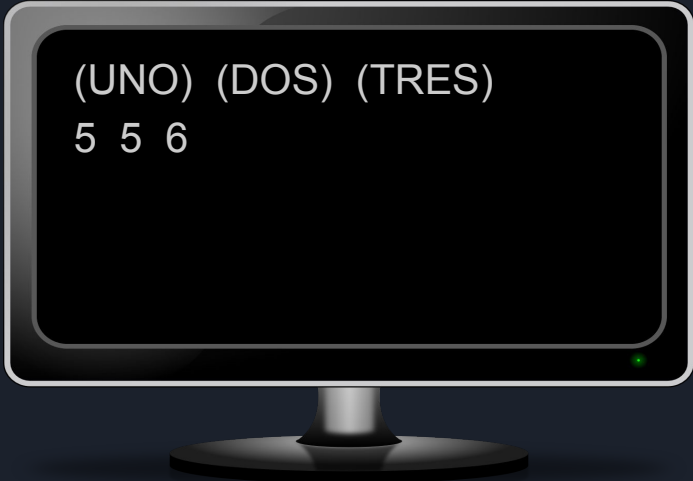
```
(UNO) (DOS) (TRES)  
5 5 6
```

Posible solución

```
List<string> lista = ["uno", "dos", "tres"];  
IEnumerable<string> secuencia = lista.Select(st => $"({st})".ToUpper());  
Mostrar(secuencia);  
IEnumerable<int> secuencia2 = secuencia.Select(st => st.Length);  
Mostrar(secuencia2);
```



Observar que `secuencia2` es de un tipo distinto a `secuencia` (el método `Select` es un método genérico, se está haciendo inferencia de parámetros de tipos a partir del argumento, en este caso de tipo `Func<string,int>`), por lo tanto se está invocando a `Select<string,int>`



```
(UNO) (DOS) (TRES)  
5 5 6
```



Posible solución

```
List<string> lista = ["uno", "dos", "tres"];  
IEnumerable<string> secuencia = lista.Select(st => $"({st})".ToUpper());  
Mostrar(secuencia);  
IEnumerable<int> secuencia2 = secuencia.Select(st => st.Length);  
Mostrar(secuencia2);
```

Este es el encabezado del método de extensión `Select` definido en la clase estática `System.Linq.Enumerable`

```
public static IEnumerable<TResult> Select<TSource, TResult>(  
    this IEnumerable<TSource> source, Func<TSource, TResult> selector)
```

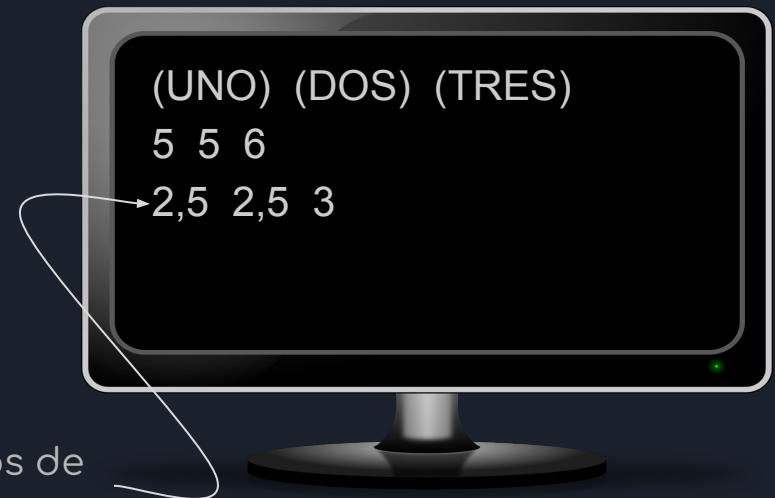


Completar la línea que falta para que la salida por consola sea la que se indica



```
List<string> lista = ["uno", "dos", "tres"];
IEnumerable<string> secuencia = lista.Select(st => $"({st})".ToUpper());
Mostrar(secuencia);
IEnumerable<int> secuencia2 = secuencia.Select(st => st.Length);
Mostrar(secuencia2);
→ IEnumerable<double> secuencia3 = xxxxxxxxxxxxxxxxxxxxxxxxx
Mostrar(secuencia3);
```

```
void Mostrar<T>(IEnumerable<T> secuencia)
{
    foreach (T elemento in secuencia)
    {
        Console.Write(elemento + " ");
    }
    Console.WriteLine();
}
```

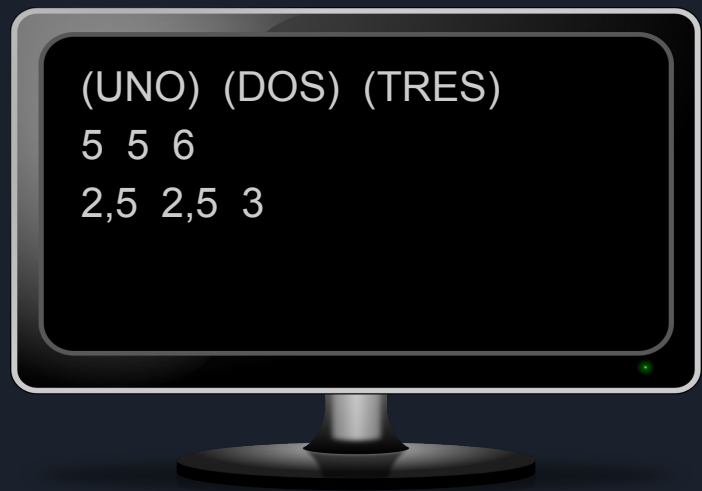


la mitad de los elementos de
la secuencia anterior

Posible solución

```
List<string> lista = ["uno", "dos", "tres"];
IEnumerable<string> secuencia = lista.Select(st => $"({st})".ToUpper());
Mostrar(secuencia);
IEnumerable<int> secuencia2 = secuencia.Select(st => st.Length);
Mostrar(secuencia2);
IEnumerable<double> secuencia3 = secuencia2.Select(n => n / 2.0);
Mostrar(secuencia3);
```

```
void Mostrar<T>(IEnumerable<T> secuencia)
{
    foreach (T elemento in secuencia)
    {
        Console.Write(elemento + " ");
    }
    Console.WriteLine();
}
```



(UNO) (DOS) (TRES)
5 5 6
2,5 2,5 3




Posible solución

```
List<string> lista = ["uno", "dos", "tres"];  
IEnumerable<string> secuencia = lista.Select(st => $"({st})".ToUpper());  
Mostrar(secuencia);  
IEnumerable<int> secuencia2 = secuencia.Select(st => st.Length);  
Mostrar(secuencia2);  
IEnumerable<double> secuencia3 = secuencia2.Select(n => n / 2.0);  
Mostrar(secuencia3);
```



Dividimos por 2.0 los elementos enteros de secuencia2 obteniendo un `IEnumerable<double>`
Los argumentos de tipo del método `Select` se infieren por medio del argumento que en este caso es de tipo `Func<int,double>`



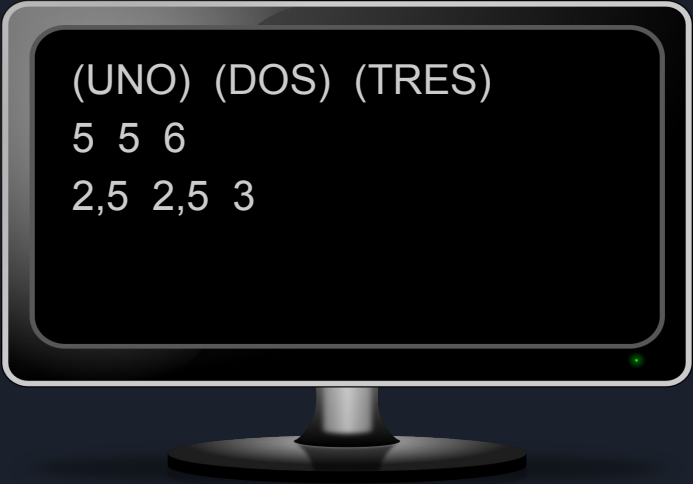
(UNO)	(DOS)	(TRES)
5	5	6
2,5	2,5	3



Posible solución

```
List<string> lista = ["uno", "dos", "tres"];  
var secuencia = lista.Select(st => $"({st})".ToUpper());  
Mostrar(secuencia);  
var secuencia2 = secuencia.Select(st => st.Length);  
Mostrar(secuencia2);  
var secuencia3 = secuencia2.Select(n => n / 2.0);  
Mostrar(secuencia3);
```

Es muy común utilizar LINQ con inferencia de tipos (palabra clave `var`) para simplificar la escritura y lectura del código



```
(UNO) (DOS) (TRES)  
5 5 6  
2,5 2,5 3
```

Interfaz Fluida de LINQ

```
List<string> lista = ["uno", "dos", "tres"];  
var secuencia = lista.Select(st => $"({st})".ToUpper())  
                      .Select(st => st.Length)  
                      .Select(n => n / 2.0);  
Mostrar(secuencia);
```

Si sólo nos interesa el último resultado es muy común utilizar la interfaz fluida ([fluent API](#)) de [LINQ](#)



2,5 2,5 3

```
List<int> numeros = [1, 10, 7, 3, 11];  
Mostrar(numeros);
```

Además de `Select`, LINQ
provee muchos otros
métodos de extensión:
`Where`, `Reverse`, `OrderBy`,
`Sum`, `Average` son
sólo alguno de ellos

A computer monitor with a black frame and a silver stand. The screen is black, and the numbers '1 10 7 3 11' are displayed in a white font. A white arrow points from the 'Mostrar(numeros);' line of code to the first number '1' on the screen.

1 10 7 3 11

```
List<int> numeros = [1, 10, 7, 3, 11];  
Mostrar(numeros);  
var mayores6 = numeros.Where(n => n > 6);  
Mostrar(mayores6);
```

Además de `Select`, LINQ
provee muchos otros
métodos de extensión:
`Where`, `Reverse`, `OrderBy`,
`Sum`, `Average` son
sólo alguno de ellos

A computer monitor with a black frame and a silver stand. The screen is black and displays the output of the LINQ query. The first line shows the original list: "1 10 7 3 11". The second line shows the filtered list: "10 7 11". A white arrow points from the `Mostrar(mayores6);` line in the code block to the second line of the monitor's output.

1 10 7 3 11
10 7 11

```
List<int> numeros = [1, 10, 7, 3, 11];  
Mostrar(numeros);  
var mayores6 = numeros.Where(n => n > 6);  
Mostrar(mayores6);  
var reverso = mayores6.Reverse();  
Mostrar(reverso);
```

Además de Select, LINQ
provee muchos otros
métodos de extensión:
Where, Reverse, OrderBy,
Sum, Average son
sólo alguno de ellos

A computer monitor with a black frame and a silver stand. The screen displays the output of the LINQ code. The first line shows the original array: "1 10 7 3 11". The second line shows the filtered array: "10 7 11". The third line shows the reversed array: "11 7 10", which is highlighted with a blue background. A white arrow points from the "Mostrar(reverso);" line of code to the third line of output on the monitor.

1	10	7	3	11
10	7	11		
11	7	10		

```
List<int> numeros = [1, 10, 7, 3, 11];  
Mostrar(numeros);  
var mayores6 = numeros.Where(n => n > 6);  
Mostrar(mayores6);  
var reverso = mayores6.Reverse();  
Mostrar(reverso);  
var ordenados = reverso.OrderBy(n => n);  
Mostrar(ordenados);
```

Además de Select, LINQ
provee muchos otros
métodos de extensión:
Where, Reverse, OrderBy,
Sum, Average son
sólo alguno de ellos

A computer monitor displaying the output of the LINQ query. The output is a list of numbers: 1 10 7 3 11, 10 7 11, 11 7 10, and 7 10 11. The last line, 7 10 11, is highlighted in blue. A white arrow points from the 'Mostrar(ordenados);' line of code to the monitor.

1	10	7	3	11
10	7	11		
11	7	10		
7	10	11		

```
List<int> numeros = [1, 10, 7, 3, 11];  
Mostrar(numeros);  
var mayores6 = numeros.Where(n => n > 6);  
Mostrar(mayores6);  
var reverso = mayores6.Reverse();  
Mostrar(reverso);  
var ordenados = reverso.OrderBy(n => n);  
Mostrar(ordenados);  
var suma = ordenados.Sum();  
var promedio = ordenados.Average();  
Console.WriteLine($"suma: {suma} promedio:{promedio:0.00}");
```

Además de Select, LINQ
provee muchos otros
métodos de extensión:
Where, Reverse, OrderBy,
Sum, Average son
sólo alguno de ellos



```
1 10 7 3 11  
10 7 11  
11 7 10  
7 10 11
```

```
suma: 28 promedio:9,33
```

LINQ - Ejemplos

```
List<int> lista1 = [1,2,3,4,5,1,2,3,4,5];  
List<int> lista2 = [4,5,6,4,5,6];  
var concat = lista1.Concat(lista2);  
Mostrar(concat);
```

Otros métodos involucran
a más de una secuencia,
Concat, Union, Intersect y
Zip son
sólo alguno de ellos



1 2 3 4 5 1 2 3 4 5 4 5 6 4 5 6

LINQ - Ejemplos

```
List<int> lista1 = [1,2,3,4,5,1,2,3,4,5];  
List<int> lista2 = [4,5,6,4,5,6];  
var concat = lista1.Concat(lista2);  
Mostrar(concat);  
var union = lista1.Union(lista2);  
Mostrar(union);
```

Representa la unión
entre conjuntos, por eso
no aparecen elementos
repetidos

Otros métodos involucran
a más de una secuencia,
Concat, Union, Intersect y
Zip son
sólo alguno de ellos

A computer monitor with a black frame and a silver stand. The screen displays two rows of numbers. The top row contains the sequence '1 2 3 4 5 1 2 3 4 5 4 5 6 4 5 6'. The bottom row contains the sequence '1 2 3 4 5 6', which is highlighted with a blue background. An arrow points from the text 'no aparecen elementos repetidos' to the bottom row.

1 2 3 4 5 1 2 3 4 5 4 5 6 4 5 6
1 2 3 4 5 6

LINQ - Ejemplos

```
List<int> lista1 = [1,2,3,4,5,1,2,3,4,5];  
List<int> lista2 = [4,5,6,4,5,6];  
var concat = lista1.Concat(lista2);  
Mostrar(concat);  
var union = lista1.Union(lista2);  
Mostrar(union);  
var interseccion = lista1.Intersect(lista2);  
Mostrar(interseccion);
```

Representa la
intersección entre
conjuntos, por eso no
aparecen elementos
repetidos

Otros métodos involucran
a más de una secuencia,
Concat, Union, Intersect y
Zip son
sólo alguno de ellos

A computer monitor is shown at the bottom right of the slide. On its screen, the intersection of the two lists is displayed. The first line shows the elements of lista1: 1 2 3 4 5 1 2 3 4 5. The second line shows the elements of lista2: 1 2 3 4 5 6. The third line shows the intersection result: 4 5, which is highlighted with a blue background. An arrow points from the 'Mostrar(interseccion);' line of code to this monitor.

1 2 3 4 5 1 2 3 4 5 1 2 3 4 5 6 4 5 6
1 2 3 4 5 6
4 5

LINQ - Ejemplos

```
List<int> lista1 = [1,2,3,4,5,1,2,3,4,5];  
List<int> lista2 = [4,5,6,4,5,6];  
var concat = lista1.Concat(lista2);  
Mostrar(concat);  
var union = lista1.Union(lista2);  
Mostrar(union);  
var interseccion = lista1.Intersect(lista2);  
Mostrar(interseccion);  
var zip = lista1.Zip(lista2, (a, b) => a + b);  
Mostrar(zip);
```

Se utiliza para combinar dos
secuencias en una sola
secuencia aplicando una
función de combinación que se
aplica elemento a elemento
posicionalmente

$$\begin{array}{r} [1,2,3,4, 5,1,2,3,4,5]; \\ + [4,5,6,4, 5,6]; \\ \hline [5,7,9,8,10,7] \end{array}$$

Otros métodos involucran
a más de una secuencia,
Concat, Union, Intersect y
Zip son
sólo alguno de ellos

A computer monitor displaying the result of the Zip operation. The screen shows three rows of numbers: the first row is '1 2 3 4 5 1 2 3 4 5 4 5 6 4 5 6', the second row is '1 2 3 4 5 6', and the third row is '4 5'. Below these, the result of the Zip operation is shown as '5 7 9 8 10 7', with the numbers 5, 7, 9, 8, 10, and 7 highlighted in blue.

1 2 3 4 5 1 2 3 4 5 4 5 6 4 5 6
1 2 3 4 5 6
4 5
5 7 9 8 10 7

//calculamos la suma de los primeros
//20 cuadrados: 1 + 4 + 9 + ... + 400

```
var suma = Enumerable.Range(1, 20)           // 1, 2, 3, ..., 20
                        .Select(n => n * n)    // 1, 4, 9, ... 400
                        .Sum();                // 1 + 4 + 9 + ... + 400
Console.WriteLine($"Resultado: {suma}");
```

Primer valor del rango

Cantidad de elementos
del rango



El método estático
Range(start,count) de la
clase System.Linq.Enumerable
devuelve un IEnumerable<int>
con la secuencia de enteros
comenzando por start
y con count elementos

Resultado: 2870



Codificar la clase
Persona que vamos a
utilizar para mostrar
más ejemplos de las
facilidades que brinda
LINQ

```
class Persona
{
    public string Nombre { get; private set; }
    public int Edad { get; private set; }
    public string Pais { get; private set; }
    public Persona(string nombre, int edad, string pais)
    {
        Nombre = nombre;
        Edad = edad;
        Pais = pais;
    }
    public override string ToString()
    {
        return $"{Nombre} ({Edad} años) {Pais.Substring(0, 3)}.";
    }

    // vamos a hardcodear una lista de personas
    // que usaremos en los siguientes ejemplos
    // para ello definimos el siguiente método estático
    public static List<Persona> GetLista()
    {
        return new List<Persona>() {
            new Persona("Pablo", 15, "Argentina"),
            new Persona("Juan", 55, "Argentina"),
            new Persona("José", 9, "Uruguay"),
            new Persona("María", 33, "Uruguay"),
            new Persona("Lucía", 16, "Perú"),
        };
    }
}
```

Copiar el código del archivo
10_RecursosParaLaTeoria.txt



Completar el código para que la salida por consola sea la indicada (mayores de 18)



```
var personas = Persona.GetLista();  
personas.ForEach(p => Console.WriteLine(p)); // lista todas las personas  
Console.WriteLine();  
. . .
```

Listar las personas
mayores de edad

Pablo (15 años) Arg.
Juan (55 años) Arg.
José (9 años) Uru.
María (33 años) Uru.
Lucía (16 años) Per.

[Juan (55 años) Arg.
[María (33 años) Uru.



Posible solución

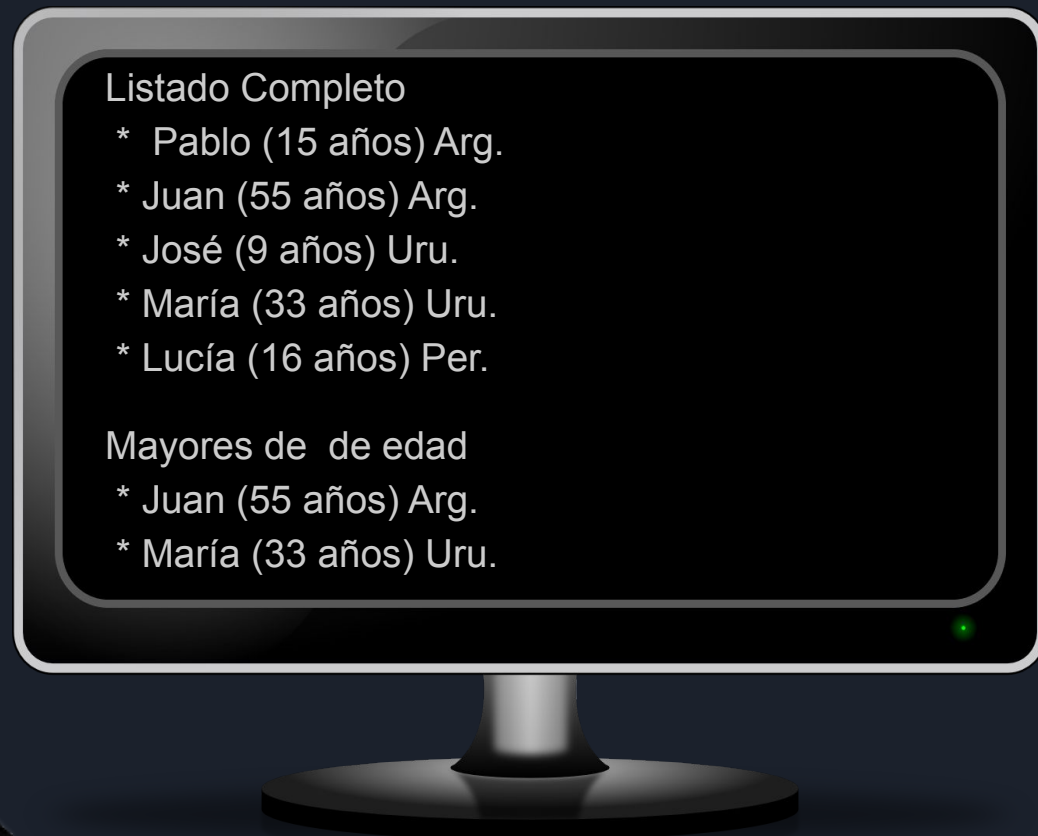

```
var personas = Persona.GetLista();  
personas.ForEach(p => Console.WriteLine(p)); // lista todas las personas  
Console.WriteLine();  
personas.Where(p => p.Edad >= 18) // un IEnumerable<Persona> no tiene método Foreach  
    .ToList() // lo convierte en un List<Persona> para usar método Foreach  
    .ForEach(p => Console.WriteLine(p)); // lista personas mayores de edad
```

Pablo (15 años) Arg.
Juan (55 años) Arg.
José (9 años) Uru.
María (33 años) Uru.
Lucía (16 años) Per.

Juan (55 años) Arg.
María (33 años) Uru.

Para pensar: ¿Cómo deberíamos codificar el método de extensión `ImprimirEnConsola`?

```
Persona.GetLista()  
    .ImprimirEnConsola("Listado Completo")  
    .Where(p => p.Edad >= 18)  
    .ImprimirEnConsola("\nMayores de de edad");
```



Listado Completo

- * Pablo (15 años) Arg.
- * Juan (55 años) Arg.
- * José (9 años) Uru.
- * María (33 años) Uru.
- * Lucía (16 años) Per.

Mayores de de edad

- * Juan (55 años) Arg.
- * María (33 años) Uru.

Para pensar: ¿Cómo deberíamos codificar el método de extensión ImprimirEnConsola?

```
static class Extension
{
    public static IEnumerable<T> ImprimirEnConsola<T>(
        this IEnumerable<T> secuencia,
        string titulo)
    {
        Console.WriteLine(titulo);
        foreach(T elemento in secuencia)
        {
            Console.WriteLine(" * " + elemento);
        }
        return secuencia;
    }
}
```



```
Persona.GetLista()  
    .OrderBy(p => p.Edad)  
    .Select(p => new { Nombre = p.Nombre, Condición = p.Edad < 18 ? "Menor" : "Mayor" })  
    .ImprimirEnConsola("Usando tipos anónimos");
```



También es común
devolver tipos anónimos.
En este caso el método
Select devuelve un
IEnumerable de un tipo
anónimo que tiene las
propiedades
Nombre y Condición

Usando tipos anónimos

- * { Nombre = José, Condición = Menor }
- * { Nombre = Pablo, Condición = Menor }
- * { Nombre = Lucía, Condición = Menor }
- * { Nombre = María, Condición = Mayor }
- * { Nombre = Juan, Condición = Mayor }

```
var personas = Persona.GetLista();  
Console.WriteLine($"Primero: {personas.First()}");  
Console.WriteLine($"Último: {personas.Last()}");  
Console.WriteLine($"Edad máxima: {personas.Max(p => p.Edad)}");  
Console.WriteLine($"Edad mínima: {personas.Min(p => p.Edad)}");  
Console.WriteLine($"Son todos mayores? {personas.All(p => p.Edad >= 18)}");  
Console.WriteLine($"Hay algún mayor? {personas.Any(p => p.Edad >= 18)}");
```



First, Last, Max, Min,
All, Any son algunos
métodos más que
brinda LINQ

A black computer monitor with a silver stand, displaying the output of the LINQ code. The text is white on a black background.

Primero: Pablo (15 años) Arg.
Último: Lucía (16 años) Per.
Edad máxima: 55
Edad mínima: 9
Son todos mayores? False
Hay algún mayor? True

```
var personas = Persona.GetLista();
var grupos = personas.GroupBy(p => p.Pais);
foreach (var grup in grupos)
{
    Console.WriteLine($"{grup.Key} ({grup.Count()})");
    foreach(var p in grup )
    {
        Console.WriteLine(" * " + p);
    }
}
```

Agrupamos por país.
grupos es un IEnumerable de
IGrouping<string,Persona>

Cada grup representa un
grupo de personas junto a
su clave de agrupación.
grup es de tipo
IGrouping<string,Persona>,
esta interfaz hereda de
IEnumerable<Persona>



LINQ Nos permite
agrupar fácilmente
por algún criterio

```
Argentina (2)
* Pablo (15 años) Arg.
* Juan (55 años) Arg.
Uruguay (2)
* José (9 años) Uru.
* María (33 años) Uru.
Perú (1)
* Lucía (16 años) Per.
```

```
Persona.GetLista()  
    .GroupBy(p => p.Pais)  
    .ToList()  
    .ForEach(grup =>  
    {  
        Console.WriteLine($"{grup.Key} ({grup.Count()})");  
        grup.ToList().ForEach(p => Console.WriteLine(" * " + p));  
    });
```

También podemos
hacerlo de esta manera
evitando la estructura
de control `foreach`.



```
Argentina (2)  
 * Pablo (15 años) Arg.  
 * Juan (55 años) Arg.  
Uruguay (2)  
 * José (9 años) Uru.  
 * María (33 años) Uru.  
Perú (1)  
 * Lucía (16 años) Per.
```

```
Persona.GetLista()  
    .GroupBy(p => p.Pais)  
    .ToList()  
    .ForEach(grup => grup.ImprimirEnConsola($"{grup.Key} ({grup.Count()})"));
```



Utilizando nuestro
método
`ImprimirEnConsola`
(100% interfaz fluida)

A computer monitor with a black bezel and a silver stand. The screen is black and displays the output of the LINQ query in white text. The output is grouped by country: Argentina (2), Uruguay (2), and Perú (1). Each group is followed by a list of names and ages, each preceded by an asterisk.

```
Argentina (2)  
* Pablo (15 años) Arg.  
* Juan (55 años) Arg.  
Uruguay (2)  
* José (9 años) Uru.  
* María (33 años) Uru.  
Perú (1)  
* Lucía (16 años) Per.
```



Utilizar GroupBy para listar las personas agrupadas por la inicial de su nombre



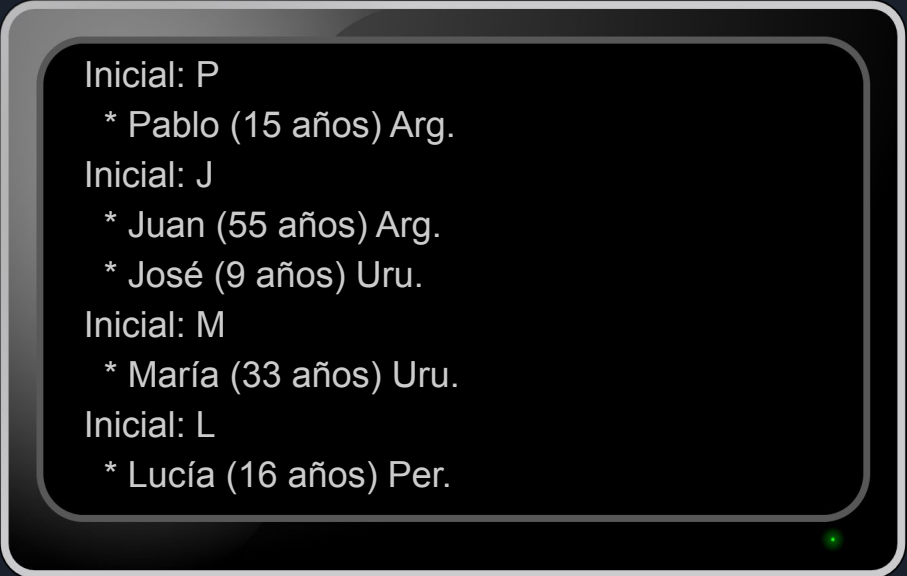
```
var personas = Persona.GetLista();  
var agrupadas = personas.GroupBy(xxxxxxxxxxxxxxxxxx);  
foreach(var grupo in agrupadas)  
    . . .
```

```
Inicial: P  
    * Pablo (15 años) Arg.  
Inicial: J  
    * Juan (55 años) Arg.  
    * José (9 años) Uru.  
Inicial: M  
    * María (33 años) Uru.  
Inicial: L  
    * Lucía (16 años) Per.
```



Posible solución

```
var personas = Persona.GetLista();  
var agrupadas = personas.GroupBy(p => p.Nombre[0]);  
foreach(var grupo in agrupadas)  
{  
    Console.WriteLine($"Inicial: {grupo.Key}");  
    grupo.ToList().ForEach(p => Console.WriteLine("    * " + p));  
}
```



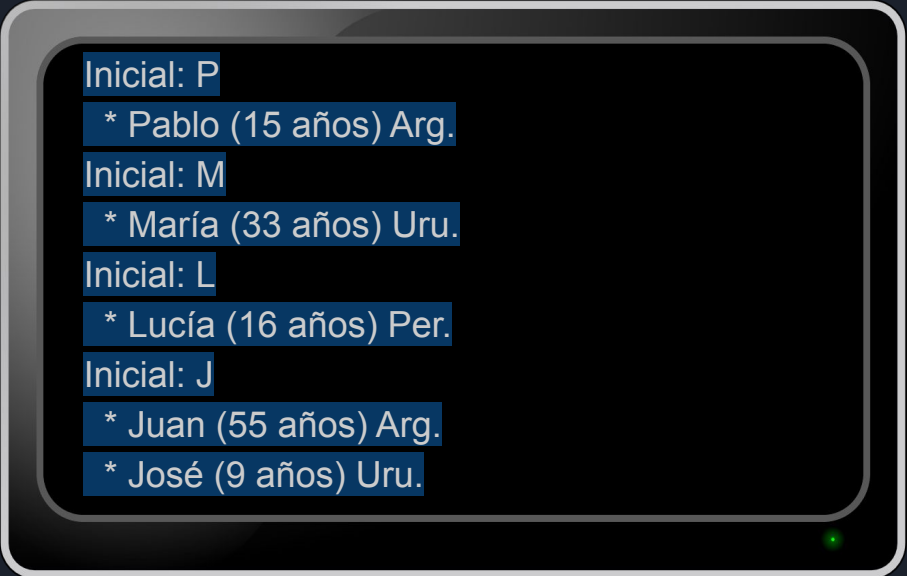
```
Inicial: P  
    * Pablo (15 años) Arg.  
Inicial: J  
    * Juan (55 años) Arg.  
    * José (9 años) Uru.  
Inicial: M  
    * María (33 años) Uru.  
Inicial: L  
    * Lucía (16 años) Per.
```




Ordenar el listado por inicial de manera descendente



```
var personas = Persona.GetLista();  
var agrupadas = personas.GroupBy(p => p.Nombre[0]).xxxxxxxxxxxxxxxxxx;  
foreach(var grupo in agrupadas)  
{  
    Console.WriteLine($"Inicial: {grupo.Key}");  
    grupo.ToList().ForEach(p => Console.WriteLine(" * " + p));  
}
```

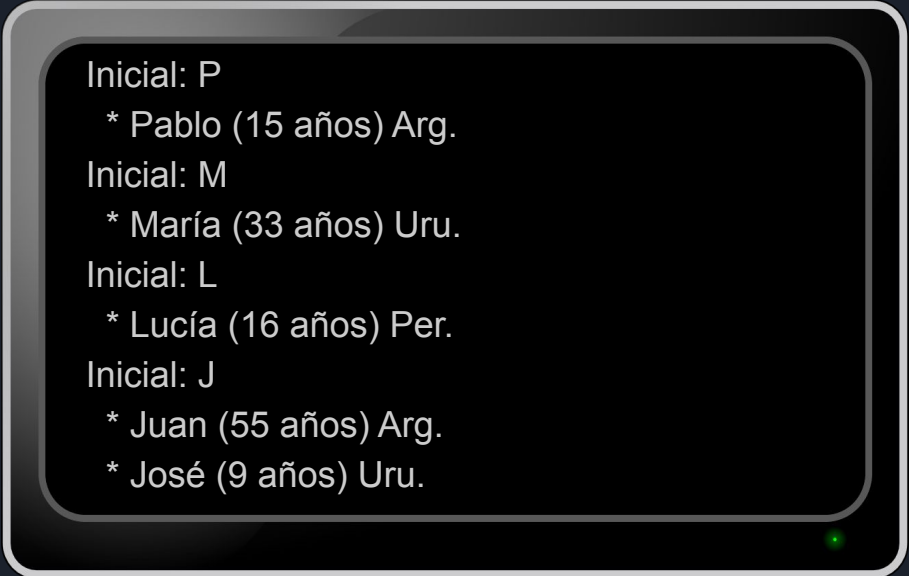


```
Inicial: P  
 * Pablo (15 años) Arg.  
Inicial: M  
 * María (33 años) Uru.  
Inicial: L  
 * Lucía (16 años) Per.  
Inicial: J  
 * Juan (55 años) Arg.  
 * José (9 años) Uru.
```



Posible solución

```
var personas = Persona.GetLista();  
var agrupadas = personas.GroupBy(p => p.Nombre[0]).OrderByDescending(g=>g.Key);  
foreach(var grupo in agrupadas)  
{  
    Console.WriteLine($"Inicial: {grupo.Key}");  
    grupo.ToList().ForEach(p => Console.WriteLine("    * " + p));  
}
```



```
Inicial: P  
    * Pablo (15 años) Arg.  
Inicial: M  
    * María (33 años) Uru.  
Inicial: L  
    * Lucía (16 años) Per.  
Inicial: J  
    * Juan (55 años) Arg.  
    * José (9 años) Uru.
```



Posible solución

```
Persona.GetLista()  
    .GroupBy(p => p.Nombre[0])  
    .OrderByDescending(g => g.Key)  
    .ToList()  
    .ForEach(grupo => grupo.ImprimirEnConsola($"Inicial: {grupo.Key}"));
```



Utilizando interfaz
fluida y nuestro
método
ImprimirEnConsola

Inicial: P
* Pablo (15 años) Arg.
Inicial: M
* María (33 años) Uru.
Inicial: L
* Lucía (16 años) Per.
Inicial: J
* Juan (55 años) Arg.
* José (9 años) Uru.



Ejercicio



Utilizando el método `Enumerable.Range` agrupar los números enteros entre 1 y 20 según el resto de dividir por 3

```
Resto de dividir por 3 = 1
```

```
1 4 7 10 13 16 19
```

```
Resto de dividir por 3 = 2
```

```
2 5 8 11 14 17 20
```

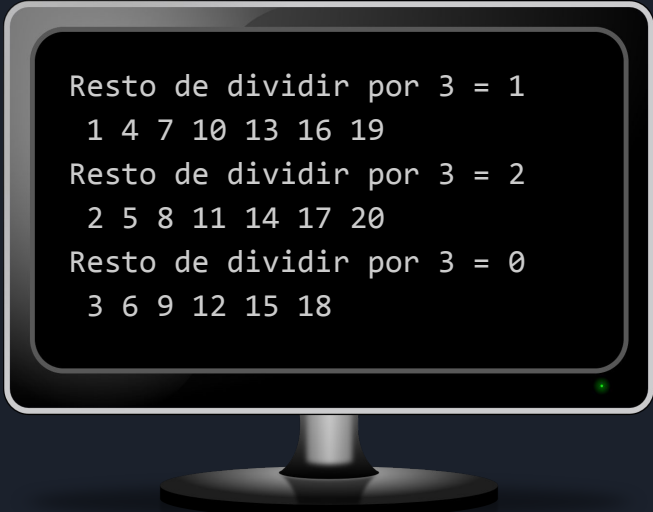
```
Resto de dividir por 3 = 0
```

```
3 6 9 12 15 18
```



Posible Solución

```
Enumerable.Range(1, 20)
    .GroupBy(i => i % 3)
    .ToList()
    .ForEach(grupo =>
    {
        Console.WriteLine($"Resto de dividir por 3 = {grupo.Key}");
        grupo.ToList().ForEach(i => Console.Write($" {i}"));
        Console.WriteLine();
    });
```



```
Resto de dividir por 3 = 1
 1 4 7 10 13 16 19
Resto de dividir por 3 = 2
 2 5 8 11 14 17 20
Resto de dividir por 3 = 0
 3 6 9 12 15 18
```



Vamos a utilizar
estas dos clases
Alumno y
Examen en los
siguientes
ejemplos

```
class Alumno
{
    public int Id { get; private set; }
    public string Nombre { get; private set; }
    public Alumno(int id, string nombre)
    {
        Id = id;
        Nombre = nombre;
    }
}

class Examen
{
    public int AlumnoId { get; private set; }
    public string Materia { get; private set; }
    public double Nota { get; private set; }
    public Examen(int alumnoId, string materia, double nota)
    {
        AlumnoId = alumnoId;
        Materia = materia;
        Nota = nota;
    }
}
```

```
var alumnos = new List<Alumno>() {  
    new Alumno(1,"Juan"),  
    new Alumno(2,"Ana"),  
    new Alumno(3,"Laura")  
};
```

```
var examenes = new List<Examen>() {  
    new Examen(2,"Inglés",9),  
    new Examen(1,"Inglés",5),  
    new Examen(1,"Álgebra",10)  
};
```

```
var listado = alumnos.Join(examenes,  
    a => a.Id, //clave de matching en alumnos  
    e => e.AlumnoId, //clave de matching en notas  
    (a, e) => new  
    {  
        Alumno = a.Nombre,  
        Materia = e.Materia,  
        Notas = e.Nota  
    });  
listado.ToList().ForEach(obj => Console.WriteLine(obj));
```

Alumnos

Nombre	Id
Juan	1
Ana	2
Laura	3

Exámenes

AlumnoId	Materia	Nota
2	Inglés	9
1	Inglés	5
1	Álgebra	10

Join es como inner join de SQL, devuelve sólo los elementos donde las claves coinciden.

```
{ Alumno = Juan, Materia = Inglés, Notas = 5 }  
{ Alumno = Juan, Materia = Álgebra, Notas = 10 }  
{ Alumno = Ana, Materia = Inglés, Notas = 9 }
```

LINQ - Ejemplos

```
var alumnos = new List<Alumno>() {  
    new Alumno(1,"Juan"),  
    new Alumno(2,"Ana"),  
    new Alumno(3,"Laura")  
};
```

```
var examenes = new List<Examen>() {  
    new Examen(2,"Inglés",9),  
    new Examen(1,"Inglés",5),  
    new Examen(1,"Álgebra",10)  
};
```

```
var listado = alumnos.GroupJoin(examenes,  
    a => a.Id, //clave de matching en alumnos  
    e => e.AlumnoId, //clave de matching en notas  
    (a, susExamenes) => new  
    {  
        NombreAlumno = a.Nombre,  
        Examenes = susExamenes  
    });
```

```
foreach (var grup in listado)  
{  
    Console.WriteLine($"Alumno: {grup.NombreAlumno}");  
    Console.WriteLine($" Cant. exams.: {grup.Examenes.Count()}");  
    foreach (var e in grup.Examenes)  
    {  
        Console.WriteLine($" {e.Materia} Nota:{e.Nota}");  
    }  
}
```

Alumnos

Nombre	Id
Juan	1
Ana	2
Laura	3

Exámenes

AlumnoId	Materia	Nota
2	Inglés	9
1	Inglés	5
1	Álgebra	10

Can GroupJoin
agrupamos a los
alumno con sus
exámenes. También
obtengo información
de quienes no rindieron
ninguno

```
Alumno: Juan, Cant. exams.: 2  
Inglés Nota:5  
Álgebra Nota:10  
Alumno: Ana, Cant. exams.: 1  
Inglés Nota:9  
Alumno: Laura, Cant. exams.: 0
```


¡LINQ es poderoso!

Hemos aprendido a consultar y transformar colecciones en memoria de forma elegante y declarativa.

Pero... ¿qué pasa cuando los datos NO están en memoria?

¿Podríamos usar una sintaxis parecida a LINQ para "hablar" con esos datos persistentes?

¡Sí! Y aquí es donde entra en juego la **persistencia de datos** y herramientas como **Entity Framework Core**.

¡A guardar y consultar datos de verdad!

Persistencia de datos

Vamos a usar SQLite para persistir datos

SQLite es una biblioteca que implementa un motor de base de datos SQL “en proceso”, autónomo, sin servidor, de configuración cero.

SQLite es de código abierto y, por lo tanto, es gratuito para su uso para cualquier propósito.





Crear una aplicación de consola llamada Escuela

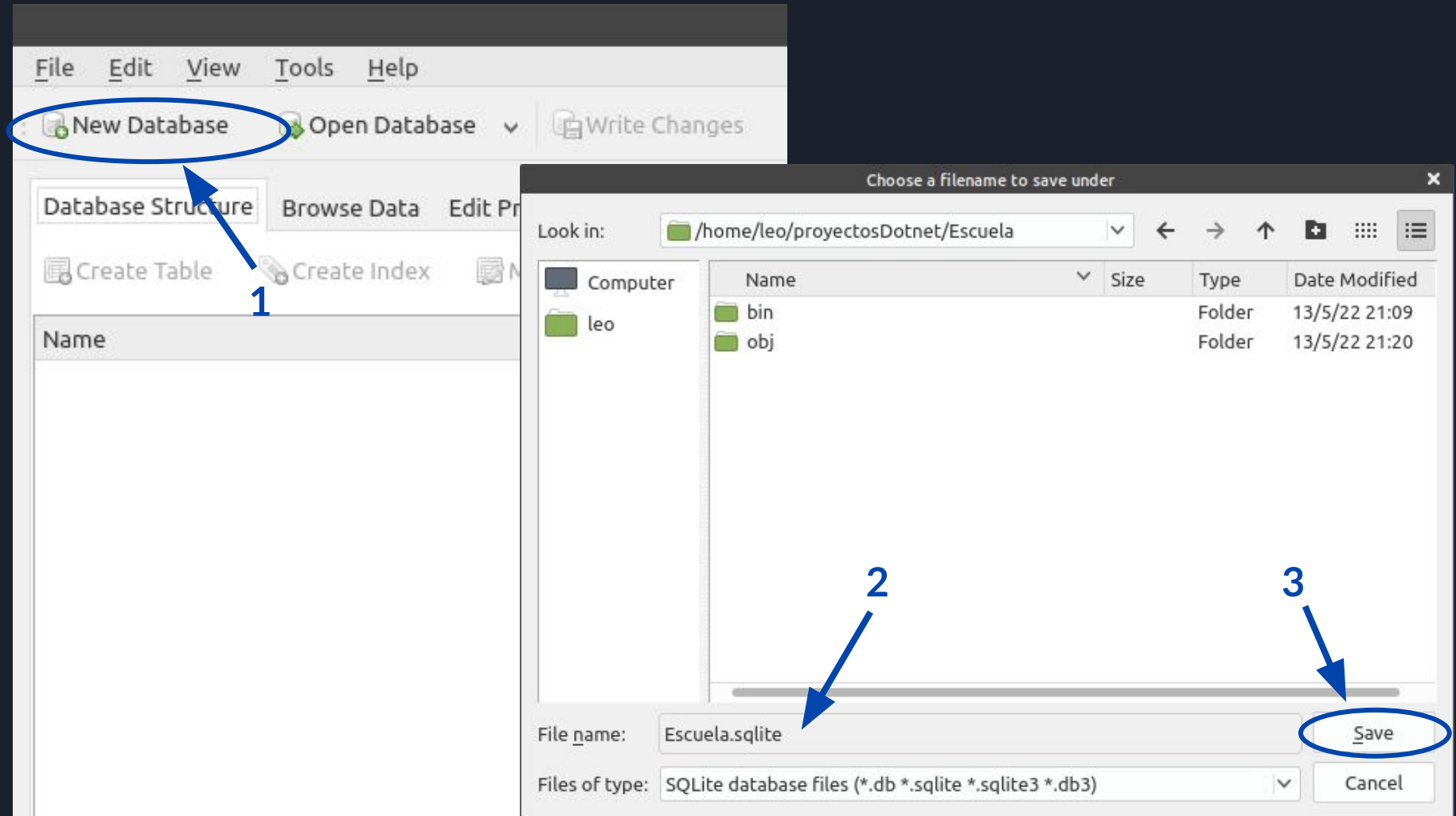


1. Abrir una terminal del sistema operativo
2. Cambiar a la carpeta `proyectosDotnet`
3. Crear la aplicación de consola `Escuela`
4. En la carpeta raíz del proyecto crear la base de datos `Escuela.sqlite` utilizando `DB Browser for SQLite`

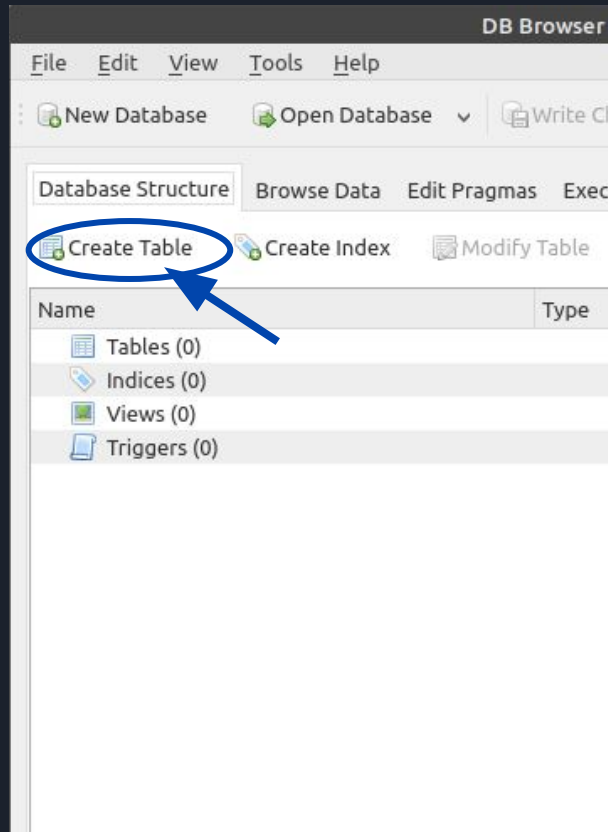
Descargar e instalar DB Browser for SQLite (<https://sqlitebrowser.org>)



Utilizando **DB Browser** crear la base **Escuela.sqlite** en la carpeta raíz del proyecto que acabamos de crear (donde se encuentra el archivo **Escuela.csproj**)



Crear la tabla **Alumnos** con los campos **Id** de tipo **INTEGER**, **Nombre** y **Email** de tipo **TEXT**



Marcamos el campo **Id** como clave de la tabla (PK). Además marcamos (AI) para que este campo sea establecido automáticamente por SQLite, de manera incremental, al momento de agregar una nueva fila a la tabla. Tanto **Nombre** como **Id** deben ser no nulos

Agregar algunos datos a la tabla Alumnos

The screenshot shows the SQLite GUI interface. The 'Browse Data' tab is selected, and the 'Alumnos' table is displayed. The table has three columns: 'Id', 'Nombre', and 'Email'. The first row contains the values '1', 'Juan', and 'juan@gmail.com'. The second and third rows have '2', 'Ana', 'NULL' and '3', 'Laura', 'NULL' respectively. A blue callout box points to the 'Browse Data' tab with the text 'Pestaña Hoja de datos'. Another blue callout box points to the 'New Record' button with the text 'Presionando este botón se agrega un registro (fila) en blanco que podemos editar. El campo Id se completa solo'.

Pestaña
Hoja de datos

File Edit View Tools Help

New Database Open Database Write Changes Revert Changes

Database Structure Browse Data Edit Pragmas Execute SQL

Table: Alumnos

	Id	Nombre	Email
	Filter	Filter	Filter
1	1	Juan	juan@gmail.com
2	2	Ana	NULL
3	3	Laura	NULL

New Record Delete Record

Presionando este botón se agrega un registro (fila) en blanco que podemos editar. El campo Id se completa solo

Crear la tabla **Exámenes** con los campos que se observan en esta diapositiva

Edit table definition

Table

Exámenes

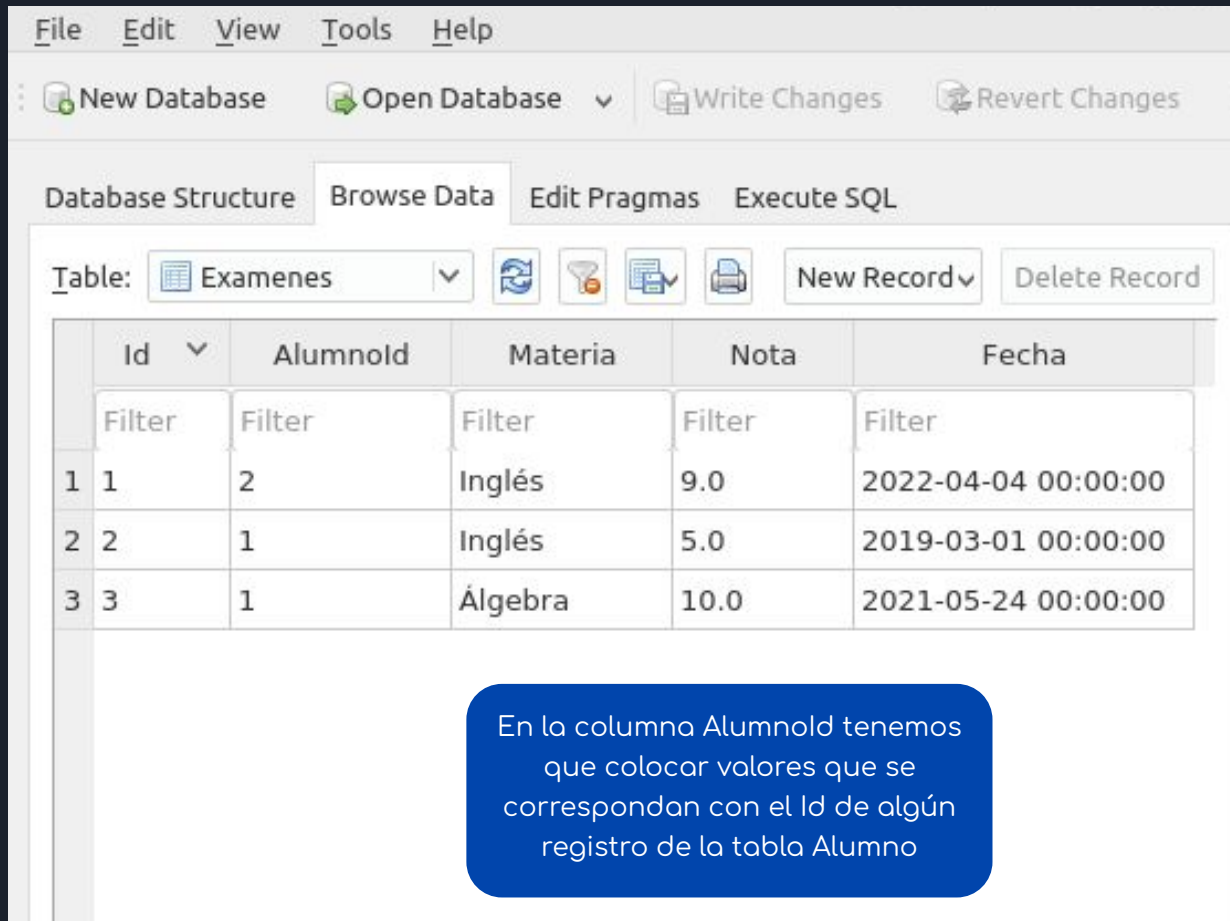
Advanced

Fields

Add field Remove field Move field up Move field down

Name	Type	NN	PK	AI	U	Default	Check	Foreign Key
Id	INTEGER	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>			
Alumnoid	INTEGER	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>			
Materia	TEXT	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>			
Nota	REAL	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>			
Fecha	TEXT	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>			

Agregar algunos datos a la tabla Exámenes



The screenshot shows the SQLite Browser application with the 'Exámenes' table selected. The table has five columns: Id, Alumnold, Materia, Nota, and Fecha. Three records are displayed. A blue callout box highlights the 'Alumnold' column, stating that values should correspond to the Id of a record in the 'Alumno' table.

	Id	Alumnold	Materia	Nota	Fecha
1	1	2	Inglés	9.0	2022-04-04 00:00:00
2	2	1	Inglés	5.0	2019-03-01 00:00:00
3	3	1	Álgebra	10.0	2021-05-24 00:00:00

En la columna Alumnold tenemos que colocar valores que se correspondan con el Id de algún registro de la tabla Alumno

Entity Framework Core

Nuestra aplicación de consola accederá a los datos de **Escuela.sqlite** mediante **Entity Framework Core**.

EF Core es un **ORM** (object-relational mapper) que permite trabajar con una base de datos utilizando objetos .Net y ahorrando la escritura de mucho código de acceso a datos





Instalar el paquete Nuget Entity Framework Core para SQLite



En la terminal del sistema operativo (o en la que provee el **Visual Studio Code**) posicionarse en la carpeta del proyecto (donde se encuentra el archivo **Escuela.csproj**) y tipear el siguiente comando:

```
dotnet add package Microsoft.EntityFrameworkCore.Sqlite
```

Este es el nombre del paquete
que se requiere instalar

Copiar el código del archivo
10_RecursosParaLaTeoria

Entity Framework Core

Con **EF Core**, el acceso a datos se realiza mediante un **modelo**. Un modelo se compone de **clases de entidad** y un **objeto de contexto** que representa **una sesión con la base de datos**. Este objeto de contexto permite consultar y guardar datos.





Codificar la clases de entidad Alumno



```
namespace Escuela;

public class Alumno
{
    public int Id { get; private set; }
    public string Nombre { get; private set; } = "";
    public string? Email { get; private set; }

    public Alumno(string nombre, string? email = null)
    {
        if (string.IsNullOrEmpty(nombre))
        {
            throw new ArgumentException("El nombre no puede ser nulo ni estar vacío");
        }

        if (email != null && !EmailValido(email))
        {
            throw new ArgumentException("El formato del email no es válido.");
        }

        this.Nombre = nombre;
        this.Email = email;
        //El Id va a ser establecido por el sistema de persistencia
    }
    protected Alumno() { } // Constructor vacío (lo utilizará EntityFramework)

    private static bool EmailValido(string email)
    {
        // Una validación de email real puede ser más compleja
        return email.Contains('@') && email.Contains('.');
    }

    // Otros métodos de la entidad Alumno que implementan su comportamiento
    // permitiendo cambiar su estado y mantener su propia consistencia
    // es decir, las invariantes de la entidad (reglas que siempre deben ser verdaderas).
}
```

Copiar el código del archivo
10_RecursosParaLaTeoria

Persistencia de datos - SQLite - Entity Framework Core

```
namespace Escuela;

public class Alumno
{
    public int Id { get; private set; }
    public string Nombre { get; private set; } = "";
    public string? Email { get; private set; }

    public Alumno(string nombre, string? email = null)
    {
        if (string.IsNullOrEmpty(nombre))
        {
            throw new ArgumentException("El nombre no puede ser nulo ni estar vacío");
        }

        if (email != null && !EmailValido(email))
        {
            throw new ArgumentException("El formato del email no es válido.");
        }

        this.Nombre = nombre;
        this.Email = email;
        //El Id va a ser establecido por el sistema de persistencia
    }
    protected Alumno() { } // Constructor vacío (lo utilizará EntityFramework)

    private static bool EmailValido(string email)
    {
        // Una validación de email real puede ser más compleja
        return email.Contains('@') && email.Contains('.');
    }

    // Otros métodos de la entidad Alumno que implementan su comportamiento
    // permitiendo cambiar su estado y mantener su propia consistencia
    // es decir, las invariantes de la entidad (reglas que siempre deben ser verdaderas).
}
```

Observar que las propiedades se corresponden con la estructura de los datos guardados en la tabla **Alumnos** en la base de datos

Persistencia de datos - SQLite - Entity Framework Core

```
namespace Escuela;
```

```
public class Alumno
```

```
{
```

```
    public int Id { get; private set; }
```

```
    public string Nombre { get; private set; } = "";
```

```
    pu
```

```
    pu
```

```
{
```

Cuando trabajamos con **Entity Framework**, si bien no es necesario en todos los casos, es una buena práctica definir un **constructor protected sin parámetros**. Así pueden evitarse dificultades en algunos escenarios, por ejemplo al implementar la carga diferida

```
    nulo ni estar vacío");
```

```
    no es válido.");
```

```
    this.Nombre = nombre;
```

```
    this.Email = email;
```

```
    //El Id va a ser establecido por el sistema de persistencia
```

```
}
```

```
protected Alumno() { } // Constructor vacío (lo utilizará EntityFramework)
```

```
private static bool EmailValido(string email)
```

```
{
```

```
    // Una validación de email real puede ser más compleja
```

```
    return email.Contains('@') && email.Contains('.');
```

```
}
```

```
// Otros métodos de la entidad Alumno que implementan su comportamiento
```

```
// permitiendo cambiar su estado y mantener su propia consistencia
```

```
// es decir, las invariantes de la entidad (reglas que siempre deben ser verdaderas).
```

```
}
```


Persistencia de datos - SQLite - Entity Framework Core

```
namespace Escuela;

public class Alumno
{
    public int Id { get; private set; }
    public string Nombre { get; private set; } = "";
    public string? Email { get; private set; }

    public Alumno(string nombre, string? email = null)
    {
        if (string.IsNullOrEmpty(nombre))
        {
            throw new ArgumentException("El nombre no puede ser vacío");
        }

        if (email != null)
        {
            throw new ArgumentException("El email no puede ser vacío");
        }

        this.Nombre = nombre;
        this.Email = email;
        //El Id se genera automáticamente por EF
    }
    protected virtual void OnModelCreating(ModelBuilder modelBuilder)
    {
        private static readonly string[] _validEmailDomains = { "com", "net", "org", "edu" };

        // Una validacion de email real puede ser mas compleja
        return email.Contains('@') && email.Contains('.') && _validEmailDomains.Contains(email.Split('@')[1]);
    }

    // Otros métodos de la entidad Alumno que implementan su comportamiento
    // permitiendo cambiar su estado y mantener su propia consistencia
    // es decir, las invariantes de la entidad (reglas que siempre deben ser verdaderas).
}
```

Usamos un parámetro opcional en el constructor. Un parámetro opcional permite establecer un valor por defecto para el caso de no especificarlo en la invocación

Para crear un alumno de nombre "Ana" sin email podemos hacer `new Alumno("Ana", null)` o bien simplemente `new Alumno("Ana");` gracias al parámetro opcional.

```
namespace Escuela;

public class Alumno
{
    public int Id { get; private set; }
    public string Nombre { get; private set; } = "";
    public string? Email { get; private set; }

    public Alumno(string nombre, string? email = null)
    {
        if (string.IsNullOrEmpty(nombre))
        {
            throw new ArgumentException("El nombre no puede ser nulo ni estar vacío");
        }

        if (email != null && !EmailValido(email))
        {
            throw new ArgumentException("El formato del email no es válido.");
        }
    }
}
```

```
this.Nombre = nombre;
this.Email = email;
//El Id va a ser asignado por el framework
}

protected Alumno()
{
    // Una validación de email
    return email;
}
```

En el constructor la entidad se asegura la **consistencia de su estado** manteniendo sus **invariantes** (reglas que siempre deben ser verdaderas)

(Entity Framework)

```
// Otros métodos de la entidad Alumno que implementan su comportamiento
// permitiendo cambiar su estado y mantener su propia consistencia
// es decir, las invariantes de la entidad (reglas que siempre deben ser verdaderas).
```

```
}
```

```
namespace Escuela;

public class Alumno
{
    public int Id { get; private set; }
    public string Nombre { get; private set; } = "";
    public string? Email { get; private set; }

    public Alumno(string nombre, string? email = null)
    {
        if (string.IsNullOrEmpty(nombre))
        {
            throw new ArgumentException("El nombre no puede estar vacío");
        }

        if (email != null && !EmailValido(email))
        {
            throw new ArgumentException("El formato del email no es válido.");
        }

        this.Nombre = nombre;
        this.Email = email;
        //El Id va a ser establecido por el sistema de persistencia
    }

    protected Alumno() { } // Constructor vacío (lo utilizará EntityFramework)

    private static bool EmailValido(string email)
    {
        // Una validación de email real puede ser más compleja
        return email.Contains('@') && email.Contains('.');
    }

    // Otros métodos de la entidad Alumno que implementan su comportamiento
    // permitiendo cambiar su estado y mantener su propia consistencia
    // es decir, las invariantes de la entidad (reglas que siempre deben ser verdaderas).
}
```

Establecemos sólo las propiedades **Nombre** y **Email**.
El **Id** será generado por el sistema de persistencia

```
namespace Escuela;  
  
public class Alumno  
{  
    public int Id { get; private set; }  
    public string Nombre { get; private set; } = "";  
    public string? Email { get; private set; }
```

NOTA

Desde una perspectiva purista de **Clean Architecture**, que la **identidad de una entidad** dependa del sistema de persistencia constituye una "fuga" de detalles de **infraestructura** hacia el **dominio**. El ideal de **Clean Architecture** es que el núcleo del dominio sea completamente independiente de la **base de datos**, **UI**, **frameworks**, etc.

Si una entidad solo obtiene su **Id** después de ser guardada en la base de datos, **no está "completa"** o **"totalmente identificada"** en memoria antes de la persistencia. Esto puede complicar escenarios donde necesitas el **Id** antes de guardar (por ejemplo, para establecer relaciones con otras entidades).

Establecemos sólo las
Email.
el sistema


```
namespace Escuela;

public class Alumno
{
    public int Id { get; private set; }
    public string Nombre { get; private set; } = "";
    public string? Email { get; private set; }
```

NOTA

Para una adherencia estricta a **Clean Architecture** la entidad debería generar su **Id** en su constructor (o mediante una factory).

Una posible alternativa es utilizar el tipo **Guid** que podemos considerar **únicos a nivel global** (128 bits).

```
public class Alumno
{
    public Guid Id { get; private set; }
    ...
    public Alumno(string nombre)
    {
        this.Id = Guid.NewGuid(); // Generado por la entidad
        ...
    }
    ...
}
```

Establecemos sólo las
Email.
el sistema

```
namespace Escuela;  
  
public class Alumno  
{  
    public int Id { get; private set; }  
    public string Nombre { get; private set; } = "";  
    public string? Email { get; private set; }
```

Trade-off

Utilizar **Guid** en lugar de **int** puede afectar el rendimiento de la base de datos.

Si el rendimiento de la base de datos es crítico, y la "impureza" de que el **Id** se establezca post-persistencia es aceptable (lo cual es común): Los **Ids** generados por la persistencia siguen siendo una opción válida.

Sin embargo, para muchos proyectos que aspiran a una arquitectura limpia, la balanza se inclina hacia los **Guids**, especialmente si se mitigan los problemas de fragmentación de índices que suelen ocasionar en la base de datos.

Estableceremos sólo las
Email.
el sistema

```
namespace Escuela;

public class Examen
{
    public int Id { get; private set; }
    public int AlumnoId { get; private set; }
    public string Materia { get; private set; } = "";
    public double Nota { get; private set; }
    public DateTime Fecha { get; private set; }
    public Examen(int alumnoId, string materia, double nota, DateTime fecha)
    {
        // completar aquí las validaciones que aseguren la consistencia de la entidad
        AlumnoId = alumnoId;
        Materia = materia;
        Nota = nota;
        Fecha = fecha;
    }

    protected Examen() { } // Constructor para EF Core

    public void CambiarNota(double nota)
    {
        if (nota < 0 || nota > 10)
        {
            throw new ArgumentOutOfRangeException("Valor para la nota no permitido");
        }
        Nota = nota;
    }

    // Otros métodos que implementan el comportamiento de la entidad
    // manteniendo sus invariantes
}
```



Codificar la clases de entidad Examen



Copiar el código del archivo
10_RecursosParaLaTeoria



Codificar la clase EscuelaContext



```
using Microsoft.EntityFrameworkCore;

namespace Escuela;

public class EscuelaContext : DbContext
{

    public DbSet<Alumno> Alumnos { get; set; }
    public DbSet<Examen> Exámenes { get; set; }

    protected override void OnConfiguring(DbContextOptionsBuilder
        optionsBuilder)
    {
        optionsBuilder.UseSqlite("data source=Escuela.sqlite");
    }
}
```

Esta clase debe heredar de `DbContext` e invalidar el método `OnConfiguring` que se utiliza para identificar la fuente de datos

Copiar el código del archivo
10_RecursosParaLaTeoria

El nombre de estas propiedades coincide con el nombre de las tablas en la base de datos



Codificar la clase EscuelaContext



```
using Microsoft.EntityFrameworkCore;
```

```
namespace Escuela;
```

```
public class EscuelaContext : DbContext
{
    #nullable disable
    public DbSet<Alumno> Alumnos { get; set; }
    public DbSet<Examen> Exámenes { get; set; }
    #nullable restore
```

Las propiedades `Alumnos` y `Exámenes` serán inicializadas por `Entity Framework`. Si el compilador arroja **warnings** (versiones anteriores) podemos deshabilitar el contexto nullable en esta sección

```
protected override void OnConfiguring(DbContextOptionsBuilder
    optionsBuilder)
{
    optionsBuilder.UseSqlite("data source=Escuela.sqlite");
}
}
```

Invalidando `OnConfiguring` podemos establecer el motor de la base de datos y el archivo



Codificar Program.cs de la siguiente manera



```
using Escuela;

using (var context = new EscuelaContext())
{
    Console.WriteLine("-- Tabla Alumnos --");
    foreach (var a in context.Alumnos)
    {
        Console.WriteLine($"{a.Id} {a.Nombre}");
    }

    Console.WriteLine("-- Tabla Exámenes --");
    foreach (var ex in context.Examenes)
    {
        Console.WriteLine($"{ex.Id} {ex.Materia} {ex.Nota}");
    }
}
```

Copiar el código del archivo
10_RecursosParaLaTeoria

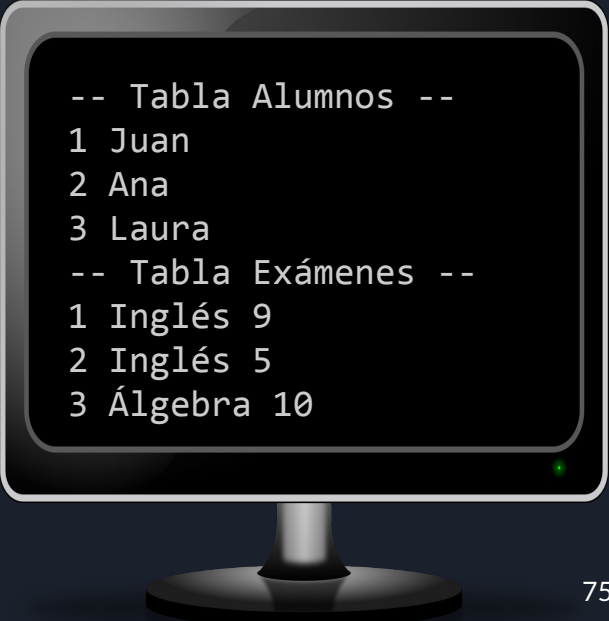
```
using Escuela;

using (var context = new EscuelaContext())
{
    Console.WriteLine("-- Tabla Alumnos --");
    foreach (var a in context.Alumnos)
    {
        Console.WriteLine($"{a.Id} {a.Nombre}");
    }

    Console.WriteLine("-- Tabla Exámenes --");
    foreach (var ex in context.Examenes)
    {
        Console.WriteLine($"{ex.Id} {ex.Materia} {ex.Nota}");
    }
}
```

Un objeto `DbContext` está diseñado para usarse durante una única unidad de trabajo, por lo que la duración de una instancia de un `DbContext` suele ser muy breve.

Es importante invocar al método `Dispose()` al terminar de usar el `DbContext` (en este caso lo realiza la instrucción `using`)



```
-- Tabla Alumnos --
1 Juan
2 Ana
3 Laura
-- Tabla Exámenes --
1 Inglés 9
2 Inglés 5
3 Álgebra 10
```

Entity Framework Core - Code First

Si empezamos un proyecto nuevo, para el cual no existe una base de datos creada, podemos crearla fácilmente a partir del código C# que ya tenemos codificado. A esta estrategia se la conoce con el nombre de “Code First”





Codificar la siguiente clase que usaremos para crear e inicializar la base de datos desde cero



```
namespace Escuela;

public class EscuelaSqlite
{
    public static void Inicializar()
    {
        using var context = new EscuelaContext();
        if (context.Database.EnsureCreated())
        {
            Console.WriteLine("Se creó base de datos");
        }
    }
}
```

Podemos utilizar una declaración using en lugar de la instrucción using

Si la base de datos no existe, se crea según el modelo definido en `EscuelaContext` y devuelve `true`

Copiar el código del archivo
10_RecursosParaLaTeoria



Borrar la base de datos **Escuela.sqlite** del proyecto, agregar la línea indicada en Program.cs y ejecutar



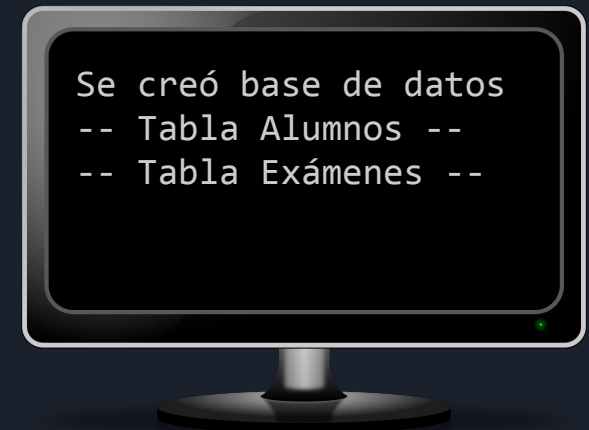
```
using Escuela;
```

```
EscuelaSqlite.Inicializar();
```



```
using (var context = new EscuelaContext())
{
    Console.WriteLine("-- Tabla Alumnos --");
    foreach (var a in context.Alumnos)
    {
        Console.WriteLine($"{a.Id} {a.Nombre}");
    }

    Console.WriteLine("-- Tabla Exámenes --");
    foreach (var ex in context.Examenes)
    {
        Console.WriteLine($"{ex.Id} {ex.Materia} {ex.Nota}");
    }
}
```



Persistencia de datos - SQLite - Entity Framework Core

Table:

Advanced

Fields

Add field Remove field Move field up Move field down

Name	Type	NN	PK	AI	U
Id	INTEGER	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
Nombre	TEXT	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Email	TEXT	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

```
1 CREATE TABLE "Alumnos" (  
2     "Id" INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT,  
3     "Nombre" TEXT NOT NULL,  
4     "Email" TEXT  
5 );
```

Cancel

Table:

Advanced

Fields

Add field Remove field Move field up Move field down

Name	Type	NN	PK	AI	U	Default	Check
Id	INTEGER	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>		
AlumnoId	INTEGER	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>		
Materia	TEXT	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>		
Nota	REAL	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>		
Fecha	TEXT	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>		

```
1 CREATE TABLE "Examenes" (  
2     "Id" INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT,  
3     "AlumnoId" INTEGER NOT NULL,  
4     "Materia" TEXT NOT NULL,  
5     "Nota" REAL NOT NULL,  
6     "Fecha" TEXT NOT NULL  
7 );
```

Cancel OK

Las tablas **Alumnos** y **Examenes** se crearon con estas configuraciones a partir de nuestro modelo utilizando ciertas convenciones por defecto

Table

Alumnos

Advanced

Fields

Add field Remove field Move field up Move field down

Name	Type	NN	PK	AI	U
Id	INTEGER	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
Nombre	TEXT	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Email	TEXT	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

```
1 CREATE TABLE "Alumnos" (  
2     "Id" INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT,  
3     "Nombre" TEXT NOT NULL,  
4     "Email" TEXT  
5 );
```

Table

Exámenes

Advanced

Fields

Add field Remove field Move field up Move field down

Name	Type	NN	PK	AI	U	Default	Check
Id	INTEGER	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>		
AlumnoId	INTEGER	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>		
Materia	TEXT	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>		
Nota	REAL	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>		
Fecha	TEXT	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>		

```
1 CREATE TABLE "Exámenes" (  
2     "Id" INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT,  
3     "AlumnoId" INTEGER NOT NULL,  
4     "Materia" TEXT NOT NULL,  
5     "Nota" REAL NOT NULL,  
6     "Fecha" TEXT NOT NULL  
7 );
```

Cancel OK

Convenciones por defecto utilizadas:

- El **nombre de las tablas** se determinó a partir del nombre de las propiedades `DbSet<>` de `EscuelaContext`
- La propiedad `Id` de las entidades se establecen como **claves** de la tablas
- Si la propiedad `Id` de una entidad es **entera**, se establece como **clave autoincremental**
- Sqlite soporta pocos tipos de datos, se utiliza un mapeo adecuado, por ejemplo las propiedades `DateTime` de .NET se establecen como campos `TEXT` en las tablas `Sqlite`

Persistencia de datos - SQLite - Entity Framework Core

```
public class EscuelaContext : DbContext
{
    #nullable disable
    public DbSet<Alumno> Alumnos { get; set; }
    public DbSet<Examen> Examenes { get; set; }
    #nullable restore
}
```

Se podría invalidar **OnModelCreating** para especificar distintos aspectos sobre cómo se realiza el mapeo entre el modelo y la base de datos utilizando Fluent API (es sólo ilustrativo, no codificarlo para seguir estas diapositivas)

```
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
{
    optionsBuilder.UseSqlite("data source=Escuela.sqlite");
}
```

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Examen>().ToTable("Parciales");
    modelBuilder.Entity<Alumno>()
        .Property(a => a.Email)
        .HasColumnName("Mail").HasDefaultValue("no especificado");
}
```

Name	Type	Schema
Tables (3)		
Alumnos		
Id	INTEGER	"Id" INTEGER NOT NULL PRIMARY KEY
Nombre	TEXT	"Nombre" TEXT NOT NULL
Mail	TEXT	"Mail" TEXT DEFAULT 'no especificado'
Parciales		CREATE TABLE "Parciales" ("Id" INTEGER
sqlite_sequence		CREATE TABLE sqlite_sequence(name,seq
Indices (0)		

Persistencia de datos - SQLite - Entity Framework Core

The screenshot shows the Visual Studio Code interface with the 'Escuela.sqlite - Escuela - Visual Studio Code' window. The 'EXTENSIONS: MARKETPLACE' sidebar is open, displaying a list of extensions. The 'SQLite Viewer' extension by Florian Klampfer is highlighted with a yellow box. A yellow arrow points from this extension to a blue callout box on the right. The callout box contains the text: 'Para visualizar el contenido de la base de datos en Visual Studio Code instalar alguna extensión para SQLite (por ejemplo SQLite Viewer)'. The main editor area shows the 'Escuela.sqlite' file with a table view displaying columns: Id, Nombre, and Email. The table has 3 records: 'Alumnos', 'sqlite_sequence', and 'Examenes'.

Escuela.sqlite - Escuela - Visual Studio Code

File Edit Selection View Go Run Terminal Help

EXTENSIONS: MARKETPLACE

sqlite

SQLite Viewer 3ms
SQLite Viewer for VSCode
Florian Klampfer

SQLite & MySQL Snippets
A snippet for MySQL and SQLite...
Rohit Chouhan
Install

SQLTools
Database management done right...
Matheus Teixeira
Install

Flutter ORM M8 Generator
Vs Code generator for orm-m8 f...
Mircea-Tiberiu MATEI
Install

dBizzy
ER Diagram Visualizer
dBizzy
Install

GrayCat SQL Formatter
A VSCode formatter extension f...
Adam Rybak
Install

MySQL
Database manager for MySQL/...
cweijan
Install

Database Client
Database manager for MySQL/...

Escuela.sqlite

Program.cs

Search tables.. Reset Filters Records: 0 Search 0 records...

Tables (3)

Alumnos

Id

Nombre

Email

sqlite_sequence

Examenes

Id

Nombre

Email

Search column...

Search column...

Search column...

Para visualizar el contenido de la base de datos en Visual Studio Code instalar alguna extensión para SQLite (por ejemplo SQLite Viewer)

Page 1 / 1 Try SQLite Viewer Web

.NET Core Launch (console) (Escuela)

Escuela

Persistencia de datos - SQLite - Entity Framework Core

Escuela.sqlite - Escuela - Visual Studio Code

File Edit Selection View Go Run Terminal Help

EXPLORER

- ESCUELA
 - .vscode
 - bin
 - obj
 - Alumno.cs
 - Escuela.csproj
 - Escuela.sqlite
 - EscuelaContext.cs
 - EscuelaIni.cs
 - Examen.cs
 - Program.cs

Escuela.sqlite

Search tables... Reset Filters Records: 0 Search 0 records...

Tables (3)

- Alumnos
 - Id
 - Nombre
 - Email
- sqlite_sequence
- Examenes

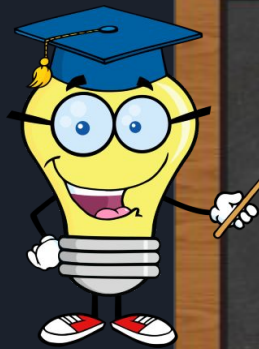
Visualizando la base de datos Escuela.sqlite

En Escuela.sqlite podemos ver las tablas Alumnos y Examenes creadas de acuerdo al modelo definido

Try SQLite Viewer Web

.NET Core Launch (console) (Escuela) Escuela

Entity Framework Core - Code First



Existe un mecanismo más sofisticado para implementar “Code First” que se llama migraciones.

Con las **migraciones** es posible realizar modificaciones incrementales en la base de datos e incluso volver para atrás removiendo la última migración

Para utilizarlas es necesario instalar las herramientas de EF necesarias



Modificar la clase `EscuelaSqlite` para que al crear la base agregue algunos alumnos y exámenes



```
namespace Escuela;

public class EscuelaSqlite
{
    public static void Inicializar()
    {
        using var context = new EscuelaContext();
        if (context.Database.EnsureCreated())
        {
            Console.WriteLine("Se creó base de datos");

            context.Alumnos.Add(new Alumno("Juan", "juan@gmail.com"));
            context.Alumnos.Add(new Alumno("Ana"));
            context.Alumnos.Add(new Alumno("Laura"));

            context.Examenes.Add(new Examen(2, "Inglés", 9, new DateTime(2022, 4, 4)));
            context.Examenes.Add(new Examen(1, "Inglés", 5, new DateTime(2019, 3, 1)));
            context.Examenes.Add(new Examen(1, "Álgebra", 10, new DateTime(2021, 5, 24)));

            context.SaveChanges();
        }
    }
}
```

Copiar el código del archivo
10_RecursosParaLaTeoria



Modificar la clase `EscuelaSqlite` para que al crear la base agregue algunos alumnos y exámenes



```
namespace Escuela;

public class EscuelaSqlite
{
    public static void Inicializar()
    {
        using var context = new EscuelaContext();
        if (context.Database.EnsureCreated())
        {
            Console.WriteLine("Se creó base de datos");

            context.Add(new Alumno("Juan", "juan@gmail.com"));
            context.Add(new Alumno("Ana"));
            context.Add(new Alumno("Laura"));

            context.Add(new Examen(2, "Inglés", 9, new DateTime(2022, 4, 4)));
            context.Add(new Examen(1, "Inglés", 5, new DateTime(2019, 3, 1)));
            context.Add(new Examen(1, "Álgebra", 10, new DateTime(2021, 5, 24)));

            context.SaveChanges();
        }
    }
}
```

Esta instrucción actualiza en la base de datos todos los cambios realizados (en este caso el alta de los alumnos y exámenes realizados)

Observar que podemos no especificar la propiedad `DbSet<>` donde agregar un `Alumno` o un `Examen`, porque existe una única propiedad `DbSet<>` por cada entidad.



Modificar la clase **EscuelaSqlite** para que al crear la base agregue algunos alumnos y exámenes



```
Se creó base de datos
-- Tabla Alumnos --
1 Juan
2 Ana
3 Laura
-- Tabla Exámenes --
1 Inglés 9
2 Inglés 5
3 Álgebra 10
```

Persistencia de datos - SQLite - Entity Framework Core

Database Structure Browse Data Edit Pragmas Execute SQL

Table: Alumnos ↺ ↻ 🔍 📄 🖨 New Record Delete Record

	Id	Nombre	Email
	Filter	Filter	Filter
1	1	Juan	juan@gmail..
2	2	Ana	NULL
3	3	Laura	NULL

⏪ ⏩ 1 - 3 of 3 ⏪ ⏩ Go to:

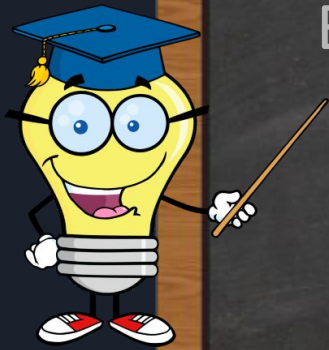
Database Structure Browse Data Edit Pragmas Execute SQL

Table: Exámenes ↺ ↻ 🔍 📄 🖨 New Record Delete Record

	Id	Alumnoid	Materia	Nota	Fecha
	Filter	Filter	Filter	Filter	Filter
1	1	2	Inglés	9.0	2022-04-04 ...
2	2	1	Inglés	5.0	2019-03-01 ...
3	3	1	Álgebra	10.0	2021-05-24 ...

⏪ ⏩ 1 - 3 of 3 ⏪ ⏩ Go to: 1

Entity Framework Core - Code First



Está claro que existe una relación “uno a muchos” entre Alumnos y Exámenes

Alumnos

Nombre	Id
Juan	1
Ana	2
Laura	3

Exámenes

Alumnoid	Materia	Nota	Id
2	Inglés	9	1
1	Inglés	5	2
1	Álgebra	10	3

Por cada alumno podemos tener 0, 1 o más exámenes



Modificar Program.cs y ejecutar

```
using Escuela;
```

```
EscuelaSqlite.Inicializar(); //solo tiene efecto si  
                             // la base de datos no existe
```

```
using var context = new EscuelaContext();  
var query = context.Alumnos.Join(context.Examenes,  
    a => a.Id,  
    e => e.AlumnoId,  
    (a, e) => new  
    {  
        Alumno = a.Nombre,  
        Materia = e.Materia,  
        Nota = e.Nota  
    });
```

```
foreach (var obj in query)  
{  
    Console.WriteLine(obj);  
}
```

Consultando
datos



Utilizamos LINQ
para realizar
consultas

Copiar el código del archivo
10_RecursosParaLaTeoria

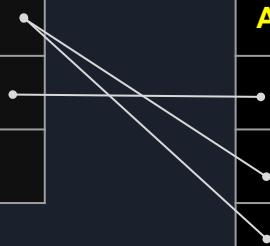
```
...  
  
var query = context.Alumnos.Join(context.Examenes,  
    a => a.Id,  
    e => e.AlumnoId,  
    (a, e) => new  
    {  
        Alumno = a.Nombre,  
        Materia = e.Materia,  
        Nota = e.Nota  
    });  
foreach (var obj in query)  
{  
    Console.WriteLine(obj);  
}
```

Alumnos

Nombre	Id
Juan	1
Ana	2
Laura	3

Exámenes

AlumnoId	Materia	Nota	Id
2	Inglés	9	1
1	Inglés	5	2
1	Álgebra	10	3



```
{ Alumno = Ana, Materia = Inglés, Nota = 9 }  
{ Alumno = Juan, Materia = Inglés, Nota = 5 }  
{ Alumno = Juan, Materia = Álgebra, Nota = 10 }
```

```
...  
  
var query = context.Alumnos.Join(context.Examenes,  
    a => a.Id,  
    e => e.AlumnoId,  
    (a, e) => new  
    {  
        Alumno = a.Nombre,  
        Materia = e.Materia,  
        Nota = e.Nota  
    });  
foreach (var obj in query)  
{  
    Console.WriteLine(obj);  
}
```

Alumnos

Nombre	Id
Juan	1
Ana	2
Laura	3

Exámenes

AlumnoId	Materia	Nota	Id
2	Inglés	9	1
1	Inglés	5	2
1	Álgebra	10	3

Para los que conocen SQL

Esta es la instrucción SQL que generó Entity Framework para recuperar los datos desde la base

```
SELECT "a"."Nombre" AS "Alumno", "e"."Materia", "e"."Nota"  
FROM "Alumnos" AS "a"  
INNER JOIN "Exámenes" AS "e" ON "a"."Id" = "e"."AlumnoId"
```

La consulta a la base de datos se realiza recién al ejecutar el `foreach`



Modificar Program.cs y ejecutar



```
using Escuela;

EscuelaSqlite.Inicializar(); //solo tiene efecto si
                             // la base de datos no existe

using var context = new EscuelaContext();

//Agregamos un nuevo alumno
var alumno = new Alumno("Pablo"); // el Id será establecido por SQLite
context.Add(alumno); // se agregará realmente con el db.SaveChanges()

context.SaveChanges(); // actualiza la base de datos.
                       // SQLite establece el valor de alumno.Id

// Agregamos un examen para el nuevo alumno
var examen = new Examen(alumno.Id, "Historia", 9.5, DateTime.Today);
context.Add(examen);

context.SaveChanges();
```

Agregando
registros

Copiar el código del archivo
10_RecursosParaLaTeoria

Estado de las tablas luego de ejecutar el código

```
using Escuela;
```

```
EscuelaSqlite.Inicializar(); //solo tiene efecto si
                             // la base de datos no existe
```

```
using var context
```

```
//Agregamos un nue
var alumno = new A
context.Add(alumno
```

```
context.SaveChanges
```

```
// Agregamos un exa
var examen = new Examen(alumno.Id, "Historia", 9.5, DateTime.Today);
context.Add(examen);
```

```
context.SaveChanges();
```

Así queda la base de datos

Alumnos		Exámenes			
Nombre	Id	Alumnoid	Materia	Nota	Id
Juan	1				
Ana	2	2	Inglés	9	1
Laura	3	1	Inglés	5	2
Pablo	4	1	Álgebra	10	3
		4	Historia	9.5	4



Modificar Program.cs y ejecutar



```
using Escuela;
```

```
EscuelaSqlite.Inicializar(); //solo tiene efecto si  
                             // la base de datos no existe
```

```
using var context = new EscuelaContext();
```

```
//borramos de la tabla Alumnos el registro con Id=3
```

```
var alumnoBorrar = context.Alumnos.Where(a => a.Id == 3).SingleOrDefault();
```

```
if (alumnoBorrar != null)
```

```
{
```

```
    context.Remove(alumnoBorrar); //se borra realmente con el context.SaveChanges()
```

```
}
```

```
//La nota en Inglés del alumno id=1 es un 5. La cambiamos a 7.5
```

```
var examenModificar = context.Examenes.Where(
```

```
    e => e.AlumnoId == 1 && e.Materia == "Inglés").SingleOrDefault();
```

```
if (examenModificar != null)
```

```
{
```

```
    examenModificar.CambiarNota(7.5); //se modifica el registro en memoria
```

```
}
```

```
context.SaveChanges(); //actualiza la base de datos.
```

Borrando y
actualizando
registros

Copiar el código del archivo
10_RecursosParaLaTeoria

Modificar Program.cs y ejecutar

```
using Escuela;
```

```
EscuelaSqlite.Inicializar(); //solo tiene efecto si
                             // la base de datos no existe
```

```
using var context = new EscuelaContext();
```

```
//borramos de la tabla Alumnos el registro con Id=3
```

```
var alumnoBorrar = context.Alumnos.Where(a => a.Id == 3).SingleOrDefault();
```

```
if (alumnoBorrar != null)
```

```
{
```

```
context.Remove(alumnoBorrar);
```

```
}
```

```
//La nota de Juan en Inglés es 7.5
```

```
var examen = context.Examenes.FirstOrDefault(a => a.Alumnoid == 1 & a.Materia == "Inglés");
```

```
if (examen != null)
```

```
{
```

```
examen.Nota = 7.5;
```

```
}
```

```
context.SaveChanges(); //actualiza la base de datos.
```

Devuelve el único elemento de la secuencia. Si no hay ninguno devuelve el valor por defecto (null en este caso). Si no es único lanza una excepción

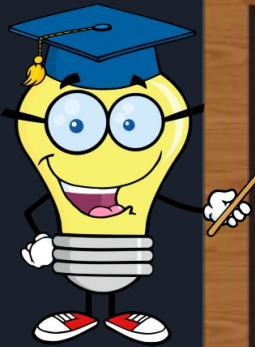
Así queda la base de datos

Alumnos	
Nombre	Id
Juan	1
Ana	2
Pablo	4

Exámenes

Alumnoid	Materia	Nota	Id
2	Inglés	9	1
1	Inglés	7.5	2
1	Álgebra	10	3
4	Historia	9.5	4

Entity Framework Core - Code First



EF Core facilita la navegación entre entidades relacionadas por medio de las propiedades de navegación

Alumnos		Exámenes			
Nombre	Id	Alumnoid	Materia	Nota	Id
Juan	1	2	Inglés	9	1
Ana	2	1	Inglés	7.5	2
Pablo	4	1	Álgebra	10	3
		4	Historia	9.5	4

Sería conveniente tener una propiedad “Exámenes” en la entidad **Alumno** que me permitiera acceder a la lista de exámenes de cada alumno



Modificar la clase Alumno



```
public class Alumno
{
    public int Id { get; private set; }
    public string Nombre { get; private set; } = "";
    public string? Email { get; private set; }

    public List<Examen>? Examenes { get; set; }

    public Alumno(string nombre, string? email = null)
    {
        if (string.IsNullOrEmpty(nombre))
        {
            throw new ArgumentException("El nombre no puede ser nulo ni estar vacío");
        }

        // Validación de Email si se proporciona (ejemplo básico)
        if (email != null && !EmailValido(email))
        {
            throw new ArgumentException("El formato del email no es válido.", nameof(email));
        }
    }
    . . .
    . . .
}
```

Propiedad de navegación



Modificar Program.cs y ejecutar



```
using Microsoft.EntityFrameworkCore;  
using Escuela;
```

```
EscuelaSqlite.Inicializar(); //solo tiene efecto si  
                             // la base de datos no existe
```

```
using var context = new EscuelaContext();
```

```
foreach (Alumno a in context.Alumnos.Include(a => a.Examenes))  
{  
    Console.WriteLine(a.Nombre);  
    a.Examenes?.ToList()  
        .ForEach(ex => Console.WriteLine($" - {ex.Materia} {ex.Nota}"));  
}
```

Es necesario incluir
explícitamente los
datos relacionados

Copiar el código del archivo
10_RecursosParaLaTeoria



Modificar Program.cs y ejecutar




```
using Microsoft.EntityFrameworkCore;  
using Escuela;
```

```
EscuelaSqlite.Inicializar(); //solo tiene efecto si  
                             // la base de datos no existe
```

```
using var context = new EscuelaContext();
```

```
foreach (Alumno a in context.Alumnos.Include(a => a.Examenes))  
{  
    Console.WriteLine(a.Nombre);  
    a.Examenes?.ToList()  
        .ForEach(ex => Console.WriteLine($" - {ex.Materia} {ex.Nota}"));  
}
```

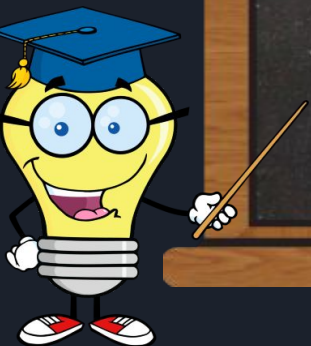


```
Juan  
- Inglés 7,5  
- Álgebra 10  
Ana  
- Inglés 9  
Pablo  
- Historia 9,5
```

Entity Framework Core - Code First

Las propiedades de navegación, también facilitan el alta de información en las tablas relacionadas de la base de datos

Podemos agregar un nuevo alumno con la información de sus exámenes en una única operación `SaveChanges()`





Modificar Program.cs y ejecutar



```
using Microsoft.EntityFrameworkCore;  
using Escuela;
```

```
EscuelaSqlite.Inicializar(); //solo tiene efecto si  
                             // la base de datos no existe
```

```
using var context = new EscuelaContext();
```

```
Alumno nuevo = new Alumno("Andrés");  
nuevo.Examenes = new List<Examen>() { //dejamos en 0 el AlumnoId porque va a ser ignorado  
    new Examen(0, "Lengua", 7, DateTime.Parse("5/5/2022")),  
    new Examen(0, "Matemática", 6, DateTime.Parse("6/5/2022"))  
};  
context.Add(nuevo);  
context.SaveChanges();
```

Se crea un alumno con su lista de exámenes y se salva de una sola vez

```
foreach (Alumno a in context.Alumnos.Include(a => a.Examenes))  
{  
    Console.WriteLine(a.Nombre);  
    a.Examenes?.ToList()  
        .ForEach(ex => Console.WriteLine($" - {ex.Materia} {ex.Nota}"));  
}
```

Copiar el código del archivo
10_RecursosParaLaTeoria



Modificar Program.cs y ejecutar

```
using Microsoft.EntityFrameworkCore;
using Escuela;

EscuelaSqlite.Inicializar(); //solo tiene efecto si
                             // la base de datos no existe
```

```
using y
```

```
Alumno
nuevo
```

```
};
```

```
context
```

```
context
```

```
foreach
```

```
{
```

```
Con
```

```
a.Ex
```

```
.ForEach(ex > console.WriteLine($" {ex.Nombre} {ex.Id} {ex.Nota} "));
```

```
}
```

Así queda la base de datos

Alumnos		Exámenes			
Nombre	Id	Alumnoid	Materia	Nota	Id
Juan	1	2	Inglés	9	1
Ana	2	1	Inglés	7.5	2
Pablo	4	1	Álgebra	10	3
Andrés	5	4	Historia	9.5	4
		5	Lengua	7	5
		5	Matemática	6	6

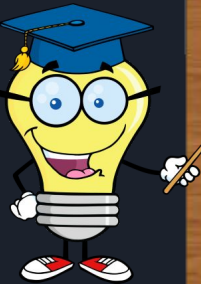
Juan
- Inglés 7,5
- Álgebra 10
Ana
- Inglés 9
Pablo
- Historia 9,5
Andrés
- Lengua 7
- Matemática 6

Integridad referencial en la base de datos

Importante

Las propiedades de navegación también influyen en el comportamiento por defecto al momento de crear la base de datos.

Si creamos nuevamente la base de datos, eliminándola y volviendo a ejecutar `EscuelaSqlite.Inicializar()`; notaremos una diferencia importante.



Integridad referencial en la base de datos

Código SQL generado por Entity Framework para crear la tabla Exámenes sin la propiedad de navegación:

```
CREATE TABLE "Exámenes" (  
    "Id" INTEGER NOT NULL CONSTRAINT "PK_Exámenes" PRIMARY KEY AUTOINCREMENT,  
    "AlumnoId" INTEGER NOT NULL,  
    "Materia" TEXT NOT NULL,  
    "Nota" REAL NOT NULL,  
    "Fecha" TEXT NOT NULL  
)
```

Con la propiedad de navegación:

```
CREATE TABLE "Exámenes" (  
    "Id" INTEGER NOT NULL CONSTRAINT "PK_Exámenes" PRIMARY KEY AUTOINCREMENT,  
    "AlumnoId" INTEGER NOT NULL,  
    "Materia" TEXT NOT NULL,  
    "Nota" REAL NOT NULL,  
    "Fecha" TEXT NOT NULL,  
    CONSTRAINT "FK_Exámenes_Alumnos_AlumnoId" FOREIGN KEY ("AlumnoId")  
        REFERENCES "Alumnos" ("Id") ON DELETE CASCADE  
)
```





Integridad referencial en la base de datos

```
CONSTRAINT "FK_Examenes_Alumnos_AlumnoId" FOREIGN KEY ("AlumnoId")  
REFERENCES "Alumnos" ("Id") ON DELETE CASCADE
```

Esta instrucción establece una relación de integridad referencial entre la tabla **Exámenes** y la tabla **Alumnos**.

Los valores en la columna **AlumnoId** deben coincidir con valores existentes en la columna **Id** de la tabla **Alumnos**.

La opción **ON DELETE CASCADE** para indicar que si se elimina un registro en la tabla **Alumnos**, también se eliminarán automáticamente los registros asociados en la tabla **Exámenes**.

Fin de Teoría 10

Práctica sobre la teoría 10

Práctica sobre la teoría 10

1) Utilizando el método **Range** de la clase **System.Linq.Enumerable** y los métodos de **LINQ** que sean necesarios, obtener:

- a) Lista con todos los múltiplos de 5 entre 100 y 200
- b) Lista con todos los números primos menores que 100
- c) Lista con las potencias de 2, desde 2^0 a 2^{10}
- d) La suma y el promedio de los valores de la lista anterior
- e) Lista de todos los n^2 que terminan con el dígito 6, para n entre 1 y 20
- f) Lista con los nombres de los días de la semana en inglés que contengan una letra 'u' (tip: utilizar el enumerativo **DayOfWeek**)

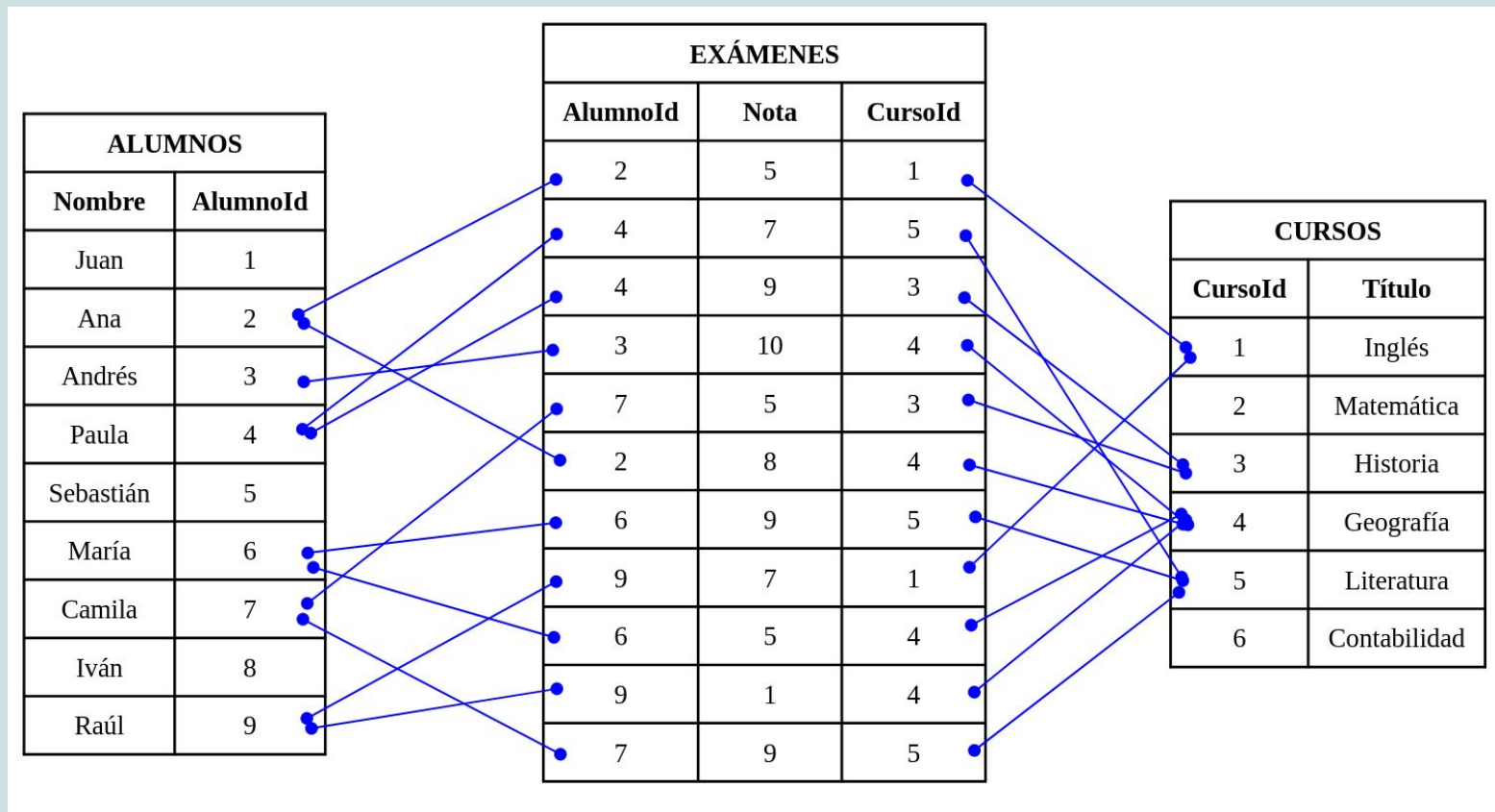
2) Listar por consola la cantidad de veces que se repiten los elementos de un vector de enteros. Ordenar por cantidad de repeticiones. Completar el siguiente código para que la salida por consola sea la indicada

```
int[] vector = [1, 3, 4, 5, 9, 4, 3, 4, 5, 1, 1, 4, 9, 4, 3, 1];  
vector.GroupBy(n => n)  
    // . . . completar aquí las líneas que faltan usando fluent API  
    .ForEach(obj => Console.WriteLine(obj));
```

Salida por consola

```
{ Numero = 5, Cantidad = 2 }  
{ Numero = 9, Cantidad = 2 }  
{ Numero = 3, Cantidad = 3 }  
{ Numero = 1, Cantidad = 4 }  
{ Numero = 4, Cantidad = 5 }
```

3) Supongamos que tenemos la siguiente información sobre alumnos, cursos y exámenes estructurada de esta manera:



Podemos ver, por ejemplo, que Ana sacó 5 en un examen de Inglés y 8 en un examen de Geografía, Juan no rindió ningún examen y nadie rindió examen en el curso de Matemáticas.

Definir las clases **Alumno**, **Examen** y **Curso**. Establecer por código las listas **alumnos** (de tipo **List<Alumno>**), **exámenes** (de tipo **List<Examen>**) y **cursos** (de tipo **List<Curso>**) con los datos que se muestran en la imagen anterior y resolver utilizando LINQ:

a) Obtener el listado con los nombres de los alumnos que rindieron al menos un examen, ordenado alfabéticamente (tip: puede utilizarse el método de extensión **Distinct()** para obtener una secuencia de elementos no repetidos). La salida debería ser:

Salida por consola

```
Ana  
Andrés  
Camila  
María  
Paula  
Raúl
```

b) Obtener el listado con los cursos donde se hayan rendido exámenes. Se debe listar el título del curso junto con la cantidad de exámenes. El listado debe ordenarse por cantidad de exámenes. La salida debería ser:

Salida por consola

```
{ Título = Inglés, Cantidad = 2 }  
{ Título = Historia, Cantidad = 2 }  
{ Título = Literatura, Cantidad = 3 }  
{ Título = Geografía, Cantidad = 4 }
```

c) Obtener el listado con los alumnos que hayan rendido al menos un examen informando el nombre del alumno, el título del curso y la nota del examen. La salida debería ser:

Salida por consola

```
{ Alumno = Ana, Curso = Inglés, Nota = 5 }  
{ Alumno = Ana, Curso = Geografía, Nota = 8 }  
{ Alumno = Andrés, Curso = Geografía, Nota = 10 }  
{ Alumno = Paula, Curso = Literatura, Nota = 7 }  
{ Alumno = Paula, Curso = Historia, Nota = 9 }  
{ Alumno = María, Curso = Literatura, Nota = 9 }  
{ Alumno = María, Curso = Geografía, Nota = 5 }  
{ Alumno = Camila, Curso = Historia, Nota = 5 }  
{ Alumno = Camila, Curso = Literatura, Nota = 9 }  
{ Alumno = Raúl, Curso = Inglés, Nota = 7 }  
{ Alumno = Raúl, Curso = Geografía, Nota = 1 }
```

d) Filtrar el listado del punto anterior para mostrar sólo los casos aprobados (nota ≥ 6).

e) Obtener el listado con los nombres de los alumnos que no han rendido ningún examen.

f) Obtener el listado de los alumnos que hayan rendido algún examen junto con el promedio de todos sus exámenes. La salida debería ser:

Salida por consola

```
{ Alumno = Ana, Promedio = 6,5 }  
{ Alumno = Andrés, Promedio = 10 }  
{ Alumno = Paula, Promedio = 8 }  
{ Alumno = María, Promedio = 7 }  
{ Alumno = Camila, Promedio = 7 }  
{ Alumno = Raúl, Promedio = 4 }
```


Práctica sobre la teoría 10

4) Se debe realizar un programa que permita las operaciones **CRUD** (**Create**, **Read**, **Update**, **Delete**) para una casa de alquiler de juegos.







Para esto, definir las siguientes tablas desde **DB Browser** o bien utilizar **Code First** para generar la base de datos como se vio en la teoría:

Tabla

Cientes

▼ Avanzado

Campos Restricciones

 Añadir  Eliminar  Mover al principio  Mover hacia arriba  Mover hacia abajo  Mover al final







Nombre	Tipo	NN	PK	AI	U	Por defecto	Check
Id	INTEGER	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>		
DNI	TEXT	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>		
ApellidoYNombre	TEXT	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>		
Direccion	TEXT	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>		
Mail	TEXT	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>		
Telefono	TEXT	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>		

Tabla

Juegos

▼ Avanzado

Campos Restricciones

 Añadir  Eliminar  Mover al principio  Mover hacia arriba  Mover hacia abajo  Mover al final

Nombre	Tipo	NN	PK	AI	U	Por defecto	Check
Id	INTEGER	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>		
Nombre	TEXT	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>		
Descripcion	TEXT	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>		
Estado	TEXT	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>		
PrecioPorDia	REAL	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>		

Práctica sobre la teoría 10

Tabla

Alquileres

▼ Avanzado

Campos Restricciones

📄 Añadir 🗑 Eliminar ⇄ Mover al principio ⬆ Mover hacia arriba ▼ Mover hacia abajo ⇄ Mover al final

Nombre	Tipo	NN	PK	AI	U	Por defecto	Check
Id	INTEGER	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>		
IdCliente	INTEGER	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>		
IdJuego	INTEGER	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>		
Fecha	TEXT	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>		
FechaTentativaDevolucion	TEXT	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>		
FechaDevolucion	TEXT	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>		
CostoTotal	REAL	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>		

Nota: El tipo DateTime no existe en SQLite. Para guardar fechas se puede utilizar TEXT REAL o INTEGER.

Codificar un programa que dado el siguiente código produzca la salida en consola indicada:

```
var juego1 = new Juegos()
{
    Nombre = "Cama Elastica",
    Descripcion = "Medida de 2 x 2",
    Estado = "Bueno",
    PrecioPorDia = 1000
};
var juego2 = new Juegos()
{
    Nombre = "Castillo",
    Descripcion = "Hasta 10 personas",
    Estado = "Nuevo",
    PrecioPorDia = 1200
};
```

Práctica sobre la teoría 10

```

AgregarJuego(juego1);
AgregarJuego(juego2);
ListarJuegos();

var cliente1 = new Clientes()
{
    DNI = "20569784",
    ApellidoYNombre = "Perez, Juan",
    Direccion = "48 e/ 5 y 6 N°520"
};
var cliente2 = new Clientes()
{
    DNI = "10569784",
    ApellidoYNombre = "Gonzalez, Alejandra",
    Direccion = "25 e/ 9 y 10 N°520",
    Mail = "gale@gmail.com",
    Telefono = "221-15-569874"
};
AgregarCliente(cliente1);
AgregarCliente(cliente2);
ListarClientes();

var alquiler1 = new Alquileres()
{
    IdCliente = 1,
    IdJuego = 1,
    Fecha = DateTime.Now
};
var alquiler2 = new Alquileres()
{
    IdCliente = 1,
    IdJuego = 2,
    Fecha = DateTime.Now
};
AgregarAlquiler(alquiler1);
AgregarAlquiler(alquiler2);
ListarAlquileres();

ModificarCliente("10569784", "Gonzalez, Alejandra", "52 e/ 9 y 10 N°520", "gale@gmail.com", "221-15-569874");
ListarClientes();
EliminarCliente("10569784");
ListarClientes();
```

Práctica sobre la teoría 10

```
ModificarJuego(1, "Cama Elastica", "Medida de 2 x 2", "Roto", 1500);  
ListarJuegos();
```

```
ModificarAlquiler(1, 1562.25, new DateTime(2021 / 11 / 12));  
ListarAlquileres();
```

Salida por consola

```
-- Se agregó el juego "Cama Elastica" al que se le asignó el Id 1  
-- Se agregó el juego "Castillo" al que se le asignó el Id 2  
  
-- Listado de Juegos --  
{ Id = 1, Nombre = Cama Elastica, Descripción = Medida de 2 x 2, Estado = Bueno, PrecioPorDia = 1000 }  
{ Id = 2, Nombre = Castillo, Descripción = Hasta 10 personas, Estado = Nuevo, PrecioPorDia = 1200 }  
  
-- Se agregó el cliente "Perez, Juan" al que se le asignó el Id 1  
-- Se agregó el cliente "Gonzalez, Alejandra" al que se le asignó el Id 2  
  
-- Listado de Clientes --  
{ Id = 1, DNI = 20569784, Nombre = Perez, Juan, Dirección = 48 e/ 5 y 6 N°520, Tel = }  
{ Id = 2, DNI = 10569784, Nombre = Gonzalez, Alejandra, Dirección = 25 e/ 9 y 10 N°520, Tel = 221-15-569874 }  
  
-- Se agrego un Alquiler al que se le asignó el id 1  
-- Se agrego un Alquiler al que se le asignó el id 2  
  
-- Listado de Alquileres  
{ Id = 1, Fecha = 03/11/2021, FechaDevolucion = / / , Cliente = Perez, Juan, Juego = Cama Elastica, Monto = 0 }  
{ Id = 2, Fecha = 03/11/2021, FechaDevolucion = / / , Cliente = Perez, Juan, Juego = Castillo, Monto = 0 }  
  
-- Se modificó cliente con dni 10569784 --  
  
-- Listado de Clientes --  
{ Id = 1, DNI = 20569784, Nombre = Perez, Juan, Dirección = 48 e/ 5 y 6 N°520, Tel = }  
{ Id = 2, DNI = 10569784, Nombre = Gonzalez, Alejandra, Dirección = 52 e/ 9 y 10 N°520, Tel = 221-15-569874 }  
  
-- Se eliminó el cliente con DNI = 10569784 --  
  
-- Listado de Clientes --  
{ Id = 1, DNI = 20569784, Nombre = Perez, Juan, Dirección = 48 e/ 5 y 6 N°520, Tel = }  
  
-- Se modificó el juego con id 1 --  
  
-- Listado de Juegos --  
{ Id = 1, Nombre = Cama Elastica, Descripción = Medida de 2 x 2, Estado = Roto, PrecioPorDia = 1500 }  
{ Id = 2, Nombre = Castillo, Descripción = Hasta 10 personas, Estado = Nuevo, PrecioPorDia = 1200 }  
  
-- Se modificó el alquiler id 1 --  
  
-- Listado de Alquileres  
{ Id = 1, Fecha = 03/11/2021, FechaDevolucion = 12/11/2021, Cliente = Perez, Juan, Juego = Cama Elastica, Monto = 1562.25 }  
{ Id = 2, Fecha = 03/11/2021, FechaDevolucion = / / , Cliente = Perez, Juan, Juego = Castillo, Monto = 0 }
```