

NET CORE

martes, 10 de diciembre de 2024 9:07

En este módulo trabajaremos con el software de **Visual Studio Enterprise 2022**

Para poder trabajar con este software necesitamos una clave para activarlo.

Tenemos nuestra clave dentro de **Education** en Azure Portal

<https://portal.azure.com/#home>

Podemos incluir dicha clave en múltiples equipos.

Home > Education

Education | Software

Product language : E

74 Items

Name ↑↓	Product category
SQL Server 2019 Developer	Database
Machine Learning Server 9.4.7 for Wind...	AI + Machine Lear...
Visual Studio Enterprise Edition 2022	Developer Tools
Microsoft R Client 9.4.7	Database
Agents for Visual Studio 2019 (version 1...	Developer Tools
Agents for Visual Studio 2019 (version 1...	Developer Tools
Azure DevOps Server Express 2022.2 (x6...	Productivity Tools

Software

Education

Visual Studio Enterprise Edition 2022

An integrated, end-to-end solution for developers looking for high productivity and seamless coordination across teams of any size.

Operating System
Windows

Product language
Multilanguage

System
64 bit

View Key

Help improve this page

Download **Cancel**

En la instalación, necesitamos los siguientes elementos.

Modificando — Visual Studio Enterprise 2022 — 17.10.3

Cargas de trabajo Componentes individuales Paquetes de idioma Ubicaciones de instalación

Web y nube (4)

- Desarrollo de ASP.NET y web Desarrollo de Azure
- Desarrollo de Python Desarrollo de Node.js

Móviles y de escritorio (5)

- Desarrollo de la interfaz de usuario de aplicaciones mu... Desarrollo de escritorio de .NET
- Desarrollo para el escritorio con C++ Desarrollo de aplicaciones de Windows

Puede agregar paquetes de idioma adicionales para la instalación de Visual Studio.

- Aleman
- Checo (Czech)
- Chino (simplificado)
- Chino (tradicional)
- Coreano
- English
- Español
- Francés
- Italiano
- Japonés
- Polaco
- Portugués (Brasil)
- Russo
- Turco



- Diseñador de clases
- GIT para Windows
- Herramientas de LINQ to SQL

Herramientas de código

- Administrador de paquetes NuGet
- administrador de paquetes vcpkg
- Clon de código
- Destinos y tareas de compilación de NuGet
- Developer Analytics Tools
- Diseñador de clases
- Editor de DGM
- GIT para Windows
- Herramientas de LINQ to SQL
- Integración de Office para Azure DevOps
- Mapa de código
- PreEmptive Protection - Dotfuscator
- Publicación de ClickOnce
- Transformación de plantilla de texto

Trabajaremos con .NET 9.0 que ha aparecido hace 3 semanas.

Dependiendo de la versión de Net Core que utilicemos, a lo mejor, tenemos que bajar A la versión **OFICIAL** para algunas compatibilidades de fabricantes.

Se trabaja con proyectos o soluciones.

Un proyecto es una arquitectura individual como, por ejemplo,
Un proyecto de Consola.

Una solución es un conjunto de proyectos, por ejemplo, un proyecto
De Consola y un proyecto Web.

Ya tenemos integrado Github dentro de Visual Studio.

Existen múltiples lenguajes a utilizar, pero nosotros utilizaremos el Oficial que es **C#**

Lo que vamos a comenzar hoy es lo que se denomina la parte **BACK**

Elementos dentro del módulo

- 1) Fundamentos del lenguaje
- 2) SQL Server (Transact SQL): Vamos a visualizar características Propias de la base de datos SQL Server, como su lenguaje.
- 3) ADO .NET: Es el acceso a datos de Visual Studio.

¿Qué es Net Core? Es la tecnología multiplataforma de Microsoft.

La versión de Net Core importa. Dependiendo de la versión, Los fabricantes de terceros tienen que ir sacando sus librerías Compatibles.

A diferencia de Front, todos los proyectos de Net Core ya generan Su producto final que es un **Assembly**. Es decir, están listos Para producción.

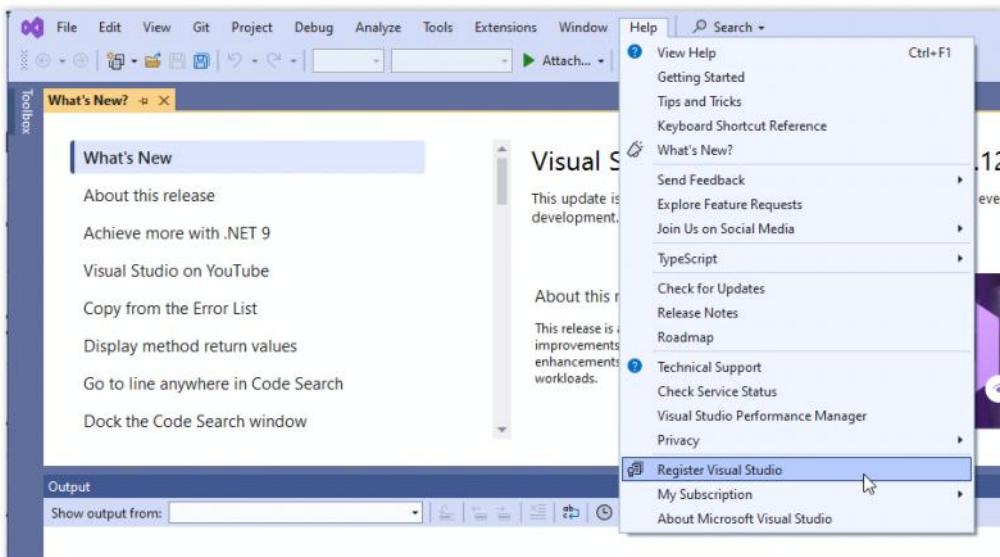
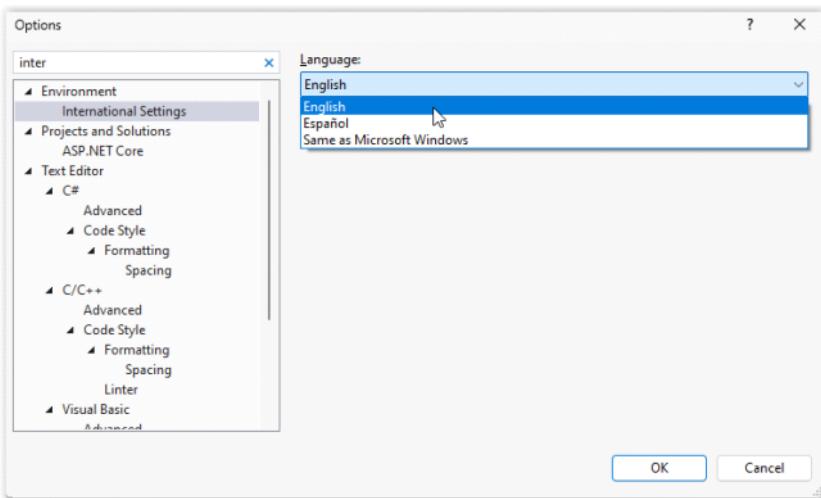
Existe una tecnología llamada **Xamarin** o la nueva versión **Maui** que Permite programar para dispositivos móviles.

Todas las pantallas están unificadas de igual forma, es decir, creamos Un proyecto común y lo compilamos para Android o lo compilamos Para iOS.

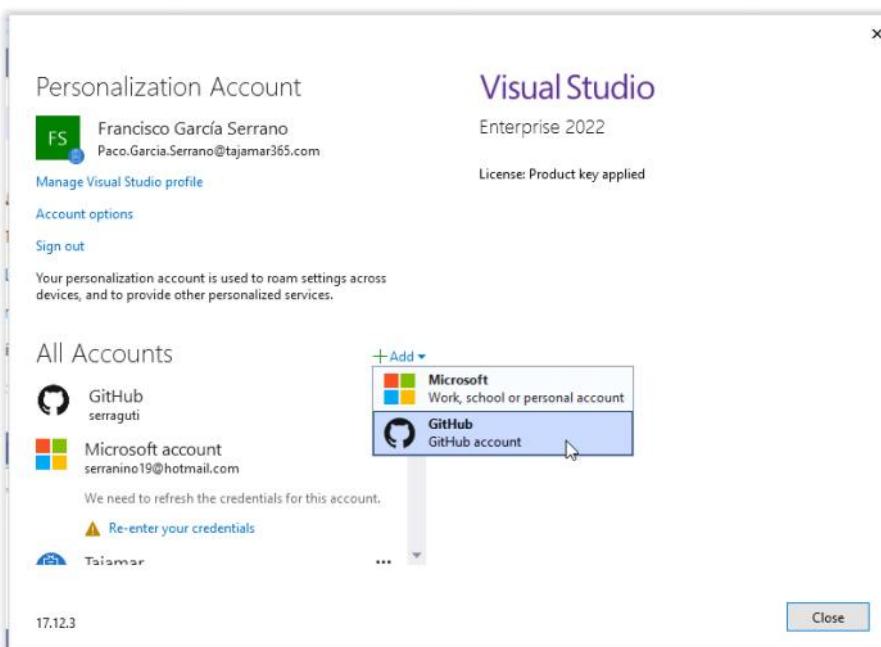
Todos los proyectos Net Core tendrán una serie de dependencias Llamadas **Nuget**

Abrimos Visual Studio 2022

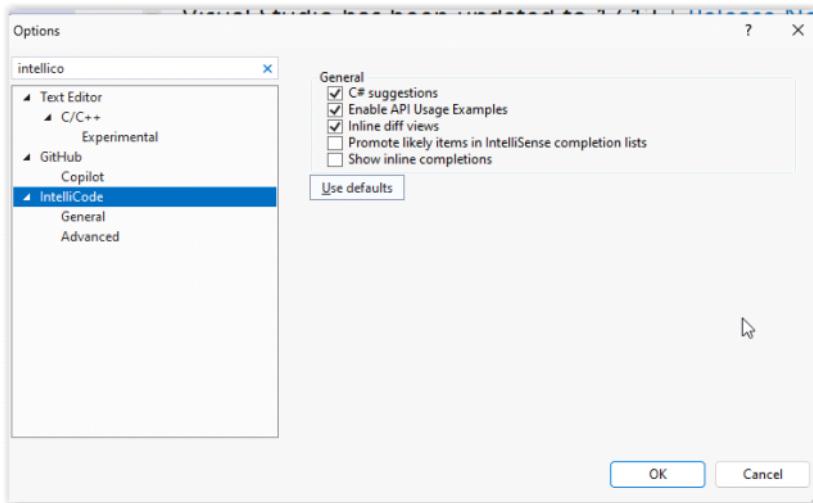
Cambiamos el idioma a inglés



Incluimos nuestra cuenta de Github



Tenemos una característica llamada **IntelliCode** que nos autogenera código AI escribir.
Si en algún momento, queréis quitarlo, ponéis las opciones que vemos en la imagen.

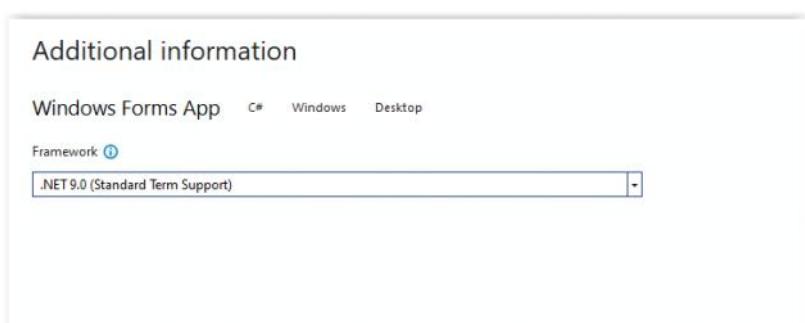
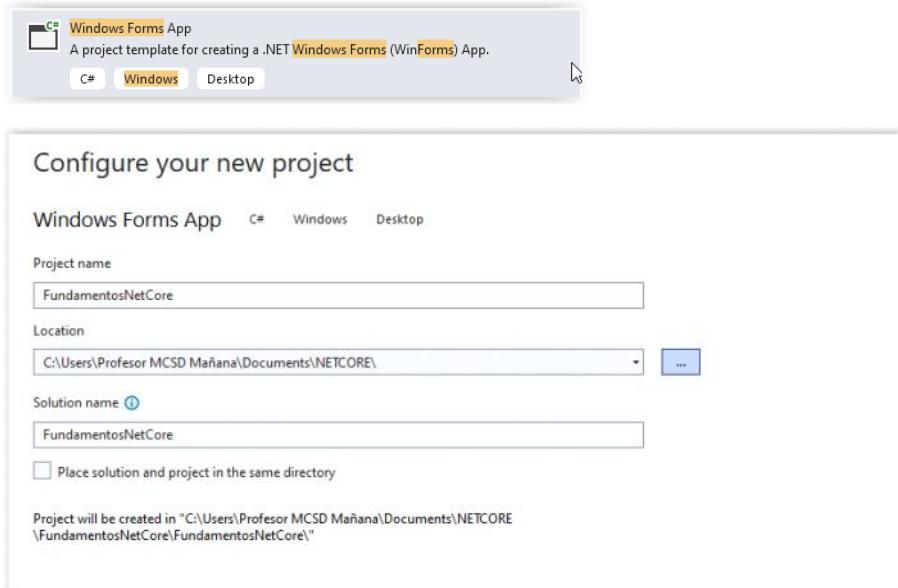


Tenemos multitud de proyectos.

Metodología de trabajo

- Tendremos un único proyecto e iremos incluyendo múltiples Funcionalidades.
- Vamos a comenzar con aplicaciones gráficas de tipo Windows Forms. No utilizo esto para aprender formularios, Es simplemente porque es más sencillo avanzar.

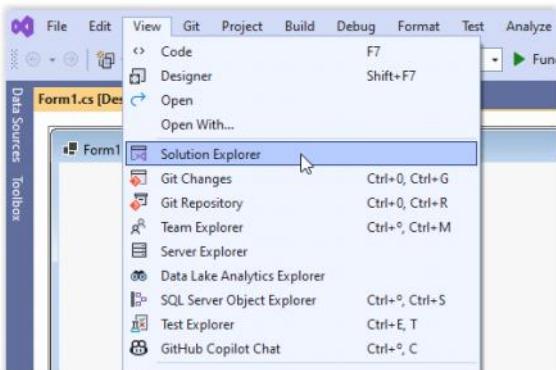
Comenzamos creando un nuevo proyecto de tipo **Windows Forms App**
Net Core llamado **FundamentosNetCore**.



Al abrir el proyecto, tendremos un sistema de ventanas para trabajar.

Algunas ventanas son propias del tipo de proyecto y otras son estandar.

Todas las ventanas están en el menú **View**



La ventana más importante es **Solution Explorer** que es la ventana donde están alojadas nuestras clases y proyectos o soluciones.

En negrita tendremos el proyecto principal a ejecutar.

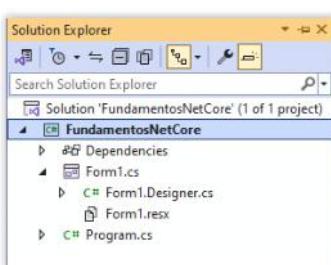
Todo en Net son clases y finalizan con CS

En particular, en este tipo de proyectos Forms tenemos una clase dividida en dos elementos (Para niños pequeños)

En cualquier clase de formularios tenemos una clase nuestra y otra

Para el **Designer**

Cuando nosotros hagamos dibujos en el Form se escribirá dentro del Código del Designer y nuestro código será el otro **FormXXX.cs**



La clase **Program** es la clase que inicia este tipo de aplicaciones.

Indica las características iniciales de nuestra aplicación, por ejemplo,

El objeto de inicio (Form)

```
[STAThread]
0 references
static void Main()
{
    // To customize application configuration see https://aka.ms/applicationconfiguration
    ApplicationConfiguration.Initialize();
    Application.Run(new Form1());
}
```

Los nombres de ficheros pueden ser distintos al nombre de las clases,
Aunque no es recomendable.

Las clases están compuestas por métodos y propiedades

Un método es una acción

Un propiedad son características de un objeto

Todo objeto de una clase debe ser instanciado (**new**)

Coche car = new Coche();
car.ARRANCARVehículo(); --> Método de clase coche
car.Color = Rojo; --> Esto es una propiedad

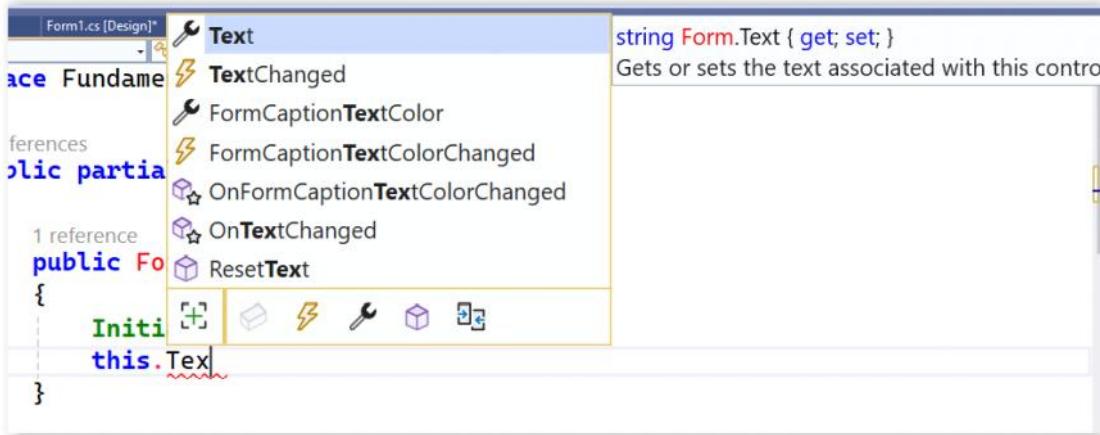
Importante: Los nombres de los métodos o propiedades son todos con la mayúscula en cada palabra.

Por supuesto, el lenguaje C# es **case sensitive**

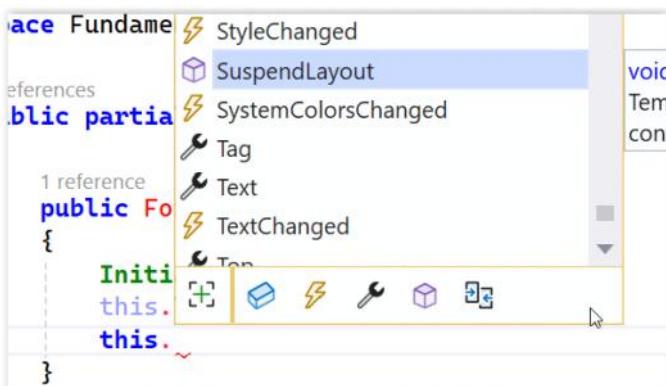
Al principio, nos tendremos que guiar por la ayuda **Intellisense** del Entorno **IDE**

Vamos al código del Form1.

Todas las propiedades tienen un tipado y son representadas con una llave Inglesa.



Los métodos se representan con un cubo Rosa y siempre deben ir con paréntesis.

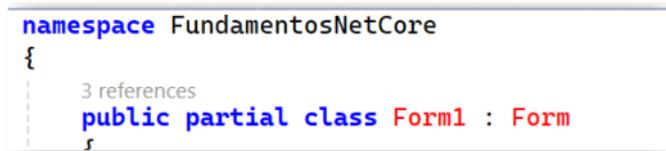


Todas las clases y propiedades tienen un tipado definido.

Todas las clases están representadas por un elemento llamado **namespace**

Dicho espacio de nombres indica una agrupación para las clases y sus tipos. La agrupación nos la inventamos nosotros.

Por defecto, se recupera en namespace del nombre del proyecto. Aunque podemos cambiarlo.



Toda clase contiene también un Constructor como, por ejemplo, **Form1**

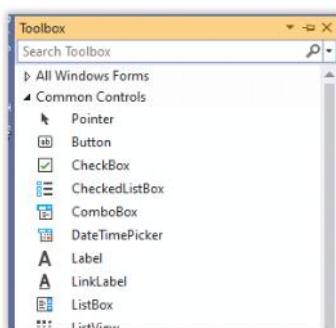
El constructor es el primer método de ejecución de una clase y se debe llamar como la propia clase.

Comenzaremos trabajando con **Controles gráficos**

Cada control tendrá una acción asociada (Evento) un Click de un botón.

Por ahora, solamente podemos escribir dentro de los métodos de Evento.

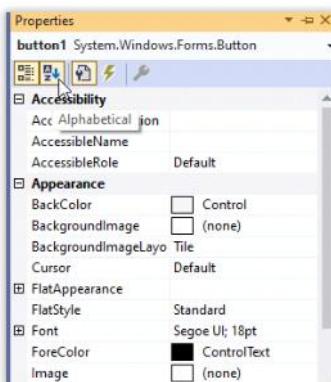
La ventana para poder utilizar controles gráficos se llama **ToolBox**



Lo que nos interesa a continuación es la ventana de **Propiedades**. De los objetos gráficos.

En dicha ventana, podemos cambiar las características de un objeto y, lo más importante, el **name** del objeto, que será el nombre de variable en código.

Para mayor comodidad, pondremos la ventana ordenada alfabéticamente

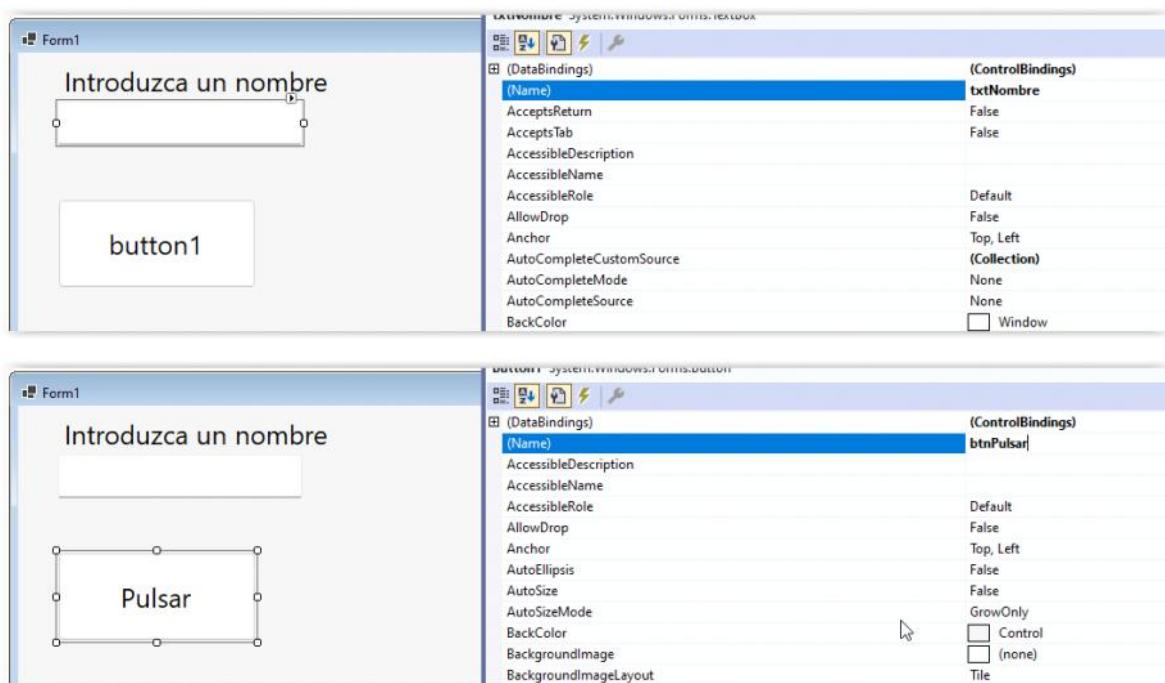


Para la nomenclatura de los controles se suele utilizar la

Nomenclatura húngara:

- TextBox: txtNombre
- Button: btnDescripcion
- Label: lblRespuesta

No quiero ver ni un solo objeto sin **name**. Me refiero a objetos con
Los que yo vaya a interactuar.



Todos los controles tienen eventos predeterminados, para
Recuperar el evento de un control, debemos realizar doble click

Por ejemplo, en nuestro nuevo botón de pulsar.

```
1 reference
private void btnPulsar_Click(object sender, EventArgs e)
{
    //SOLO ESCRIBIMOS AQUI
}
```

CONTROL DE ERRORES

¿Qué sucede si recuperamos un evento y no es que queremos y lo
Borramos?



Si borramos un evento, la clase **Designer** tiene asociado el evento y debemos eliminar la línea de código de la clase Designer

```
60
61     label1.Text = "Introduzca un nombre";
62         label1.Click += label1_Click;
//
```

Si tenemos errores en el código, se llaman **Errores de compilación**

Los errores de compilación son los más fáciles de solucionar y de encontrar, pero tenemos que saber dónde. Esto es un **Assembly**, es decir, todos los elementos tienen que estar correctos para poder ejecutar el código. Si tenemos un error en cualquier clase, no compilará.

Para ejecutar, se utiliza la tecla **F5** o el botón de Play

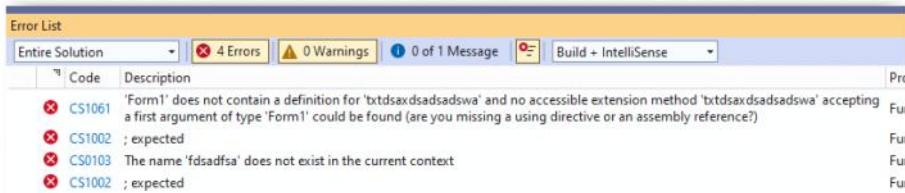


Podemos tener errores en el código, por lo que nos saldrá un mensaje indicando el error

Nota: Siempre diremos que NO y nunca marcaremos el Checkbox



En la parte inferior, tendremos la línea/s de los errores. Simplemente, vamos pulsando en cada línea y los solucionamos



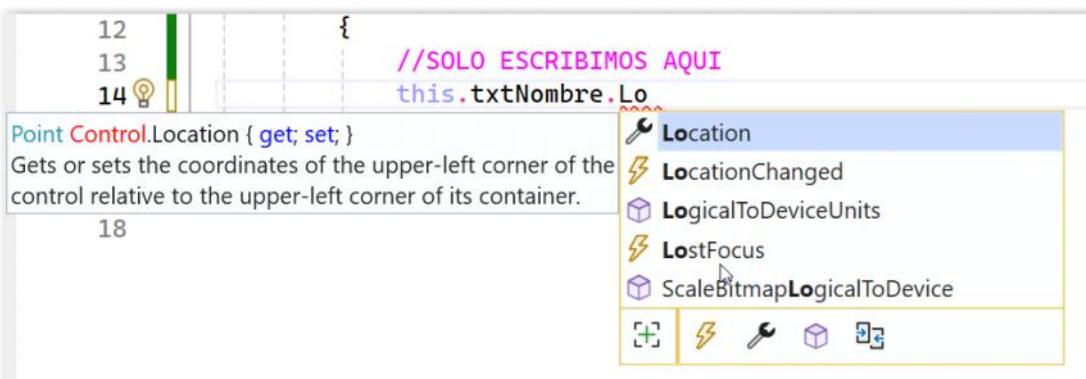
Pulsar SI es absurdo, porque no compila los nuevos cambios.

PROPIEDADES DE TIPO DE OBJETO

Como os he comentado, existen tipados para todos los elementos dentro del entorno de Net Core. (string, int, bool)

En el momento de trabajar con propiedades de objeto, no importa el tipado de dichas propiedades, solamente existen cuatro formas:

- 1) No conocemos el tipado de la propiedad: **new Object()**



```
this.txtNombre.Location = new Point(20, 10);
```

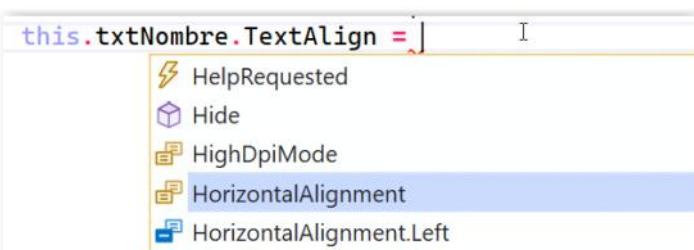
- 2) Propiedades de tipo primitivo: string, int, datetime, bool

Este tipo de propiedades se instancian mediante la igualdad al valor
Que necesitamos. Los tipos primitivos no utilizan new

```
this.txtNombre.Text = "Soy un string";
this.txtNombre.Width = 220;|
```

- 3) Propiedades enumeradas. Una enumeración son una serie de Possibilidades que se nos ofrecen para un tipo de dato.

Nota: Icóno amarillo



```
this.txtNombre.TextAlign = HorizontalAlignment.Right;|
```

- 4) Es una propiedad que sea una estructura (struct).

Una estructura es una clase que contiene una serie de enumeraciones

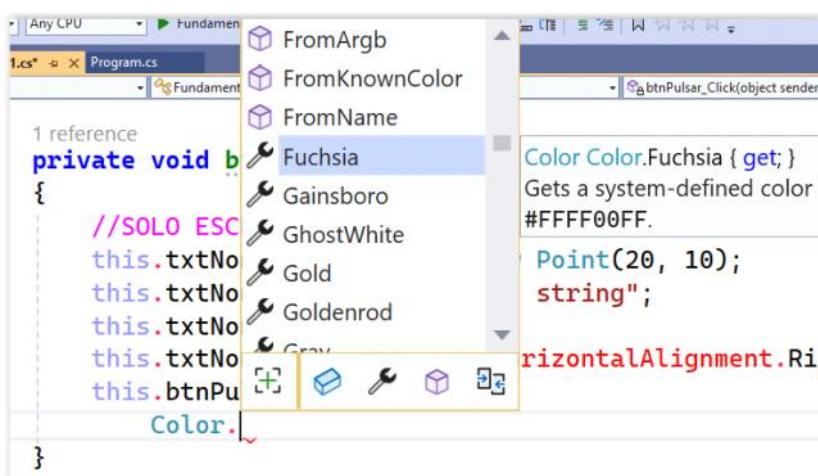
Pero también contiene métodos.

Se utiliza cuando existen valores cerrados y también posibilidad de

Generar algunos valores de forma "manual".

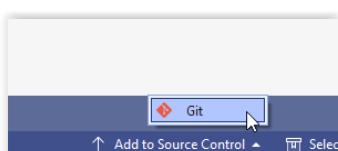
Ejemplo: colores.

Dichas clases no se instancian, se utilizan.

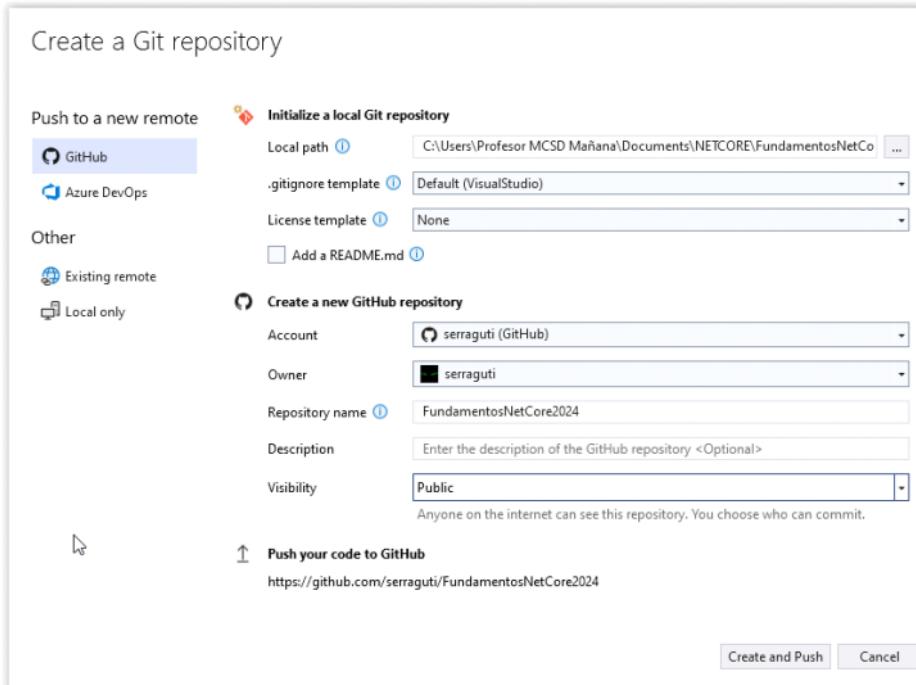


SINCRONIZAR NUESTROS PROYECTOS CON GITHUB

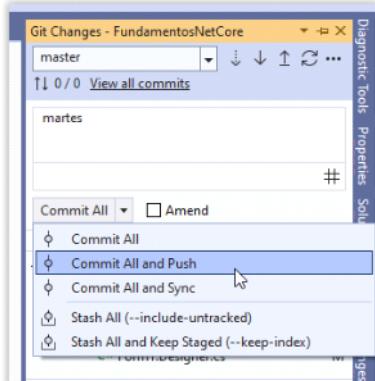
Para subir nuestros proyectos simplemente debemos indicar, en la Parte inferior derecha Add Source Control



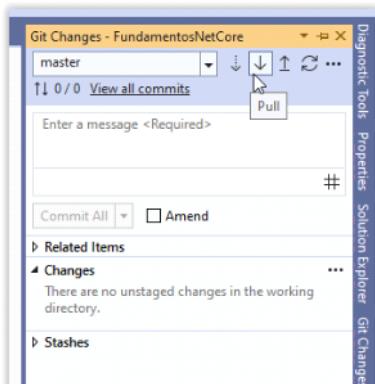
Create a Git repository



Posteriormente, cuando realicemos cambios se hace mediante la Ventana de **Git Changes**



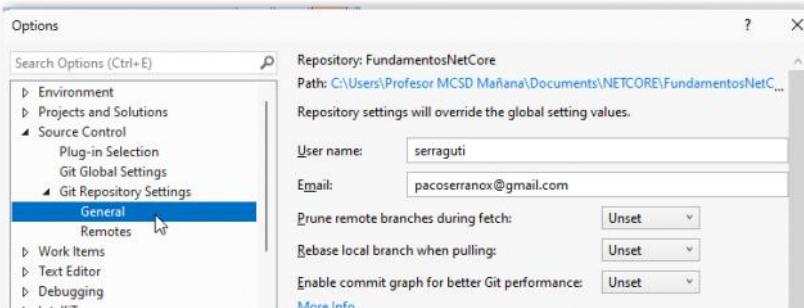
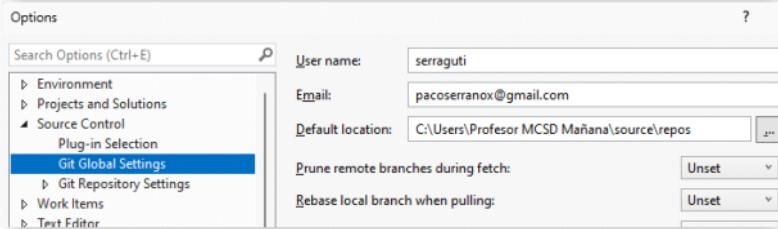
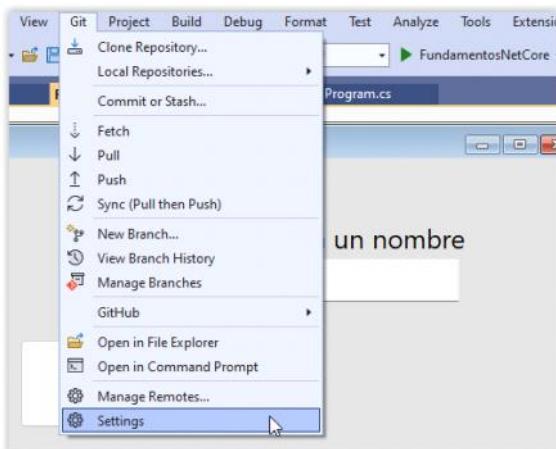
Por último, si tenemos un proyecto que deseamos descargar de la Nube de Github y actualizarlo, utilizamos un Pull.



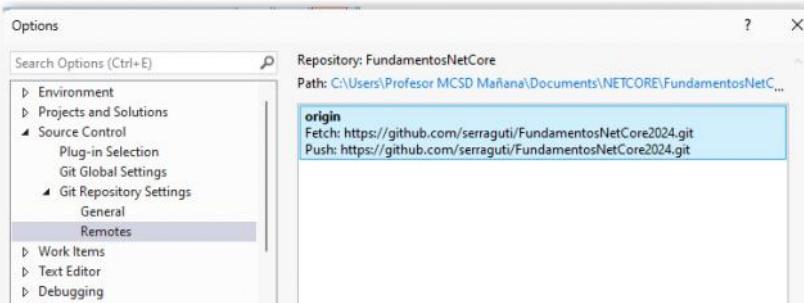
Tenemos que configurar nuestro usuario de Git para que pueda Recuperar todos los Commit o realizar los Commit.

Para ello, cada vez que cambiamos de equipo, debemos quitar la cuenta.

¿Dónde miramos la cuenta o ponemos nuestra cuenta actual?



Si necesitamos cambiar el remote o poner el nuestro, quitamos
El Origin actual



TIPOS PRIMITIVOS O WRAPPER

Un tipo Wrapper es un primitivo que NO utiliza constructor, es decir,
Su constructor está envuelto con la igualdad. Por ejemplo, string, int.

Con igualar el valor, el objeto está instanciado.

```
string texto = "Soy un string";
int numero = 14;

string texto = new string("esto no existe");



- char: Es un carácter unicode. Se representa con la comilla simple.
- byte: Número entero de 0-255
- short: Número entero de 0 - 27000
- int: Número entero 0 - 2,000,0000
- long: Número entero con precisión alta
- float, double, decimal: De menor a mayor números decimales
- bool: True/False
- DateTime: Representa fechas y horas.
- string: Representa conjuntos de textos
- object: Cualquier objeto de Net Core. De hecho, object es la clase
        Base de todo Net Core.

```

CONVERSIÓN DE DATOS

En POO necesitamos declarar los objetos con un tipado.
Si nuestros objetos no cumplen el tipado, tendremos que convertirlos

A su tipo definido.

Tenemos varios tipos de conversiones con los Wrapper.

- 1) **Conversión automática:** Se realiza esta conversión cuando el tipo De dato a almacenar es menor

```
//NO ES CONVERSION AUTOMATICA
int mayor = 88;
short pequeño = mayor;
```

```
//ES CONVERSION AUTOMATICA
short pequeño = 88;
int mayor = pequeño;
```

- 2) Casting de string a primitivo

Para poder convertir cualquier string a primitivo, siempre que sean Compatibles debemos utilizar el método **Parse**.
El método **Parse** está en todos los Wrapper. (Excepto string)

No hace milagros.

```
//STRING A PRIMITIVO
string textoNumero = "Hola mundo";
int numero = int.Parse(textoNumero);
```

```
//STRING A PRIMITIVO
string textoNumero = "Hola mundo";
int numero = int.Parse(textoNumero); X
double otro = double.Parse(textoNumero);
Exception User-Unhandled
System.FormatException: The input string 'Hola mundo' was not in a correct format.'
```

CODIGO CORRECTO

```
//STRING A PRIMITIVO
string textoNumero = "12345";
int numero = int.Parse(textoNumero);
double otro = double.Parse(textoNumero);
```

- 3) Convertir entre tipos primitivos compatibles

Se realiza la conversión de tipos compatibles utilizando **Casting**

Sintaxis: (TIPO DATO A CONVERTIR)objeto;

```
//CASTING PRIMITIVOS
int mayor = 88;
short pequeño = (short)mayor;
```

- 4) Convertir cualquier clase a string

Como os he comentado, todas las clases heredan de **Object**
La clase **Object** tiene un método llamado **ToString()** que permite
Convertir a String cualquier objeto.
Simplemente aplicamos el método **ToString()** a cualquier objeto
Que necesitemos.

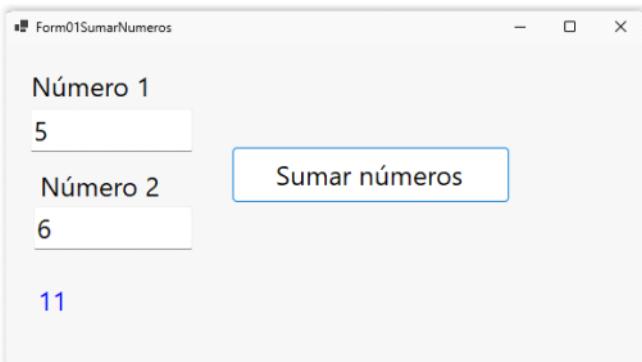
```
//CUALQUIER OBJETO A STRING
int numero = 88;
string texto = numero.ToString();
texto = btnPulsar.ToString();
```

Vamos a crear un nuevo formulario llamado **Form01SumarNumeros**

Cada vez que creamos un nuevo Form debemos cambiar en **Program** el objeto de inicio.

PROGRAM

```
0 references | serragut, 54 minutes ago | 1 author, 1 change
static void Main()
{
    // To customize application configuration such as
    // see https://aka.ms/applicationconfiguration
    ApplicationConfiguration.Initialize();
    Application.Run(new Form01SumarNumeros());
}
```



CODIGO FORMULARIO

```
private void btnSumarNumeros_Click(object sender, EventArgs e)
{
    int num1 = int.Parse(this.txtNumero1.Text);
    int num2 = int.Parse(this.txtNumero2.Text);
    int suma = num1 + num2;
    this.lblResultado.Text = suma.ToString();
}
```

PRACTICA

Creamos un formulario nuevo llamado **Form02PosicionColores**

Debemos realizar lo siguiente:

- Tendremos dos cajas de texto para poder modificar la posición de un botón (Por ejemplo)
- Al pulsar el botón, cambiaremos su posición con el valor de las cajas escritas.
- Tendremos 3 cajas de texto (Rojo, Verde, Azul) para poder cambiar el color de fondo del Formulario.
- Al pulsar un botón, cambiaremos el color de fondo del formulario con los valores escritos en las cajas.



CONDICIONALES

La sintaxis de los condicionales es exactamente igual a como hemos visto en Front

Operadores de comparación

> Mayor
>= Mayor o igual
< Menor
<= Menor o igual
== igual
!= Distinto

Operadores relacionales

&& and
|| or
! NOT

```
if (condicionA) {
    //ACCIONES A
} else if (condicionB) {
    //ACCIONES B
} else {
    //CONDICIONES ELSE
}
```

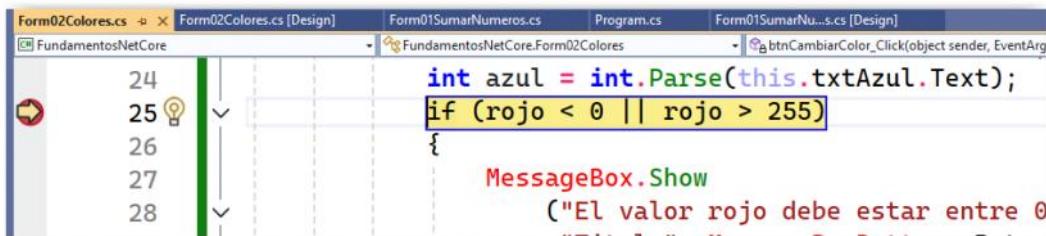
}

CONTROL DE CODIGO Y DEPURACION

Para depurar código necesitamos ir paso a paso por nuestra App y saber dónde se genera el error.
Ahora mismo, tenemos un triste código y está todo en el mismo sitio.
Llegará un momento donde tendremos los códigos repartidos no solo
En diferentes clases, sino en diferentes llamadas entre múltiples
Proyectos.

Para poder depurar debemos incluir puntos de interrupción dentro
De nuestro código.
Se realiza pulsando sobre la zona gris de la izquierda de los
números o con la tecla F9

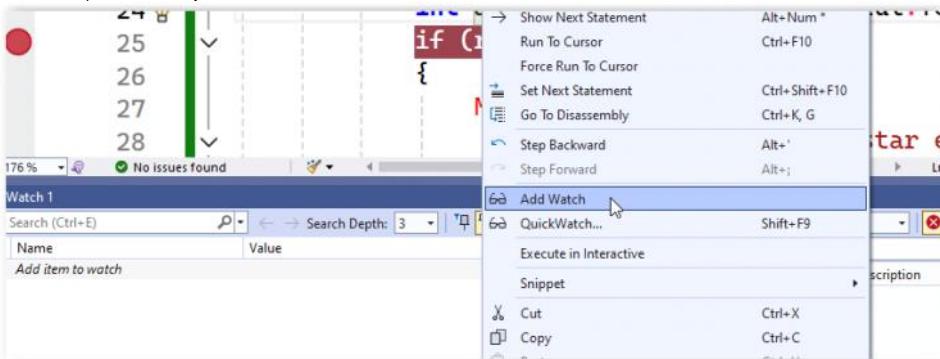
El programa se detendrá en la línea seleccionada



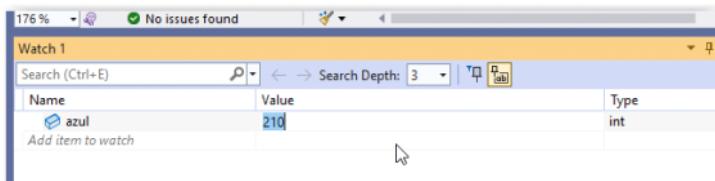
Con la tecla F11 podemos ir yendo paso a paso por nuestro código
Si situamos el cursor encima de las variables, podremos visualizar
El valor de dichas variables en nuestro programa



Dentro de la ventana **Debug** tendremos las variables de nuestro programa
Mediante las pestañas **Locals** y **Autos**



También podemos modificar las variables en tiempo de Debug



Una vez que ya he comprobado los cambios, podremos finalizar
La depuración pulsando sobre F5 o Continuar (Play)

¿Qué sucede si tuvieramos variables que no aparecen ni en Autos ni Locals?

Para poder visualizar variables o elementos que no formen parte de
Mi programa, se utiliza la ventana **Immediate**

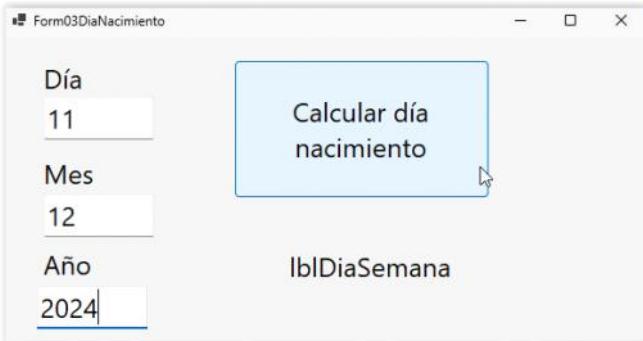
En la ventana Immediate podemos ejecutar instrucciones o preguntar
Por instrucciones, es decir, podemos tirar código.

Para preguntar se utiliza la interrogación seguido de lo que deseemos

?Pregunta

```
Immediate Window
?this.btnAddColor.BackColor
"Name = ffc1717d2, ARGB = (255, 23, 23, 210)"
A: 255
B: 210
G: 23
IsEmpty: false
IsKnownColor: false
IsNamedColor: false
IsSystemColor: false
```

Vamos a crear un nuevo formulario llamado **Form03DiaNacimiento**



ESTE EJEMPLO ES PARA PROBAR LA FUNCIONALIDAD DE LOS PUNTOS
DE INTERRUPCIÓN

- Pedir una fecha al usuario para calcular el día de la semana de su nacimiento.
- Realizaremos el diseño con tres cajas para que el usuario pueda incluir día, mes y año.
- Mediante un botón, mostraremos el día de la semana en un Label, por ejemplo.

Tenemos que tener la tabla de días de la semana para la correspondencia comenzando en sábado:

DÍA	NÚMERO
Sábado	0
Domingo	1
Lunes	2
Martes	3
Miércoles	4
Jueves	5
Viernes	6

Debemos pedir el **día**, el número de **mes** y el **año** que el usuario haya nacido.
A partir de estos datos hay que calcular lo siguiente para averiguar el día de la semana de nacimiento:
Ejemplo → **15/06/1997**

Hay que tener en cuenta el mes para realizar el cálculo, si el mes es Enero, el Mes será 13 y restaremos uno al año. Si el Mes es Febrero, el Mes será 14 y restaremos uno al año.

Para poder calcular el número final de la semana debemos seguir los siguientes pasos:

1. Multiplicar el Mes más 1 por 3 y dividirlo entre 5
 $((6 + 1) * 3) / 5 \rightarrow 4$
2. Dividir el año entre 4
 $1997 / 4 \rightarrow 499$
3. Dividir el año entre 100
 $1997 / 100 \rightarrow 19$
4. Dividir el año entre 400
 $1997 / 400 \rightarrow 4$
5. Sumar el dia, el doble del mes, el año, el resultado de la operación 1, el resultado de la operación 2, menos el resultado de la operación 3 más la operación 4 más 2.
 $15 + (6 * 2) + 1997 + 4 + 499 - 19 + 4 + 2 \rightarrow 2514$
6. Dividir el resultado anterior entre 7.
 $2514 / 7 \rightarrow 359$
7. Restar el número del paso 5 con el número del paso 6 por 7.
 $2514 - (359 * 7) \rightarrow 1$
8. Miramos la tabla y vemos que el número 1 corresponde a **DOMINGO**

CÓDIGO FORMULARIO

```
private void btnCalcularDiaNacimiento_Click(object sender, EventArgs e)
{
    int dia = int.Parse(this.txtDia.Text);
    int mes = int.Parse(this.txtMes.Text);
    int anyo = int.Parse(this.txtAnyo.Text);
    if(mes == 1)
    {
        mes = 13;
        anyo -= 1;
    }else if (mes == 2)
    {
```

```

        mes = 14;
        anyo--;
    }
    int op1 = ((mes + 1) * 3) / 5;
    int op2 = anyo / 4;
    int op3 = anyo / 100;
    int op4 = anyo / 400;
    int op5 = dia + (mes * 2) + anyo + op1 + op2 - op3 + op4 + 2;
    int op6 = op5 / 7;
    int resultado = op5 - (op6 * 7);
    if (resultado == 0)
    {
        this.lblDiaSemana.Text = "SABADO";
    }else if (resultado == 1)
    {
        this.lblDiaSemana.Text = "DOMINGO";
    }else if (resultado == 2)
    {
        this.lblDiaSemana.Text = "LUNES";
    }else if (resultado == 3)
    {
        this.lblDiaSemana.Text = "MARTES";
    }else if (resultado == 4)
    {
        this.lblDiaSemana.Text = "MIERCOLES";
    }else if (resultado == 5)
    {
        this.lblDiaSemana.Text = "JUEVES";
    }else if (resultado == 6)
    {
        this.lblDiaSemana.Text = "VIERNES";
    }else
    {
        this.lblDiaSemana.Text = "Algo ha ido muy mal!";
    }
}
}

```

CLASE DATETIME

Todos son clases dentro de Visual Studio Core. En este caso, estamos hablando de una clase **Wrapper**, es decir, una clase que no utiliza un constructor.

```
DateTime fecha = "11/12/2024";
```

Métodos y propiedades de la clase DateTime

- AddDays(numero)
- AddMonths(numero)
- ToShortDateString()
- ToLongDateString()
- ToShortTimeString()
- ToLongTimeString()
- Year
- Month
- Minutes
- Seconds
- DayOfWeek
- DayOfYear

Estos son métodos de objeto.

Dentro de las clases tenemos también métodos **static**

Un método static es una herramienta de la clase y que NO utiliza el objeto para su funcionalidad.

Ejemplo: Tenemos un método llamado **IsLeapYear(AÑO)** que indica si un año es bisiesto o no.

```
DateTime fecha = "11/12/2024";
```

Pongamos que escribo lo siguiente:

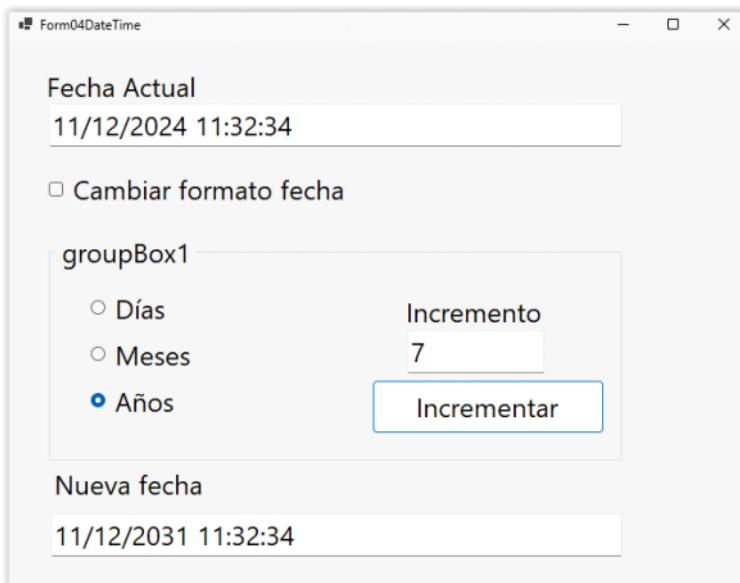
```
fecha.IsLeapYear(2030) --> A quién está haciendo caso? (2024, 2030)
```

No utiliza el objeto, hace referencia al año 2030, de hecho, se llama correctamente desde la propia clase:

```
DateTime.IsLeapYear(2030)
```

También tenemos una propiedad llamada **Now** que nos devuelve la fecha actual.

Creamos un nuevo formulario llamado **Form04DateTime**



CODIGO FORMULARIO

```

public partial class Form04DateTime : Form
{
    public Form04DateTime()
    {
        InitializeComponent();
        //AL INICIAR LA CLASE (CONSTRUCTOR)
        //ESCRIBIREMOS LA FECHA ACTUAL
        this.txtFechaActual.Text =
            DateTime.Now.ToString();
    }

    private void chkCambiarFormato_CheckedChanged(object sender, EventArgs e)
    {
        //RECUPERAMOS LA FECHA DE LA CAJA
        DateTime fecha = DateTime.Parse(this.txtFechaActual.Text);
        if (this.chkCambiarFormato.Checked == true)
        {
            this.txtFechaActual.Text = fecha.ToString("yyyy/MM/dd");
        }
        else
        {
            this.txtFechaActual.Text = fecha.ToString("dd/MM/yyyy");
        }
    }

    private void btnIncrementar_Click(object sender, EventArgs e)
    {
        int incremento = int.Parse(this.txtIncremento.Text);
        DateTime fecha = DateTime.Parse(this.txtFechaActual.Text);
        if (this.rdbDias.Checked == true)
        {
            fecha = fecha.AddDays(incremento);
        }
        else if (this.rdbMeses.Checked == true)
        {
            fecha = fecha.AddMonths(incremento);
        }
        else
        {
            fecha = fecha.AddYears(incremento);
        }
        this.txtNuevaFecha.Text = fecha.ToString();
    }
}

```

CLASE CHAR

La Clase Char almacena un solo carácter. Se utilizan las comillas simples
Para almacenar el valor

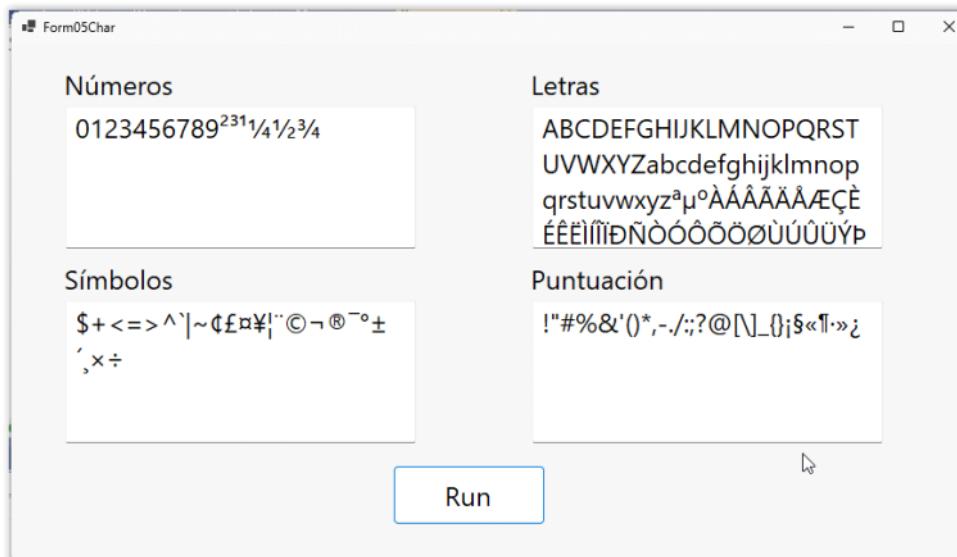
```
char letra = '@';
```

La gran mayoría de sus métodos son **static**
Esta clase es muy útil para las validaciones.

- char.IsLetter(letra)
- char.IsDigit(letra)
- IsNumber(letra)
- IsLetterOrDigit(letra)
- IsPunctuation(letra)
- IsSymbol(letra)
- IsWhiteSpace(letra)
- IsUpper(letra)
- IsLower(letra)
- ToLower(letra)
- ToUpper(letra)

Vamos a visualizar todos los caracteres del código Ascii 0-255
Y los iremos separando en cajas.

Creamos un nuevo formulario llamado **Form05Char**



CODIGO FORMULARIO

```
private void btnRun_Click(object sender, EventArgs e)
{
    for (int i = 0; i <= 255; i++)
    {
        //CONVERTIMOS EL VALOR DE i A CHAR
        char caracter = (char) i;
        if (char.IsLetter(caracter) == true)
        {
            this.txtLetras.Text += caracter;
        } else if (char.IsNumber(caracter) == true)
        {
            this.txtNumeros.Text += caracter;
        } else if (char.IsPunctuation(caracter) == true)
        {
            this.txtPuntuacion.Text += caracter;
        } else if (char.IsSymbol(caracter) == true)
        {
            this.txtSimbolos.Text += caracter;
        }
    }
}
```

CLASE STRING

La clase string es un conjunto de caracteres.
Todos los conjuntos dentro de Net Core comienzan en 0

string cadena = "Visual Studio";

0--> V
1--> i
2--> s

Dentro de los conjuntos tenemos propiedades **indizadas**
Un propiedad indizada accede a algún elemento del objeto por su índice.

cadena[2] --> s
cadena[3] --> u

Métodos y propiedades de la clase string

- Length: Indica la longitud del texto: 13
- Char[índice]: Devuelve el carácter char de la posición del índice
- StartsWith(contenido): Devuelve un bool indicando si el texto Comienza con el contenido
- EndsWith(contenido): Devuelve un bool indicando si el texto Finaliza con el contenido
- IndexOf(contenido): Devuelve la posición del contenido dentro de La cadena. Si no lo encuentra, devuelve -1.
Devuelve la primera coincidencia.

```
string cadena = "hola coca cola";
int posicion = cadena.IndexOf("o"); //posición 1
```

- IndexOf(contenido, índice): Busca el contenido y devuelve su posición
A partir del índice indicado. Si no lo encuentra devuelve -1.

```
string cadena = "hola coca cola";
int posicion = cadena.IndexOf("o", 2); //posición 6
```

- LastIndexOf(contenido): Busca el contenido y devuelve su posición
Pero comenzando al final de la cadena. Si no lo encuentra devuelve -1

```
string cadena = "hola coca cola";
int posicion = cadena.LastIndexOf("o"); //posición 11
```

- **LastIndexOf(contenido):** Busca el contenido y devuelve su posición
A partir de un índice superior
Pero comenzando al final de la cadena. Si no lo encuentra devuelve -1

```
string cadena = "hola coca cola";
int posicion = cadena.LastIndexOf("o", 10); //posición 6
```

- **Contains(contenido):** Busca un contenido en una cadena y
Nos devuelve un bool.

- **SubString(inicio):** Devuelve una subcadena desde la posición
De inicio inclusive

```
string cadena = "hola coca cola";
string subcadena = cadena.Substring(6); //oca cola
```

- **SubString(inicio, número de caracteres):** Devuelve una subcadena
Comenzando en inicio inclusive y con el número de caracteres indicado.

```
string cadena = "hola coca cola";
string subcadena = cadena.Substring(6, 3); //oca
```

- **ToUpper():** Convierte el texto a mayúsculas
- **ToLower():** Convierte el texto a Minúsculas
- **Trim():** Elimina espacios al inicio y al final de la cadena
- **Trim(carácter):** Elimina el carácter indicado al inicio y al final de la cadena
- **TrimStart():** Elimina espacios al inicio de la cadena
- **TrimEnd():** Elimina espacios al final de la cadena

```
string cadena = "@@@hola coc@@@ cola@@@@";
//hola coc@@@ cola
string subcadena = cadena.Trim('@');
```

- **Remove(inicio):** Elimina los caracteres a partir del inicio
- **Remove(inicio, número de caracteres):**
Elimina los caracteres a partir del inicio indicando el número de caracteres
A eliminar
- **PadLeft(Total):** Inserta espacios al inicio de la cadena hasta llegar
Al número total
- **PadRight(Total):**
Inserta espacios al final de la cadena hasta llegar
Al número total

```
string cadena = "hola coca cola";
//hola coca cola&&&&&
string subcadena = cadena.PadRight(30, '&');
```

- **Replace(Antiguo texto, nuevo texto):** Cambia el antiguo texto por el
Nuevo texto en todas las coincidencias.

PRACTICA VALIDAR EMAIL

No quiero bucles, solamente métodos de la clase string
Y con if else if.

Validar un Mail:

- Existe @
- Ni @ al inicio ni al final
- No existe más de una @
- Existe un punto
- Existe un punto después de @
- Dominio de 2 a 4 caracteres (com, es, org, como)

Creamos un nuevo formulario llamado **Form06ValidarMail**

CODIGO FORMULARIO

```
private void btnValidarMail_Click(object sender, EventArgs e)
{
    string email = this.txtEmail.Text;
    if (email.Contains("@") == false)
    {
        this.lblResultado.Text = "No existe @";
    }
    else if (email.IndexOf("@") == 0 || email.EndsWith("@") == true)
    {
        this.lblResultado.Text = "@ al inicio o al final";
    }
    else if (email.IndexOf("@") != email.LastIndexOf("@"))
    {
        this.lblResultado.Text = "Más de una @ en el mail";
    }
    else if (email.IndexOf(".") == -1)
    {
        this.lblResultado.Text = "no existe punto en el mail";
    }
    else if (email.LastIndexOf(".") < email.IndexOf("@"))
    {
        this.lblResultado.Text = "Necesitamos un punto después de @";
    }
    else
    {
        int ultimoPunto = email.LastIndexOf(".");
        string dominio = email.Substring(ultimoPunto + 1);
        if (dominio.Length >= 2 && dominio.Length <= 4)
        {
            this.lblResultado.Text = "Email correcto";
        }
        else
        {
            this.lblResultado.Text = "Dominio de 2 a 4 caracteres";
        }
    }
}
```

Vamos a realizar un nuevo formulario en el que recorreremos un texto numérico y sumaremos todos sus caracteres.

1234 = La suma es 10

Creamos un nuevo formulario llamado **Form07SumarNumerosString**

CODIGO FORMULARIO

```
private void btnSumarNumeros_Click(object sender, EventArgs e)
{
    string textoNumeros = this.txtNumeros.Text;
    int suma = 0;
    for (int i = 0; i < textoNumeros.Length; i++)
    {
        //RECUPERAMOS CADA UNO DE LOS CARACTERES
        char caracter = textoNumeros[i];
        //CONVERTIMOS EL CARÁCTER A NÚMERO
        //CON ESTA CONVERSIÓN ESTA RECUPERANDO EL VALOR
        //ASCII DEL NÚMERO
        //int numero = caracter;
        //int numero = int.Parse(caracter.ToString());
        int numero = Convert.ToInt32(caracter.ToString());
        suma += numero;
    }
    this.lblResultado.Text = "La suma es " + suma;
}
```

Vamos a visualizar clases que no tenemos directamente agregadas dentro de nuestro Formulario (Clase)
Cada conjunto de clases está alojada dentro de un **namespace**

Si necesitamos utilizar clases especializadas en nuestro código, debemos incluir
Nuestro namespace para poder acceder a ellas.

Por ejemplo, existen clases para trabajar con ficheros.
Dichas clases no están accesibles por defecto dentro de nuestro código.
Si yo necesito acceder a clases de acceso a ficheros, debo utilizar el
Namespace de System.IO

```
using System;
using System.IO;
```

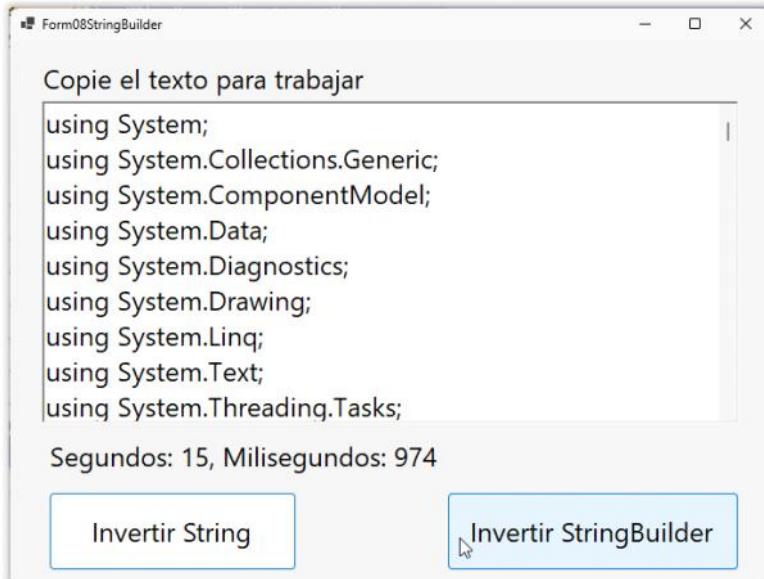
Vamos a realizar una aplicación para medir el tiempo que tarda un proceso.
Dicho proceso comparará la clase **String** en el momento de realizar un proceso
Más o menos complicado.

Por ejemplo, la clase **string** está bien mientras que no superemos los 65.000
Caracteres, a partir de ahí, le cuesta un mundo.

Para medir el proceso debemos utilizar una clase llamada **Stopwatch** y que está
Dentro del namespace **System.Diagnostics**

Una vez que hayamos medido el tiempo del proceso, lo que haremos será
Mejorar la aplicación y utilizar una clase llamada **StringBuilder**

Creamos un nuevo formulario llamado **Form08StringBuilder**



CODIGO FORMULARIO

```
using System;
using System.Text;
using System.Diagnostics;

private void btnInvertirString_Click(object sender, EventArgs e)
{
    Stopwatch krono = new Stopwatch();
    string cadena = this.txtTexto.Text;
    int longitud = cadena.Length;
    krono.Start();
    for (int i = 0; i < cadena.Length; i++)
    {
        //ALOH
        //RECUPERAMOS LA ULTIMA LETRA
        char letra = cadena[longitud - 1];
        //ELIMINAMOS LA ULTIMA LETRA
        cadena = cadena.Remove(longitud - 1, 1);
        //INSERTAMOS LA LETRA EN LA POSICION DEL BUCLE
        cadena = cadena.Insert(i, letra.ToString());
    }
    krono.Stop();
    //EL OBJETO krono CONTIENE UNA SERIE DE PROPIEDADES
    //PARA SABER EL TIEMPO QUE HA PASADO
    this.lblTiempo.Text = "Segundos: " +
        krono.Elapsed.Seconds
        + ", Milisegundos: "
        + krono.Elapsed.Milliseconds;
    this.txtTexto.Text = cadena;
}

private void btnInvertirStringBuilder_Click(object sender, EventArgs e)
{
    //StringBuilder se utiliza para grandes cantidades de texto.
    Stopwatch krono = new Stopwatch();
    StringBuilder cadena = new StringBuilder();
    //PARA AÑADIR CONTENIDO AL STRINGBUILDER
    cadena.Append(this.txtTexto.Text);
    int longitud = cadena.Length;
    krono.Start();
    for (int i = 0; i < cadena.Length; i++)
    {
        char letra = cadena[longitud - 1];
        cadena = cadena.Remove(longitud - 1, 1);
        cadena = cadena.Insert(i, letra);
    }
    krono.Stop();
    this.lblTiempo.Text = "Segundos: " +
        + krono.Elapsed.Seconds
        + ", Milisegundos: "
        + krono.Elapsed.Milliseconds;
    this.txtTexto.Text = cadena.ToString();
}
```

VALIDAR ISBN

- Realizar una aplicación que nos permita validar si el ISBN de un libro es correcto o incorrecto.
- Comprobar que el usuario introduce 10 caracteres.
- Realizamos un nuevo formulario llamado **Form09ISBN**

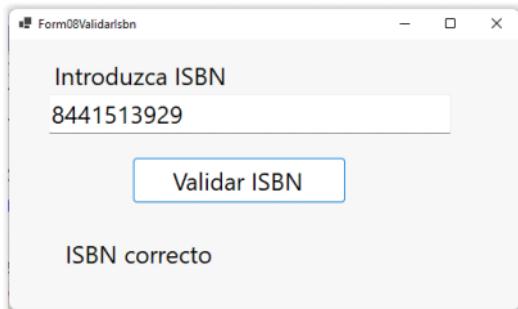
EJEMPLO DE NUMERO ISBN QUE ESTÁ BIEN:

8441513929

- 1) Se descompone la cadena y se multiplica cada número por la posición que ocupa en la cadena:

```
8 * 1
4 * 2
4 * 3
1 * 4
5 * 5
.
.
9 * 10
```

- 1) La suma de todas estas multiplicaciones se divide entre 11, y si el resto es cero, el número ISBN es correcto.



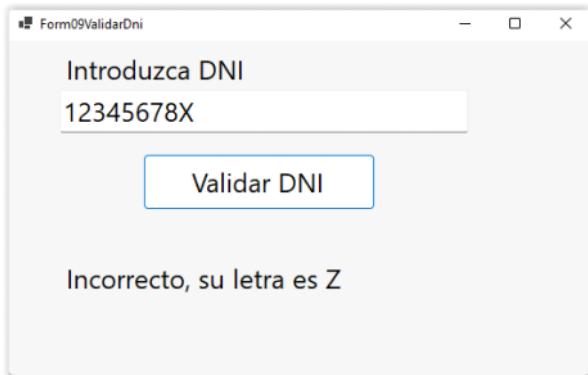
CODIGO FORMULARIO

```
private void btnValidarIsbn_Click(object sender, EventArgs e)
{
    string isbn = this.txtIsbn.Text;
    if (isbn.Length != 10)
    {
        this.lblResultado.Text = "El ISBN debe ser de 10 caracteres";
    }
    else
    {
        int suma = 0;
        for (int i = 0; i < isbn.Length; i++)
        {
            char caracter = isbn[i];
            int numero = int.Parse(caracter.ToString());
            int operacion = numero * (i + 1);
            suma += operacion;
        }
        if (suma % 11 == 0)
        {
            this.lblResultado.Text = "ISBN correcto";
            this.lblResultado.ForeColor = Color.Blue;
        }
        else
        {
            this.lblResultado.Text = "ISBN incorrecto";
            this.lblResultado.ForeColor = Color.Red;
        }
    }
}
```

CALCULAR LA LETRA DEL D.N.I

- Realizar una aplicación para conocer la letra del Documento Nacional de Identidad a través del número de DNI.
- La fórmula para calcular la letra del número del DNI se halla de la siguiente manera:
- Se calcula el valor de la siguiente resta
 $(\text{nº DNI} - (\text{PARTE ENTERA}(\text{nº DNI} / 23) * 23))$
- Se mira la equivalencia en la siguiente tabla

0=T	4=G	8=P	12=N	16=Q	20=C
1=R	5=M	9=D	13=J	17=V	21=K
2=W	6=Y	10=X	14=Z	18=H	22=E
3=A	7=F	11=B	15=S	19=L	23=T



CONJUNTOS DE OBJETOS

Los objetos dentro de Net Core pueden conformar conjuntos de dos formas:
Array o Colección.

Por ejemplo, también tenemos un conjunto que acabamos de probar: String.

- 1) Array: Es un conjunto de objetos definidos por un índice, pero es un Conjunto estático, es decir, no puede cambiar.
Por ejemplo, si creamos un Array con 5 elementos, siempre tendrá 5 elementos, aunque sus posiciones estén vacías.

```
int[] miArray = new int[5];  
int miArray = {6, 7, 8, 8};
```

Propiedades y métodos:

- GetUpperBound()
- GetLength()
- GetLowerBound()
- Length

- 1) **Colecciones:** Es un conjunto de objetos definidos por un índice, pero su Contenido es dinámico, es decir, puede crecer o decrecer.
Siempre nos dará el número exacto de elementos dentro del conjunto.
Optimiza la memoria.

Propiedades y métodos de una colección:

- Count: Recupera el número de elementos de la colección
- Add(objeto): Añade un elemento a la colección
- Clear(): Elimina todos los elementos de la colección
- Remove(objeto): Elimina un elemento de la colección por su objeto
- RemoveAt(indice): elimina un objeto de la colección por su índice
- Contains(objeto): Busca un objeto y devuelve bool
- IndexOf(objeto): Busca el objeto y devuelve su posición.

Nosotros no sabremos que objetos estamos manejando, cuando hablamos de Conjuntos de objetos, nos importará cada elemento individual.
No importa la forma del conjunto: Array o Colección, solamente tenemos Que reconocerlo.

Esto un concepto de POO llamado **Abstracción**

Dentro de los conjuntos, tenemos los bucles de referencia.

Un bucle de referencia es un **ReadOnly**, no se puede tocar el Conjunto.

```
foreach (Clase objeto in Conjunto) {  
    //objeto es la variable de referencia  
}
```

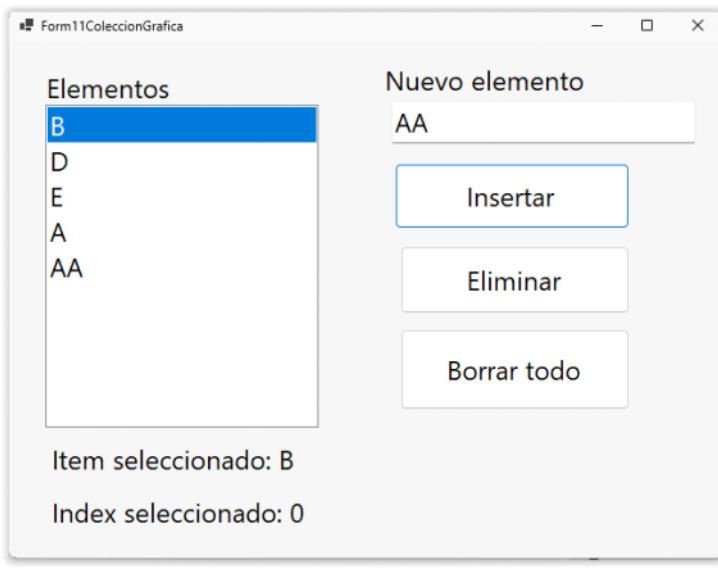
```
string texto = "juernes";  
  
foreach (char letra in texto) {  
    texto += letra;  
}
```

Para los conjuntos vamos a comenzar utilizando un objeto gráfico llamado **ListBox** que permite incluir en su interior (colección) un conjunto.

Las propiedades más interesantes:

- **Items:** Es la colección de elementos dentro del ListBox
- **SelectedItems:** Colección de elementos seleccionados
- **SelectedIndices:** Colección de índices de los elementos seleccionados
- **SelectedIndex:** Indica el índice del elemento seleccionado
- **SelectedItem:** Indica el Item seleccionado
- **SelectionMode:** Cambiar la selección a múltiple o Single

Creamos un nuevo formulario llamado **Form11ColeccionGrafica**



CODIGO FORMULARIO

```
public partial class Form11ColeccionGrafica : Form
{
    public Form11ColeccionGrafica()
    {
        InitializeComponent();
    }

    private void btnInsertar_Click(object sender, EventArgs e)
    {
        string elem = this.txtNuevoElemento.Text;
        this.lstElementos.Items.Add(elem);
    }

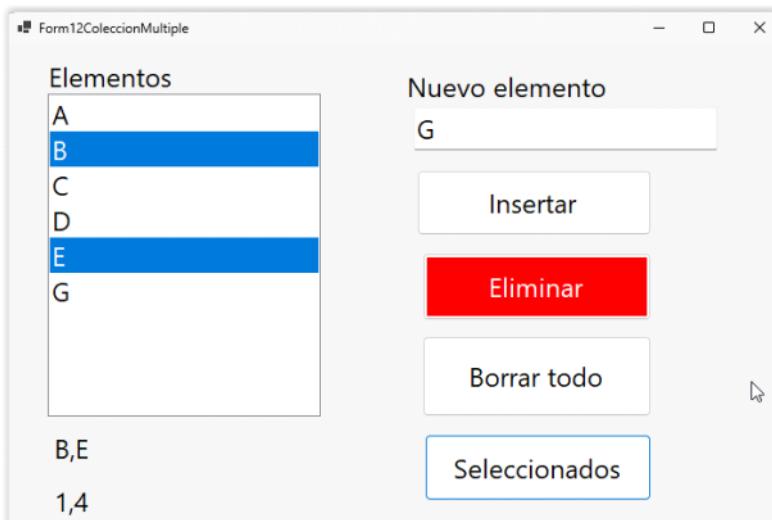
    private void btnEliminar_Click(object sender, EventArgs e)
    {
        //string elementoSeleccionado =
        //this.lstElementos.SelectedItem.ToString();
        //this.lstElementos.Items.Remove(elementoSeleccionado);
        int indexSeleccionado =
            this.lstElementos.SelectedIndex;
        this.lstElementos.Items.RemoveAt(indexSeleccionado);
    }

    private void btnBorrarTodo_Click(object sender, EventArgs e)
    {
        this.lstElementos.Items.Clear();
    }

    private void lstElementos_SelectedIndexChanged(object sender, EventArgs e)
    {
        //DEBERIAMOS PREGUNTAR EN ESTE EVENTO
        //SI TENEMOS ELEMENTOS SELECCIONADOS ANTES DE JUGAR
        if (this.lstElementos.SelectedIndex != -1)
        {
            this.lblIndexSeleccionado.Text =
                "Index seleccionado: " + this.lstElementos.SelectedIndex;
            this.lblItemSeleccionado.Text =
                "Item seleccionado: " + this.lstElementos.SelectedItem;
        }
    }
}
```

El siguiente paso que vamos a realizar es visualizar el comportamiento cuando manejamos Una colección y deseamos realizar acciones múltiples (Selección múltiple)

Creamos un nuevo formulario llamado **Form12ColeccionMultiple**



CODIGO FORMULARIO

```
public partial class Form12ColeccionMultiple : Form
```

```

{
    public Form12ColeccionMultiple()
    {
        InitializeComponent();
        this.lstElementos.SelectionMode =
            SelectionMode.MultiExtended;
    }

    private void btnInsertar_Click(object sender, EventArgs e)
    {
        string elem = this.txtNuevoElemento.Text;
        this.lstElementos.Items.Add(elem);
        this.txtNuevoElemento.Focus();
        this.txtNuevoElemento.SelectAll();
    }

    private void btnSeleccionados_Click(object sender, EventArgs e)
    {
        string indices = "";
        string items = "";
        foreach (int index in this.lstElementos.SelectedIndices)
        {
            indices += index + ",";
        }
        this.lblIndexSeleccionados.Text = indices.Trim(',');
        foreach (string item in this.lstElementos.SelectedItems)
        {
            items += item + ",";
        }
        this.lblItemsSeleccionados.Text = items.Trim(',');
    }

    private void btnBorrarTodo_Click(object sender, EventArgs e)
    {
        this.lstElementos.Items.Clear();
    }

    private void btnEliminar_Click(object sender, EventArgs e)
    {
        //NECESITAMOS ELIMINAR MULTIPLES ELEMENTOS
        //DENTRO DE LA LISTA. ELIMINAREMOS MEDIANTE SU index
        //DEBEMOS RECORRER LA COLECCION SelectedIndices
        //RECUPERAMOS CADA INDICE Y ELIMINAMOS CADA ITEM
        //COMO TENEMOS REPOSICIONAMIENTO DE INDICES AL ELIMINAR
        //DE CUALQUIER COLECCION, DEBEMOS HACER EL RECORRIDO
        //DE FORMA INVERSA
        int numSeleccionados = this.lstElementos.SelectedIndices.Count;
        for (int i = numSeleccionados - 1; i >= 0; i--)
        {
            int index = this.lstElementos.SelectedIndices[i];
            //ELIMINAMOS CADA ELEMENTO DE LA COLECCION
            this.lstElementos.Items.RemoveAt(index);
        }
    }
}

```

Creamos otro formulario llamado **Form13ColecciónNumeros**

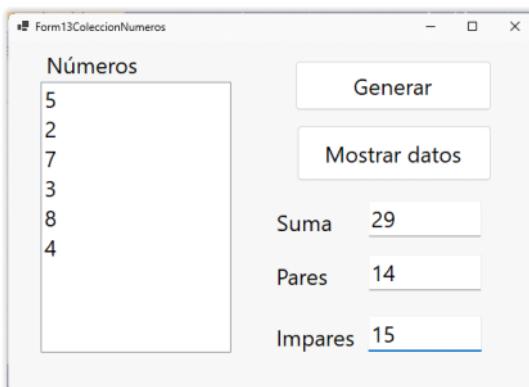
- Generar 10 aleatorios al pulsar sobre el botón **Generar** y los dibujamos Dentro del ListBox
- Al pulsar sobre Mostrar datos, veremos la suma total de todos los números
- La suma de sus Pares y la suma de Impares

Para generar números aleatorios existe una clase llamada **Random**

Debemos crear un objeto: **Random random = new Random();**

El objeto contiene tres métodos para la generación de aleatorios

- **Next(min, max)**: Genera números aleatorios enteros entre min y max
- **NextDouble()**: Genera un número random comprendido entre 0 y 1
- **NextBytes()**: Devuelve un Array de Bytes con números aleatorios en su interior



CODIGO FORMULARIO

```

public partial class Form13ColecciónNumeros : Form
{
    public Form13ColecciónNumeros()
    {
        InitializeComponent();
    }

    private void btnGenerar_Click(object sender, EventArgs e)
    {
        this.lstNumeros.Items.Clear();
        Random random = new Random();
        for (int i = 1; i <= 10; i++)
        {
            int aleat = random.Next(1, 50);
            this.lstNumeros.Items.Add(aleat);
        }
    }

    private void btnMostrarDatos_Click(object sender, EventArgs e)

```

```

{
    int suma = 0;
    int sumaPares = 0;
    int sumaImpares = 0;
    foreach (int num in this.lstNumeros.Items)
    {
        suma += num;
        if (num % 2 == 0)
        {
            sumaPares += num;
        }
        else
        {
            sumaImpares += num;
        }
    }
    this.txtSuma.Text = suma.ToString();
    this.txtPares.Text = sumaPares.ToString();
    this.txtImpares.Text = sumaImpares.ToString();
}
}

```

PRACTICA TIENDA DE PRODUCTOS

Creamos un nuevo formulario llamado **Form14TiendaProductos**

La zona de la izquierda es para **Tienda**.

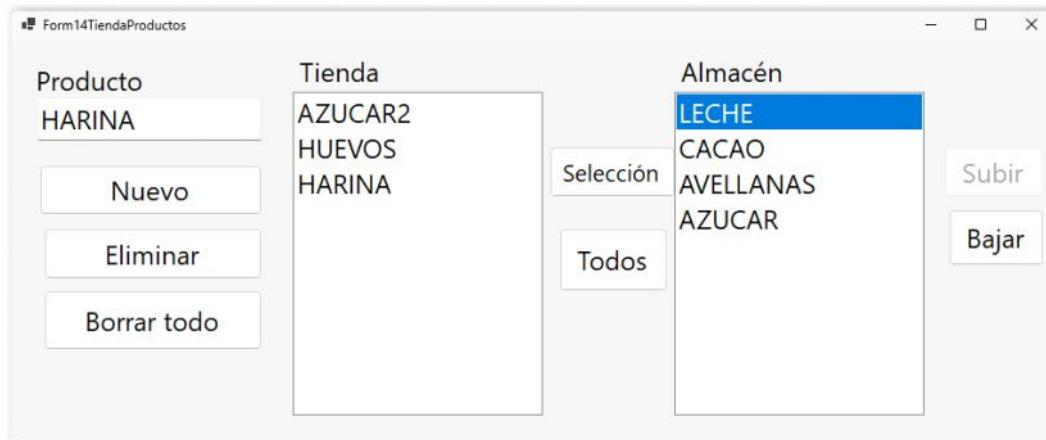
- Iremos añadiendo Productos a la tienda. No podremos tener productos Repetidos en Tienda. Si tenemos alguno repetido, no lo agregamos y lo marcamos
- Podremos eliminar múltiples productos de Tienda o borrar todos

Botones centrales:

- El botón **Selección**, enviará los elementos seleccionados de la tienda hasta el Almacén. Eliminamos los productos seleccionados de la tienda.
- El botón **Todos** enviará todo lo de la tienda al Almacén. Eliminará todos los productos de la tienda.

Botones derecha:

- El botón **subir**, subirá el elemento seleccionado una posición
- El botón **bajar**, bajará el elemento seleccionado una posición.
- Debemos comprobar las posiciones



CODIGO FORMULARIO

```

public partial class Form14TiendaProductos : Form
{
    public Form14TiendaProductos()
    {
        InitializeComponent();
        this.lstTienda.SelectionMode =
            SelectionMode.MultiExtended;
    }

    private void btnNuevo_Click(object sender, EventArgs e)
    {
        string producto = this.txtProducto.Text.ToUpper();
        int index = this.lstTienda.Items.IndexOf(producto);
        if (index == -1)
        {
            this.lstTienda.Items.Add(producto);
            this.txtProducto.SelectAll();
            this.txtProducto.Focus();
        }
        else
        {
            this.lstTienda.SelectedIndex = index;
        }
    }

    private void btnEliminar_Click(object sender, EventArgs e)
    {
        int numElementos = this.lstTienda.SelectedIndices.Count;
        for (int i = numElementos - 1; i >= 0; i--)
        {
            int index = this.lstTienda.SelectedIndices[i];
            this.lstTienda.Items.RemoveAt(index);
        }
    }

    private void btnBorrarTodos_Click(object sender, EventArgs e)
    {
        this.lstTienda.Items.Clear();
    }

    private void btnSeleccionados_Click(object sender, EventArgs e)
    {
        ...
    }
}

```

```

//ELIMINAMOS DE LA TIENDA LOS SELECCIONADOS
int numElementos = this.lstTienda.SelectedIndices.Count;
for (int i = numElementos - 1; i >= 0; i--)
{
    int index = this.lstTienda.SelectedIndices[i];
    string producto = this.lstTienda.SelectedItems[i].ToString();
    this.lstAlmacen.Items.Add(producto);
    this.lstTienda.Items.RemoveAt(index);
}

private void btnTodos_Click(object sender, EventArgs e)
{
    //TENEMOS UN METODO LLAMADO AddRange EN LAS COLECCIONES
    //QUE PERMITEN AGREGAR UN CONJUNTO A LA VEZ
    this.lstAlmacen.Items.AddRange(this.lstTienda.Items);
    this.lstTienda.Items.Clear();
}

private void btnSubir_Click(object sender, EventArgs e)
{
    int index = this.lstAlmacen.SelectedIndex;
    string producto = this.lstAlmacen.SelectedItem.ToString();
    this.lstAlmacen.Items.RemoveAt(index);
    this.lstAlmacen.Items.Insert(index - 1, producto);
    this.lstAlmacen.SelectedIndex = index - 1;
}

private void btnBajar_Click(object sender, EventArgs e)
{
    int index = this.lstAlmacen.SelectedIndex;
    string producto = this.lstAlmacen.SelectedItem.ToString();
    this.lstAlmacen.Items.RemoveAt(index);
    this.lstAlmacen.Items.Insert(index + 1, producto);
    this.lstAlmacen.SelectedIndex = index + 1;
}

private void lstAlmacen_SelectedIndexChanged(object sender, EventArgs e)
{
    int index = this.lstAlmacen.SelectedIndex;
    if (index == 0)
    {
        this.btnSubir.Enabled = false;
    }
    else
    {
        this.btnSubir.Enabled = true;
    }
    if (index == this.lstAlmacen.Items.Count - 1)
    {
        this.btnBajar.Enabled = false;
    }
    else
    {
        this.btnBajar.Enabled = true;
    }
}
}

```

METODOS DE CLASE

Un método es un conjunto de acciones.

Dentro de la categoría de métodos tenemos 2/3:

Los métodos en C# utilizan cada letra de cada palabra en mayúscula

- 1) Métodos de acción **void**. Son métodos que solamente realizan acciones

```

void LimpiarCajas() {
    this.txt1.Text = "";
    this.txt2.Text = "";
    this.txt3.Text = "";
}

```

En cualquier otro método, realizamos la llamada:

```

void button1_Click() {
    this.LimpiarCajas();
}

```

- 1) Métodos **return**. Son métodos que ejecutan una serie de acciones y devuelven
Un valor al lugar donde han sido invocados.

```

TipoDatos NombreMetodo() {
    //ACCIONES
    return ValorTipoDatos;
}

```

```

1 reference | 0 changes | 0 authors, 0 changes
int SumarNumeros(int num1, int num2)
{
    int suma = num1 + num2;
    return suma;
}

```

```

1 reference | 0 changes | 0 authors, 0 changes
private void btnNuevo_Click(object sender, EventArgs e)
{
    int total = this.SumarNumeros(7, 9);
}

```

- 1) Métodos **delegate**. Estos métodos son **void** pero están delegados a la escucha de
Un objeto, por ejemplo para entenderlos, son **eventos**.
Estos métodos siempre reciben los datos:

- **Sender:** Es el objeto que realiza la llamada al método
- **EventArgs:** Es la variable de argumentos del Evento. Dependiendo del evento Podría tener información útil

En C# los tipos de dato **Wrapper/Primitivos** se envían como VALOR.

El resto de objetos, se envían como Referencia.

Estamos hablando de parámetros en un método.

Una referencia es un objeto accesible desde múltiples variables, siendo el mismo objeto.

```
Button[] botones = new Button[2];
botones[0] = this.btnPulsar;
```

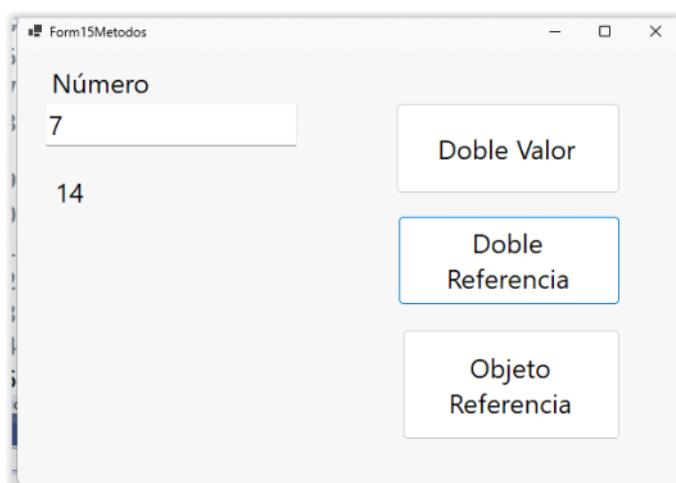
```
//AHORA MISMO, TENEMOS DOS REFERENCIAS AL OBJETO
this.btnPulsar.BackColor = Color.Yellow;
botones[0].BackColor = Color.Green;
```

También podemos enviar variables Wrapper/Primitivas como referencia.

Aunque esto se utiliza en pocas ocasiones, puede llegar a ser muy útil en programación.

Vamos a realizar un formulario para ver la diferencia entre Valor y Referencia en los métodos

Creamos un nuevo formulario llamado **Form15Metodos**



Número y num son variables diferentes en distintos espacios de memoria
Solamente se utiliza el VALOR

```
//RECIBIMOS UN NUMERO WRAPPER POR VALOR
1 reference | 0 changes | 0 authors, 0 changes
void GetDobleValor(int num)
{
    num = num * 2;
}
```

```
private void btnDobleValor_Click(object sender, EventArgs e)
{
    int numero = int.Parse(this.txtNumero.Text);
    this.GetDobleValor(numero);
    this.lblResultado.Text = numero.ToString();
}
```

A continuación, vamos a visualizar los objetos por Referencia.

Los botones cambian de color, son enviados como Referencia, es decir,
La variable Button boton y las variables btnDoble son iguales.

```

2 references | 0 changes | 0 authors, 0 changes
void CambiarColor(Button boton)
{
    boton.BackColor = Color.LightGreen;
}

1 reference | 0 changes | 0 authors, 0 changes
private void btnObjetoReferencia_Click(object sender, EventArgs e)
{
    this.CambiarColor(this.btnDobleReferencia);
    this.CambiarColor(this.btnDobleValor);
}

```

También tendremos la posibilidad de enviar tipos Wrapper/Primitivos como referencia.

Para enviar las variables de referencia se utiliza la palabra **ref** en los métodos y Peticiones.

De esa forma, si la variable Wrapper cambia, también cambia el objeto de la petición.

```

void GetDobleReferencia(ref int num)
{
    num = num * 2;
}

1 reference | 0 changes | 0 authors, 0 changes
private void btnDobleReferencia_Click(object sender, EventArgs e)
{
    int numero = int.Parse(this.txtNumero.Text);
    //LAS DOS VARIABLES APUNTAN AL MISMO ESPACIO
    //DE MEMORIA
    this.GetDobleReferencia(ref numero);
    this.lblResultado.Text = numero.ToString();
}

```

Por último, en realidad, nunca utilizaría este código con variable de referencia, simplemente
Lo más correcto es utilizar **return**

```

int GetDoble(int num)
{
    return num * 2;
}

1 reference | 0 changes | 0 authors, 0 changes
private void btnDobleReferencia_Click(object sender, EventArgs e)
{
    int numero = int.Parse(this.txtNumero.Text);
    numero = this.GetDoble(numero);
    //LAS DOS VARIABLES APUNTAN AL MISMO ESPACIO
    //DE MEMORIA
    //this.GetDobleReferencia(ref numero);
    this.lblResultado.Text = numero.ToString();
}

```

METODOS DELEGADOS

Un método delegado es un método de Evento. Es un void y recibe dos parámetros
Siempre: Objeto de la llamada y la variable de argumentos del método.

Dependiendo del tipo de Evento, tendrá información o no.

Truco: Si es **EventArgs** no tiene nada.

```

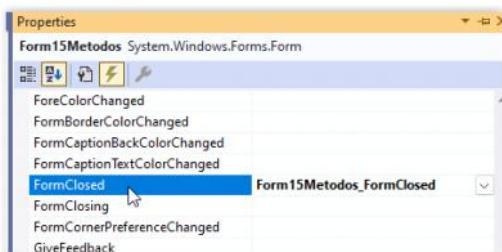
private void MetodoEvento(object sender, EventArgs e)
{
}

```

e.

- Equals
- GetHashCode
- GetType
- ToString

Dependiendo del tipo de método, la información puede ser útil.
En la ventana de propiedades tenemos un Rayo que indica que podemos recuperar cualquier Evento.



```

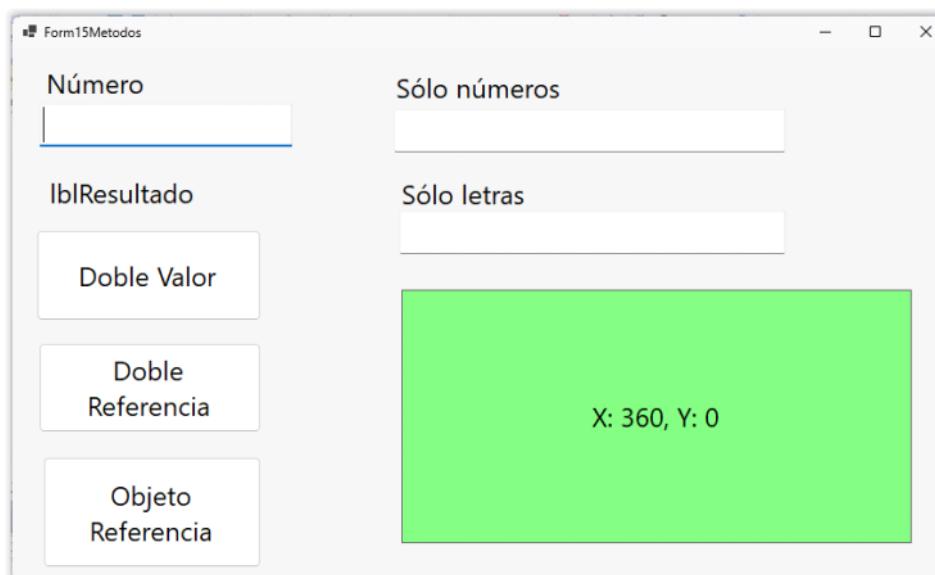
private void Form15Metodos_FormClosed(object sender, FormClosedEventArgs e)
{
}

```

e.

CloseReason

Vamos a incluir más funcionalidad dentro del mismo formulario.



CODIGO FORMULARIO

```

public partial class Form15Metodos : Form
{
    public Form15Metodos()
    {
        InitializeComponent();
    }

    //RECIBIMOS UN NUMERO WRAPPER POR VALOR
    void GetDobleValor(int num)
    {
        num = num * 2;
    }

    private void btnDobleValor_Click(object sender, EventArgs e)
    {
        int numero = int.Parse(this.txtNumero.Text);
        this.GetDobleValor(numero);
        this.lblResultado.Text = numero.ToString();
    }

    void CambiarColor(Button boton)
    {
        boton.BackColor = Color.LightGreen;
    }

    private void btnObjetoReferencia_Click(object sender, EventArgs e)
    {
        this.CambiarColor(this.btnDobleReferencia);
        this.CambiarColor(this.btnDobleValor);
    }
}

```

```

void GetDobleReferencia(ref int num)
{
    num = num * 2;
}

int GetDoble(int num)
{
    return num * 2;
}

private void btnDobleReferencia_Click(object sender, EventArgs e)
{
    int numero = int.Parse(this.txtNumero.Text);
    numero = this.GetDoble(numero);
    //LAS DOS VARIABLES APUNTAN AL MISMO ESPACIO
    //DE MEMORIA
    //this.GetDobleReferencia(ref numero);
    this.lblResultado.Text = numero.ToString();
}

private void Form15Metodos_FormClosed(object sender, FormClosedEventArgs e)
{
}
}

private void lblRaton_MouseMove(object sender, MouseEventArgs e)
{
    this.lblRaton.Text = "X: " + e.X + ", Y: " + e.Y;
}

private void txtSoloNumeros_KeyPress(object sender, KeyPressEventArgs e)
{
    //COMO NO PODEMOS ELIMINAR, DEBEMOS TAMBIEN ADMITIR
    //LA TECLA SE BORRAR (TENEMOS UN CHAR), DEBEMOS SABER
    //QUE TECLA TRADUCIDA A CHAR CORRESPONDE CON LA DE BORRAR
    //EXISTE UNA ENUMERACION LLAMADA Keys QUE NOS OFRECE LOS
    //CÓDIGOS ASCII DE TODAS LAS TECLAS
    char teclaBorrar = (char)Keys.Back;

    if (char.IsDigit(e.KeyChar) == false
        && e.KeyChar != teclaBorrar)
    {
        e.Handled = true;
    }
}

private void txtSoloLetras_KeyPress(object sender, KeyPressEventArgs e)
{
    char teclaBorrar = (char)Keys.Back;

    if (char.IsLetter(e.KeyChar) == false
        && e.KeyChar != teclaBorrar)
    {
        e.Handled = true;
    }
}
}

```

COLECCIONES NO GRAFICAS

Todas las colecciones son iguales. Lo único es que la colección gráfica nos permite Visualizar lo que hacemos.
Lo que nos importa a nosotros es cada objeto individual.

El namespace de las colecciones es **System.Collections**

Tenemos dos tipos de colecciones:

- Colecciones NO tipadas:
 - Es una colección que no contiene tipado en su interior, todos los elementos Son OBJECT y debemos realizar CASTING para poder acceder a las Características de cada objeto real del interior de la colección Podemos almacenar múltiples tipos de objeto Esta colección se llama **ArrayList**
Casting: (CLASE)objeto;

```

ArrayList elementos = new ArrayList();
elementos.Add("soy un texto");
elementos.Add(12345);
elementos.Add(this.btnAcción);

```

Aun así, para acceder a cada elemento individual de la colección, debemos Convertirlo para poder utilizar sus métodos o propiedades.

```

elementos[0].ToUpper(); //ESTO DA ERROR PORQUE NO RECONOCE EL TIPADO
Si necesitamos acceder a sus métodos o propiedades, debemos convertir a su
Tipo definido (string)

elementos[0].ToString().ToUpper()

```

Si deseamos acceder a la posición 2 del botón y cambiar su color, debemos hacer Un Casting entre clases.

```

elementos.Add(this.btnAcción);--> 2
((Button)elementos[2]).BackColor = Color.Yellow;

```

También podemos almacenar el objeto en una variable:

```
Button boton = (Button)elementos[2];
```

CONCEPTO DE ABSTRACCION

La abstracción es un concepto de POO dónde indica que, a partir de cualquier objeto, Todos pertenecen a una familia. La abstracción nos permite reconocer un objeto y situarlo dentro de una familia.

Este concepto llevado a programación:

```

ArrayList botones = new ArrayList();
botones.Add(this.btn1);

```

```
botones.Add(this.btn2);
botones.Add(this.btn3);
```

Si deseamos recorrer todos los objetos de la colección, utilizamos la clase **Button**

```
foreach (Button boton in botones) {
    boton.BackColor = Color.Green;
}
```

¿Qué sucede si dentro de la colección incluyo otros controles distintos a Button?

```
ArrayList botones = new ArrayList();
botones.Add(this.btn1);
botones.Add(this.btn2);
botones.Add(this.txtDato);
```

Ya no podemos recorrerlos con la clase **Button**, debido a que NO son botones.

Debemos abstraernos y buscar otra familia.

```
foreach (Button boton in botones) {
    boton.BackColor = Color.Green;
}
```

Todos los controles de Forms tienen una misma familia llamada **Control**

Todo objeto gráfico hereda de la clase **Control**

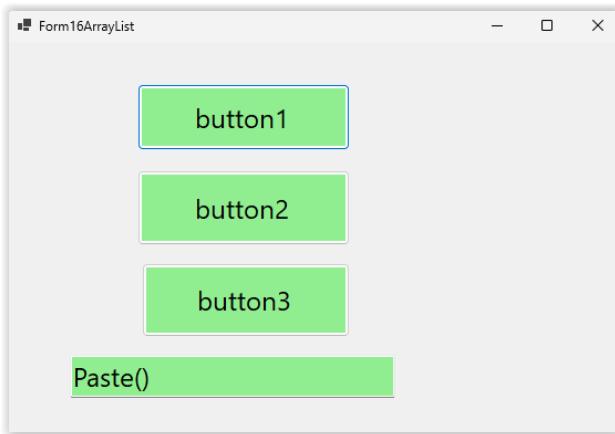
```
foreach (Control miControl in botones) {
    miControl.BackColor = Color.Green;
}
```

Si a continuación hacemos este código:

```
ArrayList botones = new ArrayList();
botones.Add(this.btn2);
botones.Add(this.txtDato);
botones.Add("Soy un String");
botones.Add("button1");

foreach (Object objeto in botones) {
    if (objeto is Clase) { ... }
    if (objeto is string) { ... }
    if (objeto is TextBox) { ... }
}
```

Vamos a crear un nuevo formulario llamado **Form16ArrayList**



CODIGO FORMULARIO

```
public Form16ArrayList()
{
    InitializeComponent();
    //VAMOS A CREAR UNA COLECCION NO TIPADA
    //Y ALMACENAR LOS BOTONES
    ArrayList colección = new ArrayList();
    colección.Add(this.button1);
    colección.Add(this.button2);
    colección.Add(this.button3);
    //AÑADIMOS UN TEXTBOX A LA COLECCION
    colección.Add(this.textBox1);
    //SI INTENTAMOS ACCEDER A UN ELEMENTO DE LA COLECCION
    //A SUS PROPIEDADES, NO PODEMOS SIN HACER CASTING
    ((Button)colección[0]).BackColor = Color.Yellow;
    //MEDIANTE LA ABSTRACCION, PODEMOS CONVERTIR Y RECORRER
    //TODOS LOS OBJETOS DENTRO DE BUCLES DE REFERENCIA
    //foreach (Button boton in colección)
    //{
    //    boton.BackColor = Color.LightBlue;
    //}
    //SI TENGO OBJETOS DE CLASES DISTINTAS, DEBEMOS
    //ABSTRAERNOS
    foreach (Control control in colección)
    {
        control.BackColor = Color.LightGreen;
        //LA CLASE TEXTBOX CONTIENE UN METODO Paste()
        //PARA PEGAR EL CONTENIDO DEL PORTAPAPELES EN
        //EL INTERIOR DEL CONTROL.
        //DEBEMOS PREGUNTAR POR EL TIPO DEL CONTROL
        if (control is TextBox)
        {
            ((TextBox)control).Paste();
        }
    }
}
```

COLECCIONES GENERICAS

Las colecciones genéricas son colecciones que contienen un tipado en los objetos.

La abstracción se realiza en el tipado de la colección.

Están dentro del namespace de **System.Collections.Generic**

El tipado indicará los objetos a almacenar en su interior.

Es más fácil trabajar con ellos porque nos quitamos de errores.

Los errores aparecerán en compilación, no en ejecución

Optimizan mucho mejor la memoria porque la colección "sabe" lo que tiene que almacenar.

Un genérico se define mediante la letra **T**

Se utilizan los diamantes para reconocer el tipado genérico <T>

T es cualquier clase. La gran diferencia está en que T es reconocida por el programa y

Nos ofrece sus propiedades y métodos, no tenemos que hacer Casting.

object cosa;

cosa = "Soy un texto";

cosa.ToUpper();

cosa = this.btn1;

cosa.BackColor = Color.Green;

Con un genérico

T cosa;

cosa = "Soy un texto";

cosa.ToUpper();

cosa = this.btn1;

cosa.BackColor = Color.Green;

La colección genérica dentro de Net Core se llama **List<T>**

```
List<Coche> coches = new List<Coche>();
coches.Add(Audi);
coches.Add(4x4);
//LA VENTAJA ESTA EN LA ABSTRACTION Y EL COMPILADOR
coches.Add(lancha);
```

La mayor ventaja está en que no necesitamos Casting, ya son reconocibles los objetos

```
coches[0].Arrancar();
```

```
//DECLARAMOS UNA COLECCION DE Button
List<Button> botones = new List<Button>();
botones.Add(this.button1);
botones.Add(this.button2);
botones.Add(this.button3);
//EL OBJETO ES RECONOCIBLE POR SU TIPADO T
botones[0].BackColor = Color.AliceBlue;
//TENEMOS ERRORES DE COMPIILACION
botones.Add(this.textBox1);
```

DELEGATE/LISTENER EN COLECCIONES

Vamos a realizar métodos delegados/listener por código para los objetos de la colección.

Esto podemos hacerlo sin problemas sin necesidad de colección, pero vamos a combinar

Para jugar más.

Lo primero será tener un Método delegado en nuestro código:

```
void MetodoDelegado(object sender, EventArgs e)
{
}
```

Para asociar un Evento al método se realiza con la siguiente sintaxis:

```
Objeto.Accion += MetodoDelegado;
```

El diseñador también puede generar los métodos dinámicamente mediante la tecla TAB

```
this.button1.Click += MetodoDelegado;
```

```
this.button1.Click +=
```

Button1_Click; (Press TAB to insert)

Vamos a crear un nuevo Form llamado **Form17ListDelegados**



CODIGO FORMULARIO

```
public partial class Form17ListDelegados : Form
{
    //DECLARAMOS UNA VARIABLE CONTADOR
    int contador;
    List<Button> botones;

    public Form17ListDelegados()
    {
        InitializeComponent();
        this.contador = 0;
        this.botones = new List<Button>();
        //this.button1.Click += BotonPulsado;
        //SI ALMACENAMOS TODOS LOS OBJETOS DENTRO
        //DE LA COLECCION, PODEMOS HACER LA ACCION
        //DELEGADA CON UN FOREACH
        this.botones.Add(this.button1);
        this.botones.Add(this.button2);
        this.botones.Add(this.button3);
        this.botones.Add(this.button4);
        foreach (Button boton in this.botones)
        {
            boton.Click += BotonPulsado;
        }
    }

    //QUIERO QUE AL PULSAR CUALQUIER BOTON EN ESTE METODO
    //CAMBIE DE COLOR EL BOTON PULSADO
    void BotonPulsado(object sender, EventArgs e)
    {
        this.contador += 1;
        this.txtContador.Text = this.contador.ToString();
        //sender es el objeto que realiza la llamada
        Button miBoton = (Button)sender;
        miBoton.BackColor = Color.Yellow;
    }
}
```

Utilizaremos siempre nuestras propias colecciones.
Dentro de este entorno tenemos una colección llamada **Controls** que nos permite acceder a todos los controles del Formulario por código

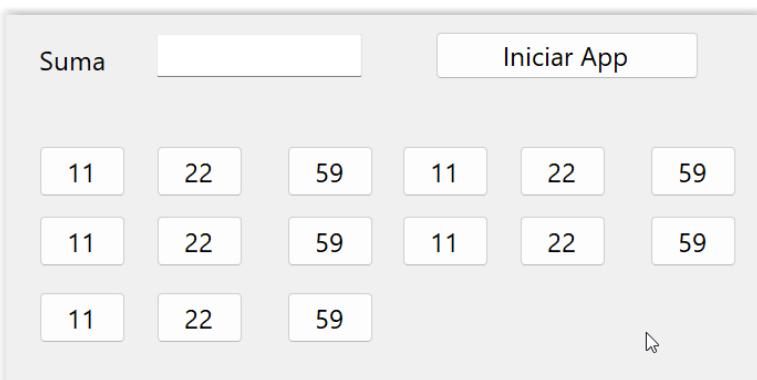
Esto nos puede servir para **rellenar** la colección dinámicamente.

Ejemplo:

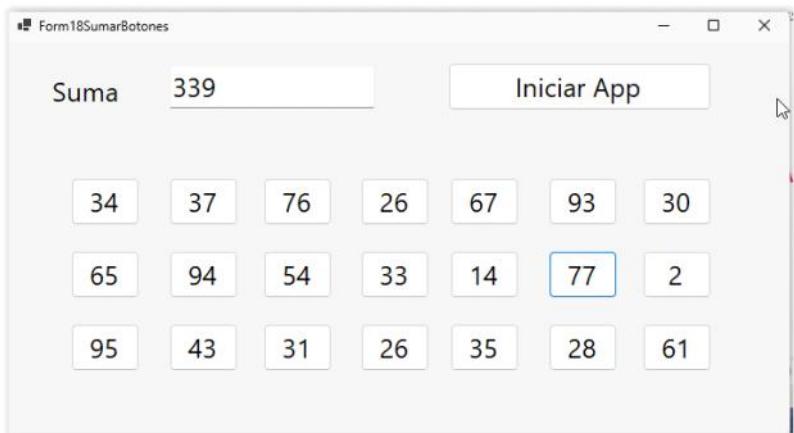
```
//CON LA COLECCION CONTROLS PODREMOS RECORRER UN CONJUNTO
//DE OBJETOS Y RELLENAR DINÁMICAMENTE NUESTRAS COLECCIONES
//PROPIAS
foreach (Control miControl in this.Controls)
{
    //DEBEMOS PREGUNTAR POR LOS BUTTON
    if (miControl is Button)
    {
        this.botones.Add((Button)miControl);
    }
}
```

Creamos un nuevo formulario llamado **Form18SumarBotones**

Si necesitamos separar Grupos de controles para nuestro For Each
Se realiza mediante Containers.
Los controles que pongamos dentro de un Container, los podemos recorrer
Con **Controls** de dicho Container.



Al iniciar la aplicación, almacenamos todos los controles Button del Panel dentro de una colección propia
Cada botón tendrá un número Random que generaremos al pulsar sobre **Iniciar App**
Cuando pulsemos un botón, iremos sumando cada valor del botón pulsado.



CODIGO FORMULARIO

```
public partial class Form18SumarBotones : Form
{
    List<Button> botones;
    int suma;

    public Form18SumarBotones()
    {
        InitializeComponent();
        this.botones = new List<Button>();
        this.suma = 0;
        //ALMACENAMOS TODOS LOS CONTROLES DEL PANEL
        //DENTRO DE NUESTRA COLECCION
        foreach (Button boton in this.panel1.Controls)
        {
            this.botones.Add(boton);
            //DELEGAMOS LA ACCION DE LOS BOTONES
            boton.Click += SumarNumeros;
        }
    }

    void SumarNumeros(object sender, EventArgs e)
    {
        //CAPTURAMOS EL BOTON
        Button boton = (Button)sender;
        int numero = int.Parse(boton.Text);
        this.suma += numero;
        this.txtSuma.Text = suma.ToString();
    }

    private void btnIniciarApp_Click(object sender, EventArgs e)
    {
        Random random = new Random();
        this.suma = 0;
        this.txtSuma.Text = suma.ToString();
        //RECORREMOS TODOS LOS BOTONES DE MI COLECCION
        foreach (Button boton in this.botones)
        {
            int numAleat = random.Next(1, 99);
            boton.Text = numAleat.ToString();
        }
    }
}
```

Quiero lo mismo, pero utilizando controles CheckBox

Creamos un nuevo formulario llamado **Form19SumarCheckbox**

Al iniciar la aplicación, generamos números aleatorios para cada Checkbox

Tendremos un botón para iniciar los números aleatorios de nuevo.

Al pulsar sobre un Checkbox, sumamos el valor del texto y, al deseleccionar,

Restamos el valor del texto.

Visualizaremos en una caja el resultado.

PROGRAMACION ORIENTADA A OBJETOS

Dentro de POO tenemos cuatro reglas independiente al lenguaje

- Abstracción: Ya la conocemos, implica reconocer un objeto por sus características
- Encapsulación: Mediante los modificadores de acceso, permitimos o impidimos la visualización de objetos en las clases
- Polimorfismo: Un método puede tener múltiples formas
- Herencia: Un objeto hereda de otro sus características e implementa sus métodos y propiedades

Tenemos dos formas de crear clases:

- Proyecto: Tenemos un proyecto de clases y lo utilizamos para agregar las librerías a otros proyectos.
- Nuget: Tenemos un proyecto de clases y lo publicamos para la comunidad

Nosotros para ver los conceptos, vamos a trabajar con librerías.

MODIFICADORES DE ACCESO

Un modificador de acceso es para la encapsulación y, dependiendo de cómo declaramos una variables tendrá una visibilidad u otra. Esta teoría es para la visibilidad entre los objetos de las clases.

- **public**: Visibilidad completa de las propiedades/métodos
- **private**: Visibilidad solamente para la propia clase
- **internal**: Visibilidad completa de propiedades/métodos pero solamente para las clases del proyecto.
- **protected**: Visibilidad solamente para las clases que heredan.

Cuando creamos un proyecto, se genera un **namespace**

Cuando creamos una carpeta se genera un **namespace**

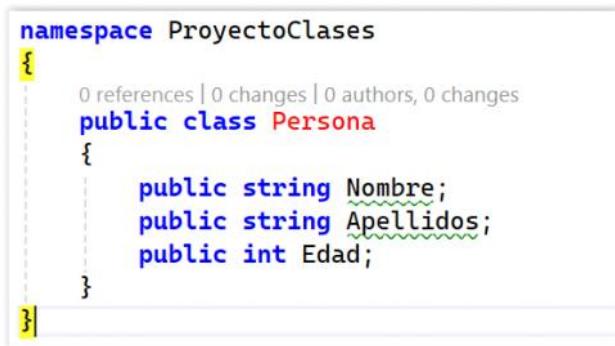
Por comodidad, vamos a crear un proyecto de clases sobre la misma solución

Creamos un nuevo proyecto de tipo **Class Library** llamado **ProyectoClases**



```
namespace ProyectoClases
{
    public class Class1
    {
    }
}
```

Comenzamos creando una nueva clase llamada **Persona**



```
namespace ProyectoClases
{
    public class Persona
    {
        public string Nombre;
        public string Apellidos;
        public int Edad;
    }
}
```

Vamos a trabajar sobre el proyecto de **Fundamentos**, solamente nos servirá para probar las clases que estamos creando.

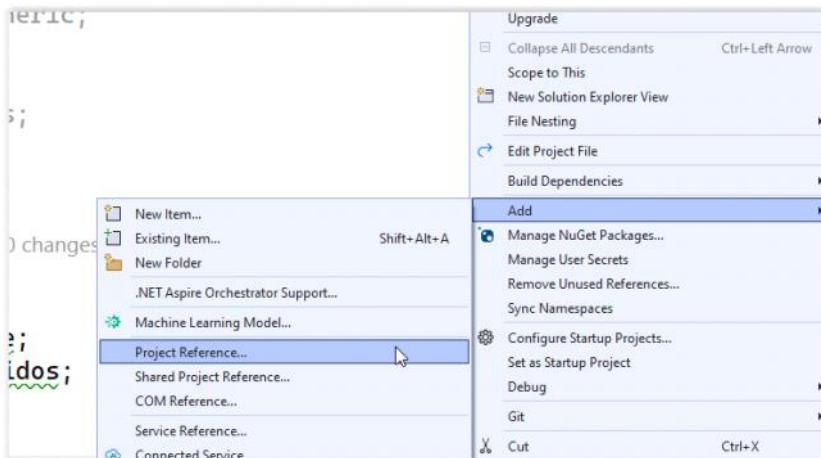
El siguiente paso es agregar nuestra librería de clases sobre el

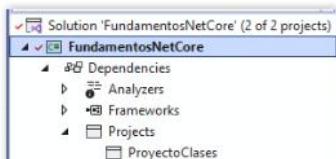
Proyecto de **Fundamentos**

Para agregar la librería, primero debemos compilar el proyecto.

Sobre el proyecto de clases, **Build**

Sobre el proyecto de **Fundamentos**, **Add Project Reference**





Sobre el proyecto **Fundamentos** creamos un nuevo Form
Llamado **Form20TestClases**

Para poder utilizar nuestra clase **Persona**, lo primero que debemos hacer
Es utilizar el **namespace**

```
using ProyectoClases;
```

```
private void btnPersona_Click(object sender, EventArgs e)
{
    Persona persona = new Persona();
    persona.Nombre = "Alumno";
    persona.Apellidos = "Navidad";
    persona.Edad = -25;
    this.lstClases.Items.Add("Nombre: "
        + persona.Nombre + ", Apellidos: "
        + persona.Apellidos);
    this.lstClases.Items.Add("Edad: " + persona.Edad);
}
```

Hemos puesto un código que es funcional, pero que en nuestro
Concepto de una Persona, no es lógico
Necesitamos algún "sitio para preguntar por la Edad que están asociando.

```
persona.Edad = -25;
```

Debemos crear una Propiedad
Lo que acabamos de crear es un **Campo/Field**

Las propiedades son las que son externas.

Definición de una Propiedad:

```
public string NombrePropiedad
{
    get{ //devolvemos el valor
        set { //recuperamos el valor con value }
    }
}
```

value es una variable interna que indica el valor que ha recibido
La Propiedad

```
persona.Edad = -25;
```

Imaginemos que hacemos este código
Estamos haciendo un bucle infinito porque la Edad está entrando en el Set de forma
Continua.

```
public int Edad
{
    get { return this.Edad; }
    set
    {
        this.Edad = value;
    }
}
```

Para las propiedades se utilizan campos que manejan la propiedad
En **get** y **set**.

Dichos campos son privados y solamente sirven para la propiedad.
Se suelen denominar los campos de propiedad con guión bajo.

```
//CAMPO DE PROPIEDAD
private int _Edad;
2 references | 0 changes | 0 authors, 0 changes
public int Edad
{
    get { return this._Edad; }
    set
    {
        this._Edad = value;
    }
}
```

Pues vamos a crear una Propiedad solo para Edad que compruebe
Que la edad no es Benjamin Button

```
private int _Edad;
```

```

public int Edad
{
    get { return this._Edad; }
    set {
        //DEBEMOS COMPROBAR EL VALOR DE LA EDAD
        //QUE VIENE EN value
        if (value < 0)
        {
            //SI NOS DAN UN VALOR INCORRECTO
            this._Edad = 0;
        }
        else
        {
            //TODO CORRECTO
            this._Edad = value;
        }
    }
}

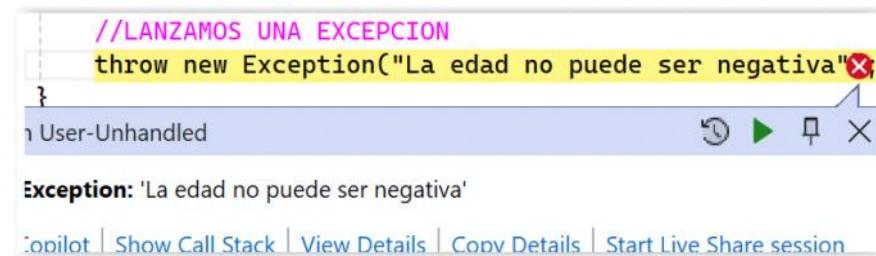
```

También tenemos la posibilidad de lanzar excepciones

```

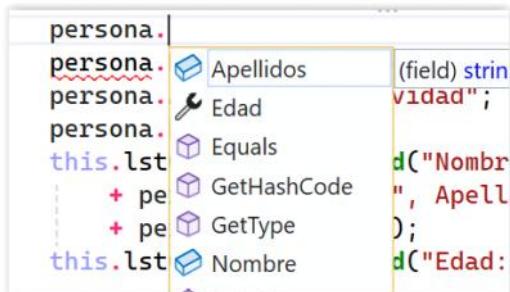
private int _Edad;
public int Edad
{
    get { return this._Edad; }
    set {
        //DEBEMOS COMPROBAR EL VALOR DE LA EDAD
        //QUE VIENE EN value
        if (value < 0)
        {
            //SI NOS DAN UN VALOR INCORRECTO
            //ERROR SILENCIOSO
            //this._Edad = 0;
            //LANZAMOS UNA EXCEPCION
            throw new Exception("La edad no puede ser negativa");
        }
        else
        {
            //TODO CORRECTO
            this._Edad = value;
        }
    }
}

```



Debemos mantener la integridad del lenguaje

Por ejemplo, ahora mismo, Nombre y Apellidos no están como
Propiedades en el lenguaje



PROPIEDADES AUTOIMPLEMENTADAS

Cuando no necesitamos comprobar nada de la propiedad,
Podemos declarar la propiedad en una sola línea, sin campo.

```
public int NombrePropiedad {get;set;}
```

```
public string Nombre { get; set; }  
public string Apellidos { get; set; }
```

PROPIEDADES ENUMERADAS

Una enumeración es una herramienta para los desarrolladores
Ofrece una serie de posibilidades para la propiedad, en realidad,
Cada posibilidad es un número interno para el programa

La declaración de una enumeración es el ÚNICO código que se
Escribe en el namespace

Por un lado, declaramos la enumeración y, por otro, la propiedad
De dicha enumeración.

Vamos a crear dos propiedades. Genero, Nacionalidad

```
namespace ProyectoClases
{
    0 references | 0 changes | 0 authors, 0 changes
    public enum TipoGenero { Masculino, Femenino}
    0 references | 0 changes | 0 authors, 0 changes
    public enum Paises { España, Francia, Alemania, Andorra, Tabarnia}
```

A continuación, escribimos las propiedades de cada tipo de enumeración

```
public class Persona
{
    0 references | 0 changes | 0 authors, 0 changes
    public TipoGenero Genero { get; set; }
    0 references | 0 changes | 0 authors, 0 changes
    public Paises Nacionalidad { get; set; }
```

```
persona.Genero = TipoGenero.Femenino;
persona.Nacionalidad = Paises.Andorra;
```

Nota: En las enumeraciones debemos comprobar siempre
El valor que nos están ofreciendo...

```
persona.Genero = (TipoGenero)99;
```

Código correcto

```
private TipoGenero _Genero;
2 references | 0 changes | 0 authors, 0 changes
public TipoGenero Genero
{
    get { return this._Genero; }
    set {
        if (value != TipoGenero.Masculino &&
            value != TipoGenero.Femenino)
        {
            throw new Exception("Valor de género incorrecto");
        }
        else
        {
            this._Genero = value;
        }
    }
}
```

METODOS

Ya hemos explicado la funcionalidad de los métodos.

Vamos a utilizar una herramienta del lenguaje simplemente por
Organización. **Region**

La herramienta **region** nos permite agrupar nuestro código por
funcionalidades.

```
public class Persona
{
    PROPIEDADES
    #region METODOS
    #endregion
}
```

```
public string GetNombreCompleto()
{
    return this.Nombre + " " + this.Apellidos;
}
```

Estamos devolviendo un método con el Nombre y Apellidos

Ahora quiero un método para devolver los Apellidos y el Nombre

```
public string GetNombreCompletoDelReves()
{
    return this.Apellidos + " " + this.Nombre;
}
```

Necesito un método para devolver el nombre y apellidos pero en Mayúscula

```
//METODO PARA NOMBRE Y APELLIDOS EN MAYUSCULA
0 references | 0 changes | 0 authors, 0 changes
public string GetNombreCompletoMayuscula()
{
    return (this.Nombre + " " + this.Apellidos).ToUpper();
}
```

Necesito un método para devolver el nombre y apellidos pero en Minúscula

```
0 references | 0 changes | 0 authors, 0 changes
public string GetNombreCompletoMinuscula()
{
    return (this.Nombre + " " + this.Apellidos).ToLower();
}
```

POLIMORFISMO

Cuando tenemos múltiples métodos (como lo anterior) que realizan Una misma acción común (Nombre completo) podríamos tener El mismo nombre de Método, pero realizando acciones diferentes

Para poder realizar Polimorfismo, debemos tener el MISMO NOMBRE DE METODO y DISTINTOS PARAMETROS.
Lo que importa es la Firma del método

```
GetNombreCompletoMinuscula()
```

Siempre que tengamos distintas firmas, podremos realizar Polimorfismo.

```
public string GetNombreCompleto()

public string GetNombreCompleto(bool mayusculas)
{
```

```
0 references | 0 changes | 0 authors, 0 changes
public string GetNombreCompleto(int orden)
```

```
0 references | 0 changes | 0 authors, 0 changes
public string GetNombreCompleto(int num1, int num2)
{
    return "";
}
```

```
0 references | 0 changes | 0 authors, 0 changes
public void GetNombreCompleto(string dato) { }
```

```
0 references | 0 changes | 0 authors, 0 changes
public void GetNombreCompleto(string dato, int num) { }
```

```
this.lstClases.Items.Add(persona.GetNombreCompleto())
this.lstClases.Items.A ▲ 2 of 6 ▼ string Persona.GetNombreCompleto(bool mayusculas)
this.lstClases.Items
```

CONSTRUCTOR DE CLASES

Un constructor es el primer método que se ejecuta en una clase.
Es el lugar donde debemos inicializar los objetos de nuestra clase.
El constructor debe llamarse como la clase
También puede tener Polimorfismo/Sobrecarga

```
public NombreClase()
{
    //ACCIONES
}
```

Cuando llamamos a la creación de una clase estamos llamando
A su constructor:

```
NombreClase clase = new NombreClase();
```

¿Qué sucede si no tenemos constructor? Se llama al constructor de
Su clase **Base**. La clase de la que hereda. (**object**)

Si deseamos que la clase NO pueda ser instanciada, se pone un
Constructor como **private**

Vamos a crear una nueva clase llamada **Direccion** y que utilizará la
Clase **Persona**

DIRECCION

```
public class Direccion
{
    public string Calle { get; set; }
    public string Ciudad { get; set; }
    public int CodigoPostal { get; set; }

    public Direccion()
    {
        Debug.WriteLine("Constructor DIRECCION sin parámetros");
    }

    public Direccion(string calle, string ciudad)
    {
        this.Calle = calle;
        this.Ciudad = ciudad;
        Debug.WriteLine("Constructor DIRECCION con dos parámetros");
    }

    public Direccion(string calle, string ciudad, int cp)
    {
        this.Calle = calle;
        this.Ciudad = ciudad;
        this.CodigoPostal = cp;
        Debug.WriteLine("Constructor DIRECCION con tres parámetros");
    }
}
```

Sobre la clase **Persona** creamos dos propiedades de la clase

Direccion

PERSONA

```
0 references | 0 changes | 0 authors, 0 changes
public Direccion Domicilio { get; set; }
0 references | 0 changes | 0 authors, 0 changes
public Direccion DomicilioVacaciones { get; set; }
```

Sobre el formulario, vamos a incluir una dirección para la Persona

FORM

```
persona.Domicilio.Calle = "Oficina principal de Correos";
persona.Domicilio.Ciudad = "Napapiiri, Finlandia";
persona.Domicilio.CodigoPostal = 96930;
this.lstClases.Items.Add
    ("Dirección: "
    + persona.Domicilio.Calle
    + ", " + persona.Domicilio.Ciudad
    + ", " + persona.Domicilio.CodigoPostal);
```

Cuando ejecutamos, nos está dando una excepción

```

persona.Domicilio.Calle = "Oficina principal de Correos";
persona.Domicilio.Ciudad = "Napapiiri, Finlandia";

```

Thrown

NullReferenceException: 'Object reference not set to an instance of an object.'

Este error nos va a perseguir desde aquí hasta AWS.

Existe la propiedad Domicilio, pero no existe la instancia de Domicilio, es decir, un **new**

Es decisión nuestra decidir si el usuario creará el objeto o Lo haremos nosotros en la clase Persona

FORM

```

persona.Domicilio = new Direccion();
persona.Domicilio.Calle = "Oficina principal de Correos";
persona.Domicilio.Ciudad = "Napapiiri, Finlandia";
persona.Domicilio.CodigoPostal = 96930;

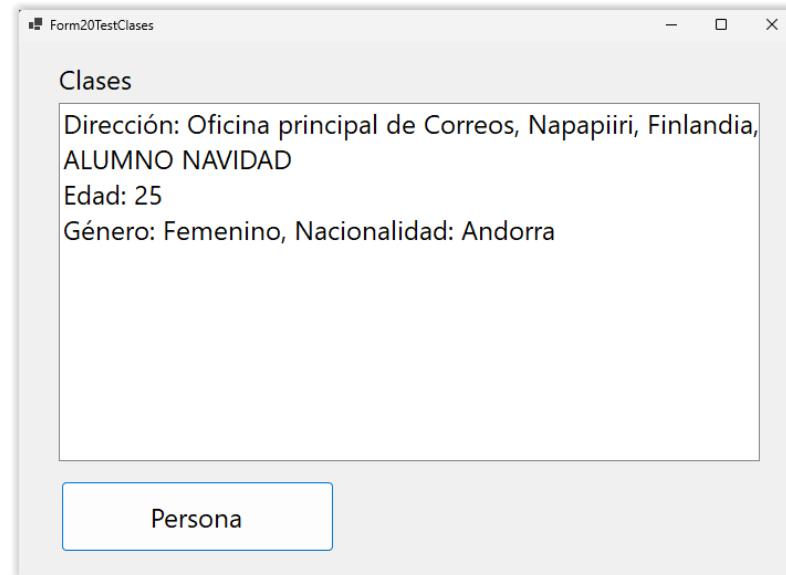
```

PERSONA

```

3 references | 0 changes | 0 authors, 0 changes
public class Persona
{
    public Persona()
    {
        this.Domicilio = new Direccion();
    }
}

```



PROPIEDADES INDIZADAS

Las propiedades indizadas nos permiten almacenar en un objeto Múltiples elementos denominados por su índice.

```
string texto = "Frozen";
```

```
texto[0] --> F;
```

Ejemplo en nuestra clase

```
Persona persona = new Persona();
persona.Nombre = "Frozen";
persona[0] --> "Es rubia";
persona[1] --> "Canta en idiomas...";
```

Sintaxis:

```
public TipoDatos this[int index]
{
    get{}
    set{}
}
```

}

Vamos a crear una propiedad dentro de **Persona**

PERSONA

```
private string _DescripcionThis;
0 references | 0 changes | 0 authors, 0 changes
public string this[int indice]
{
    get { return this._DescripcionThis; }
    set {
        Random random = new Random();
        int aleat = random.Next(1, 20);
        this._DescripcionThis = "Descripción " + aleat;
    }
}
```

No importa lo que devuelve la Propiedad

HERENCIA DE CLASES

Todo objeto hereda de una clase. Al heredar, recupera todas sus propiedades y Métodos y puede implementarlos. En realidad, es el mismo objeto, pero con más Características que podemos agregar/modificar.

Por ejemplo, un objeto Coche.

Si necesitamos una clase Deportivo, no vamos a crear de nuevo todas las propiedades Y métodos que contiene un coche, sino que cogemos un Coche y lo mejoramos.

Todas las clases heredan de **System.Object** en Net

Tenemos dos conceptos dentro de la herencia:

- **Implementación:** Agregar nuevas propiedades/métodos a la clase
- **Sobrescritura:** Modificamos el comportamiento de los métodos/propiedades y los Adaptamos.

No existe la herencia múltiple. Podemos realizar herencia múltiple mediante **Interface**.

Para heredar, debemos aplicar la siguiente sintaxis:

```
public class ClaseHereda: ClasePadre
{
    //IMPLEMENTAR
    //SOBRESCRIBIR
}
```

Lo primero que debemos saber es el comportamiento de los constructores.

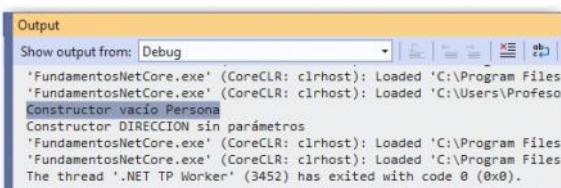
Para ello, vamos a crear una clase llamada **Empleado** que heredará de **Persona**

```
public class Persona
{
    1 reference | serraguti, 19 days ago | 1 author, 1 change
    public Persona()
    {
        Debug.WriteLine("Constructor PERSONA vacío");
        this.Domicilio = new Direccion();
    }
}
```

2 references | 0 changes | 0 authors, 0 changes

```
public class Empleado: Persona
{
}
```

Sin constructor en la clase **Empleado**, vemos que leerá el constructor de la clase **Persona**



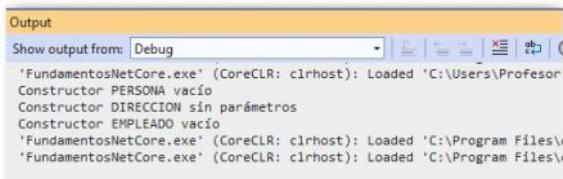
A continuación, vamos a incluir un constructor dentro de la clase **Empleado**

```

public class Empleado: Persona
{
    1 reference | 0 changes | 0 authors, 0 changes
    public Empleado()
    {
        Debug.WriteLine("Constructor EMPLEADO vacío");
    }
}

```

Cuando ejecutamos, podremos comprobar que un constructor **NO sustituye** al otro.
Para construir un **Empleado**, primero se debe construir una **Persona**.



El siguiente paso es visualizar cómo se comporta con varios constructores.
Dentro de la clase **Persona** vamos a crear un constructor que reciba dos parámetros
(nombre y apellidos)

```

public Persona(string nombre, string apellidos)
{
    Debug.WriteLine("Constructor PERSONA 2 parámetros");
    this.Nombre = nombre;
    this.Apellidos = apellidos;
}

```

Haremos lo mismo en **Empleado**

```

public Empleado(string nombre, string apellidos)
{
    Debug.WriteLine("Constructor EMPLEADO 2 parámetros");
    this.Nombre = nombre;
    this.Apellidos = apellidos;
}

```

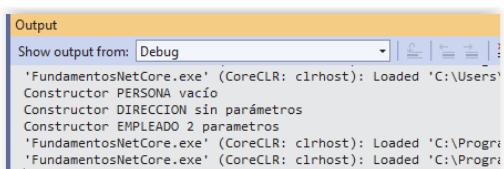
En el formulario, creamos un **Empleado** con el constructor de dos parámetros

```

Empleado empleado = new Empleado("Empleado", "Empleado");
this.lstClases.Items.Add(empleado.GetNombreCompleto());

```

Debemos comprobar qué constructor de la clase **Persona** leerá.
Veremos que leerá el constructor vacío de **Persona**.



Una clase **SIEMPRE** leerá el constructor vacío de la clase que hereda. (Si lo tiene...)

Tenemos dos posibilidades:

- 1) Queremos leer otro constructor que NO sea el vacío (Por defecto)
- 2) Que la clase **base** no tenga constructor por defecto

Podemos decir explícitamente qué constructor deseamos leer en el momento de la Herencia.

```

public class Constructor(): base()
{
}

```

Vamos a indicar, en la clase **Empleado** que leeremos el constructor de la clase **Persona** con dos parámetros

EMPLEADO

```

public Empleado():base("Nombre", "Apellidos")
{
    Debug.WriteLine("Constructor EMPLEADO vacío");
}

```

El resultado de crear un Empleado sin parámetros...

```

Output
Show output from: Debug
'FundamentosNetCore.exe' (CoreCLR: clrhost): Loaded 'C:\Program Files\dotnet\shared\Microsoft.NET Core\2.1.8\Microsoft.NETCore.App\2.1.8\mscorlib.dll'
'FundamentosNetCore.exe' (CoreCLR: clrhost): Loaded 'C:\Program Files\dotnet\shared\Microsoft.NET Core\2.1.8\Microsoft.NETCore.App\2.1.8\FundamentosNetCore.exe'
'FundamentosNetCore.exe' (CoreCLR: clrhost): Loaded 'C:\Users\Profesor MCSD Mai\OneDrive - Universidad de Alcalá\Documentos\Visual Studio 2019\Projects\FundamentosNetCore\FundamentosNetCore\bin\Debug\netcoreapp2.1\FundamentosNetCore.dll'
Constructor PERSONA 2 parámetros
Constructor EMPLEADO vacío
'FundamentosNetCore.exe' (CoreCLR: clrhost): Loaded 'C:\Program Files\dotnet\shared\Microsoft.NET Core\2.1.8\Microsoft.NETCore.App\2.1.8\mscorlib.dll'
'FundamentosNetCore.exe' (CoreCLR: clrhost): Loaded 'C:\Program Files\dotnet\shared\Microsoft.NET Core\2.1.8\Microsoft.NETCore.App\2.1.8\FundamentosNetCore.exe'
The program '[13780] FundamentosNetCore.exe: Program Trace' has exited with code 0 (0x0).
The program '[13780] FundamentosNetCore.exe' has exited with code 0 (0x0).

```

El siguiente escenario está en que la clase **Base** no tenga constructor. Es decir, la clase **Persona** no tenga un constructor vacío por defecto.

Vamos a **Persona** y comentaremos el código del constructor vacío.

```

//public Persona()
//{
//    Debug.WriteLine("Constructor PERSONA vacío");
//    this.Domicilio = new Direccion();
//}

2 references | 0 changes | 0 authors, 0 changes
public Persona(string nombre, string apellidos)
{
    Debug.WriteLine("Constructor PERSONA 2 parámetros");
    this.Nombre = nombre;
    this.Apellidos = apellidos;
}

```

Estamos forzando a las clases que hereden que deben utilizar el constructor de **Persona** con parámetros a la fuerza.

La única forma de crear un **Empleado** es crear una **Persona** con parámetros

```

1 reference | 0 changes | 0 authors, 0 changes
public Empleado() : base("Nombre", "Apellidos")
{
    Debug.WriteLine("Constructor EMPLEADO vacío");
}

0 references | 0 changes | 0 authors, 0 changes
public Empleado(string nombre, string apellidos)
:base(nombre, apellidos)
{
    Debug.WriteLine("Constructor EMPLEADO 2 parámetros");
    this.Nombre = nombre;
    this.Apellidos = apellidos;
}

```

Volvemos a dejar la clase **Persona** con su constructor por defecto.

A continuación, vamos a crear un supuesto para visualizar el concepto de Clases y Herencia.

Regla: Todo Empleado siempre cobrará 1.000 €

Lo que haremos será agregar una nueva Propiedad en Empleado.

EMPLEADO

```

public class Empleado: Persona
{
    1 reference | 0 changes | 0 authors, 0 changes
    public int SalarioMinimo { get; set; }

    1 reference | 0 changes | 0 authors, 0 changes
    public Empleado()
    {
        Debug.WriteLine("Constructor EMPLEADO vacío");
        this.SalarioMinimo = 1000;
    }
}

```

A continuación, creamos una clase llamada Director
Un Director hereda de un Empleado

Un Director cobrará 200 € más que un Empleado

```

public class Director: Empleado
{
    0 references | 0 changes | 0 authors, 0 changes
    public Director()
    {
        Debug.WriteLine("Constructor DIRECTOR");
        this.SalarioMinimo = 1200;
    }
}

```

Comprobamos si todo está correcto en el Form

```

Empleado empleado = new Empleado();
empleado.Nombre = "Empleado";
empleado.Apellidos = "Empleado";

this.lstClases.Items.Add(empleado.GetNombreCompleto());
this.lstClases.Items.Add("Salario empleado "
    + empleado.SalarioMinimo);

Director dire = new Director();
dire.Nombre = "Dire";
dire.Apellidos = "Director";
this.lstClases.Items.Add(dire.GetNombreCompleto());
this.lstClases.Items.Add("Salario Dire "
    + dire.SalarioMinimo);

```

Un empleado cobra 1.000
Un director cobra 200 más que un Empleado 1200

Podemos cambiar, desde el Form el Salario mínimo de un Empleado.

¿Queremos dejar que se pueda realizar eso?

Una clase Form no debería poder cambiar el Salario mínimo, es una característica

Que deseamos que sea "cerrada" en la clase Empleado/Director

```

empleado.Apellidos = "Empleado";
empleado.SalarioMinimo = 4000;
this.lstClases.Items.Add(empleado.GetNombreCompleto());
this.lstClases.Items.Add("Salario empleado "
    + empleado.SalarioMinimo); I

Director dire = new Director();
dire.Nombre = "Dire";
dire.Apellidos = "Director";
this.lstClases.Items.Add(dire.GetNombreCompleto());
this.lstClases.Items.Add("Salario Dire "
    + dire.SalarioMinimo);

```

Para poder crear Propiedades/Métodos que solamente sean utilizadas dentro de las Clases: **protected**

Protected es un modificador de acceso para la herencia. Solamente las clases que hereden de un objeto, tendrán acceso a las propiedades o métodos.

Modificamos la clase **Empleado**

```
public class Empleado: Persona
{
    6 references | 0 changes | 0 authors, 0 changes
    protected int SalarioMinimo { get; set; }

    1 reference | 0 changes | 0 authors, 0 changes
    public Empleado()
    {
        Debug.WriteLine("Constructor EMPLEADO vacío");
        this.SalarioMinimo = 1000;
    }

    0 references | 0 changes | 0 authors, 0 changes
    public int GetSalarioMinimo()
    {
        return this.SalarioMinimo;
    }
}
```

La vida sube, y ahora un Empleado cobra 1400€

Un empleado cobra 1.400

Un director cobra 200 más que un Empleado 1.600

```
FUNDAMENTOSNETCORE.exe (CoreCLR: CIPROST): Loaded C:\
```

```
Constructor PERSONA vacío
```

```
Constructor DIRECCION sin parámetros
```

```
Constructor EMPLEADO vacío
```

```
Constructor DIRECTOR
```

```
-----
```

```
public class Director: Empleado
{
    1 reference | 0 changes | 0 authors, 0 changes
    public Director()
    {
        Debug.WriteLine("Constructor DIRECTOR");
        this.SalarioMinimo += 200;
    }
}
```

SOBRESCRITURA DE METODOS

Sobrescribir un método de una clase **base** implica **sustituir**

Lo adaptamos a nuestro código propio y ya no realizará el código de la clase **base**

Podemos utilizar la palabra **override** (recomendable) aunque es opcional.

Si utilizamos la palabra **override** debemos incluir en la clase **base** la palabra **virtual**

Vamos a jugar con la lógica de **Vacaciones** de Empleado y Director.

Un **Empleado** tendrá 22 días de vacaciones

EMPLEADO

```
public int GetDiasVacaciones()
{
    Debug.WriteLine("GetVacaciones() EMPLEADO");
    return 22;
}
```

A continuación, mostramos las vacaciones de Empleado y Director en el Form

```

        this.lstClases.Items.Add(empleado.GetNombreCompleto());
        this.lstClases.Items.Add("Vacaciones Empleado "
            + empleado.GetDiasVacaciones());
        this.lstClases.Items.Add("Salario empleado "
            + empleado.GetSalarioMinimo());

        Director dire = new Director();
        dire.Nombre = "Dire";
        dire.Apellidos = "Director";
        this.lstClases.Items.Add(dire.GetNombreCompleto());
        this.lstClases.Items.Add("Vacaciones Director "
            + dire.GetDiasVacaciones());
    }
}

```

Queremos que un **Director** tenga **30** días de vacaciones.

Creamos un método **override** en la clase **Director** que sustituirá el código de **Empleado**

DIRECTOR

```

//METODO SOBRESCRITO
1 reference | 0 changes | 0 authors, 0 changes
public int GetDiasVacaciones()
{
    Debug.WriteLine("GetVacaciones() DIRECTOR");
    return 30;
}

```

Como podemos comprobar, el código de **EMPLEADO** es sustituido

No es necesario utilizar la palabra **override**

Sintaxis: Vamos a incluir la palabra **virtual/override** (Buena praxis)

EMPLEADO

```

//METODO SOBRESCRITO
0 references, 0 changes, 0 authors, 0 changes
public virtual int GetDiasVacaciones()
{
    Debug.WriteLine("GetVacaciones() EMPLEADO");
    return 22;
}

```

DIRECTOR

```

//METODO SOBRESCRITO
3 references | 0 changes | 0 authors, 0 changes
public override int GetDiasVacaciones()
{
    Debug.WriteLine("GetVacaciones() DIRECTOR");
    return 30;
}

```

Si no tenemos la palabra **VIRTUAL** no podemos utilizar **override**

Seremos incapaces de saber (mala praxis) si estamos sustituyendo o creando un método
Nuevo en la clase.

Para saber si estoy sobrescribiendo y no tenemos la palabra **virtual**, también podemos
Utilizar la palabra **new** para **sobrescribir**

```

1 reference | 0 changes | 0 authors, 0 changes
public Director()
{
    Debug.WriteLine("Constructor DIRECTOR");
    this.SalarioMinimo += 200;
}

//METODO SOBRESCRITO
1 reference | 0 changes | 0 authors, 0 changes
public new int GetDiasVacaciones()
{
    Debug.WriteLine("GetVacaciones() DIRECTOR");
    return 30;
}

```

Debemos diferenciar entre métodos sobrescritos (new/override) o métodos propios de la Clase.

```

//METODO SOBRESCRITO
1 reference | 0 changes | 0 authors, 0 changes
public new int GetDiasVacaciones()
{
    Debug.WriteLine("GetVacaciones() DIRECTOR");
    return 30;
}

//METODO IMPLEMENTADO
0 references | 0 changes | 0 authors, 0 changes
public int GetDiasVacaciones(int extras)
{
    return 30 + extras;
}

```

Todo está perfecto según nuestro planteamiento.
Un empleado tiene 22 días de vacaciones
Un Director tiene 30 días de vacaciones.

En algunos ejemplos, podríamos necesitar leer el código del método de la clase **Base** que estamos sobrescribiendo.

Por ejemplo, quiero que un director tenga **8 días más de vacaciones** que un Empleado

Necesitamos llamar al método **GetVacaciones()** de Empleado primero y después Utilizar dicho valor 22.

Para llamar a un método de la clase **base**: **base.Method()**

Nota: Debe ser la primera línea del método.

Modificamos el código de **Director** y su método **GetVacaciones()** para llamar al método De **Empleado** primero.

```

//METODO SOBRESCRITO
1 reference | 0 changes | 0 authors, 0 changes
public new int GetDiasVacaciones()
{
    Debug.WriteLine("GetVacaciones() DIRECTOR");
    int vacacionesEmpleado =
        base.GetDiasVacaciones();
    return vacacionesEmpleado + 8;
}

```

En los métodos implementados podremos llamar entre métodos de la clase.

```
//METODO IMPLEMENTADO
0 references | 0 changes | 0 authors, 0 changes
public int GetDiasVacaciones(int extras)
{
    return this.GetDiasVacaciones() + extras;
}
```

ACCESO A FICHEROS

El namespace para trabajar con ficheros es **System.IO**

Tenemos todo tipo de clases, desde clases para leer Directorios hasta clases para Leer/Escribir ficheros de texto plano o Stream

DirectoryInfo: Nos permite acceder a los ficheros de un directorio o información sobre El directorio.

FileInfo: Nos permite acceder a la información de un Fichero

Clases para Leer/Escribir ficheros:

TextReader/TextWriter: Ficheros de texto plano

StreamReader/StreamWriter: Ficheros de texto plano y también bytes[]

Cuando leemos un fichero, sabemos el tiempo que va a tardar en realizar la acción?

Para poder leer ficheros necesitamos realizar las operaciones de forma asíncrona.

```
TextReader file = new TextReader("1.txt");
string contenido = file.Read();
this.txt1.Text = contenido;
```

Para realizar operaciones asíncronas tenemos dos opciones:

- 1) **Thread**: Es una clase que nos permite incluir dentro de un hilo la ejecución de un código. Simplemente llamamos a dicho Hilo y nos dará igual cuando termine.

```
public string LeerFichero()
{
    TextReader file = new TextReader("1.txt");
    string contenido = file.Read();
    return contenido;
}

public void button1_Click()
{
    Thread hilo = new Thread();
    string data = hilo.Start(LeerFichero());
    This.txt1.Text = data;
}
```

- 2) **async/await**: Se utiliza para operaciones asíncronas. Podemos realizar peticiones con La combinación de estas dos palabras en métodos **Async()**
Async se incluye dentro de los métodos
Await se incluye en las llamadas asíncronas

```
public string LeerFicheroAsync()
{
    TextReader file = new TextReader("1.txt");
    string contenido = file.Read();
    return contenido;
}

public async void button1_Click()
{
    string data = await LeerFicheroAsync();
    This.txt1.Text = data;
}
```

UTILIZACION DE USING EN CODIGO

Using es una estructura de código que se utiliza para asegurarse que un objeto Ha sido construido antes de ser utilizado. Esto nos permite tener códigos seguros sin Excepciones.

También optimiza la memoria ya que destruye el objeto al finalizar el código de **using**

Sintaxis:

```
using(Objeto) {
    //ACCESO A DICHO OBJETO
}
```

Pongamos el siguiente ejemplo:

```
public class AccesoDatos
{
    public AccesoDatos(string cadenaConexion)
    {
        //NOS CONECTAMOS
    }

    public List LeerRegistros()
    {
        //LEEMOS DATOS
    }
}
```

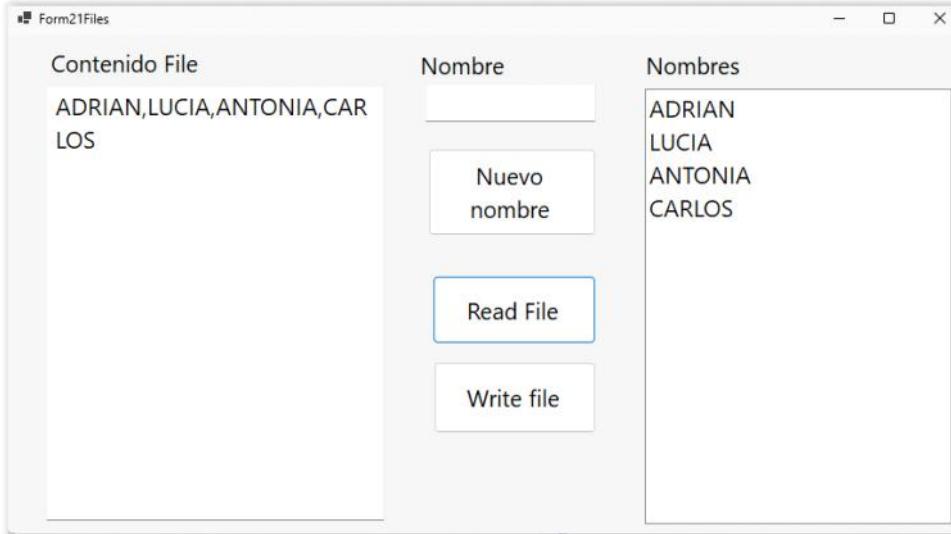
```
Al utilizar nuestra clase, este es el código que haríamos actualmente.
AccesoDatos data = new AccesoDatos("server=localhost; user id=sa....");
List colección = data.LeerRegistros();
//DIBUJAMOS...
```

El código correcto estaría en utilizar **using**

```
using (AccesoDatos data = new AccesoDatos("server=localhost; user id=sa...."))
{
    List colección = data.LeerRegistros();
}
```

Vamos a crear un ejemplo para leer textos.

Creamos un nuevo Form llamado **Form21Files**



CODIGO FORMULARIO

```
using System;
using System.IO;

public partial class Form21Files : Form
{
    private string Path { get; set; }

    public Form21Files()
    {
        InitializeComponent();
        //CUANDO HABLAMOS DE RUTAS DE FICHEROS, CON CARACTERES
        //ESPECIALES: \ TENESEMOS DOS POSIBILIDADES
        //C:\carpeta\file1.txt
        //1) UTILIZAR DOBLE CONTRABARRA
        this.Path = "C:\\carpeta\\\\file1.txt";
        //2) INDICAR QUE EL STRING SERA LITERAL: @
        this.Path = @"C:\carpeta\file1.txt";
        this.Path = "file1.txt";
    }

    private void btnNuevoNombre_Click(object sender, EventArgs e)
    {
        this.lstNombres.Items.Add(this.txtNombre.Text);
        this.txtNombre.SelectAll();
        this.txtNombre.Focus();
    }

    public string GetNombresListBox()
    {
        string data = "";
        foreach (string name in this.lstNombres.Items)
        {
            data += name + ",";
        }
        data = data.Trim(',');
        return data;
    }

    private async void btnWriteFile_Click(object sender, EventArgs e)
    {
        //TENEMOS UNA CLASE LLAMADA FileInfo NOS DEVUELVE UN FILE
        //Y PODEMOS GENERAR WRITER O READER
        FileInfo file = new FileInfo(this.Path);
        //CREAMOS EL FICHERO
        using (TextWriter writer = file.CreateText())
        {
            //RECUPERAMOS LOS NOMBRES DEL LISTBOX
            string contenido = this.GetNombresListBox();
            //ESCRIBIMOS DENTRO DEL FICHERO
            await writer.WriteLineAsync(contenido);
            //DESPUES DE ESCRIBIR EN CUALQUIER FICHERO
            //SE DEBE APLICAR EL METODO Flush()
            await writer.FlushAsync();
            //CERRAMOS EL FICHERO
            writer.Close();
            MessageBox.Show("Datos almacenados");
        }
    }

    private async void btnReadFile_Click(object sender, EventArgs e)
    {
        FileInfo file = new FileInfo(this.Path);
        using (TextReader reader = file.OpenText())
        {
            string contenido = await reader.ReadToEndAsync();
        }
    }
}
```

```

        //SIEMPRE SE LIBERAN LOS FICHEROS CON Close
        reader.Close();
        this.txtContenido.Text = contenido;
        this.RellenarListBox(contenido);
    }
}

public void RellenarListBox(string nombres)
{
    string[] data = nombres.Split(",");
    this.lstNombres.Items.Clear();
    foreach (string name in data)
    {
        this.lstNombres.Items.Add(name);
    }
}

```

A continuación, vamos a realizar la misma App pero utilizando Clases.
Nos llevaremos todo lo que NO tiene que ver con la parte gráfica a clases.
(read y write)

Sobre el **proyecto de clases**, vamos a crear una nueva carpeta llamada **Helpers** y una
Clase llamada **HelperFiles**

HELPFILES

```

public class HelperFiles
{
    //NECESITAMOS DOS METODOS: LEER Y ESCRIBIR
    //CON DICHOS METODOS DEBEMOS UTILIZAR ASYNC/AWAIT
    //CUANDO CREAMOS METODOS ASYNC/AWAIT PROPIOS, DEBEMOS
    //UTILIZAR LA CLASE Task
    //NO SE UTILIZA void
    //SI ES UN void: Task
    //SI ES UN return: Task<ClaseReturn>
    public async Task<string> ReadFileAsync(string path)
    {
        FileInfo file = new FileInfo(path);
        using (TextReader reader = file.OpenText())
        {
            string data = await reader.ReadToEndAsync();
            reader.Close();
            return data;
        }
    }

    public async Task WriteFileAsync(string path, string data)
    {
        FileInfo file = new FileInfo(path);
        using (TextWriter writer = file.CreateText())
        {
            await writer.WriteLineAsync(data);
            await writer.FlushAsync();
            writer.Close();
        }
    }
}

```

Modificamos el código del Form...

CODIGO FORMULARIO

```

private async void btnWriteFile_Click(object sender, EventArgs e)
{
    HelperFiles helper = new HelperFiles();
    string data = this.GetNombresListBox();
    await helper.WriteFileAsync(this.Path, data);
    MessageBox.Show("Datos guardados");
}

private async void btnReadFile_Click(object sender, EventArgs e)
{
    HelperFiles helper = new HelperFiles();
    string data = await helper.ReadFileAsync(this.Path);
    this.RellenarListBox(data);
    this.txtContenido.Text = data;
}

```

El objeto HelperFiles a nivel de clase

```

3 references | 0 changes | 0 authors, 0 changes
public partial class Form21Files : Form
{
    5 references | 0 changes | 0 authors, 0 changes
    private string Path { get; set; }
    private HelperFiles helper;

    1 reference | 0 changes | 0 authors, 0 changes
    public Form21Files()
    {
        InitializeComponent();
        this.helper = new HelperFiles();
        //CUANDO HABIMOS DF RUTAS DF FICHROS. CON C
    }
}

```

```

1 reference | 0 changes | 0 authors, 0 changes
private async void btnWriteFile_Click(object sender, EventArgs e)
{
    string data = this.GetNombresListBox();
    await this.helper.WriteFileAsync(this.Path, data);
    MessageBox.Show("Datos guardados");
}

1 reference | 0 changes | 0 authors, 0 changes
private async void btnReadFile_Click(object sender, EventArgs e)
{
    string data = await this.helper.ReadFileAsync(this.Path);
    this.RellenarListBox(data);
    this.txtContenido.Text = data;
}

```

Podríamos poner **static** a los métodos ya que NO utilizan el objeto HelperFiles

```

public static async Task<string> ReadFileAsync(string path)
{
    FileInfo file = new FileInfo(path);
    using (TextReader reader = file.OpenText())
    {
        string data = await reader.ReadToEndAsync();
        reader.Close();
        return data;
    }
}

```

```

1 reference | 0 changes | 0 authors, 0 changes
public static async Task WriteFileAsync(string path, string data)
{
    FileInfo file = new FileInfo(path);
    using (TextWriter writer = file.CreateText())
    {

```

```

private async void btnWriteFile_Click(object sender, EventArgs e)
{
    string data = this.GetNombresListBox();
    await HelperFiles.WriteFileAsync(this.Path, data);
    MessageBox.Show("Datos guardados");
}

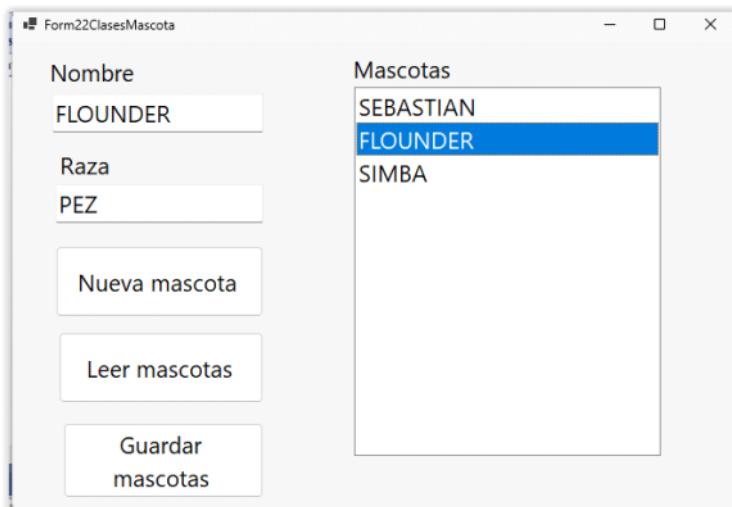
```

```

1 reference | 0 changes | 0 authors, 0 changes
private async void btnReadFile_Click(object sender, EventArgs e)
{
    string data = await HelperFiles.ReadFileAsync(this.Path);
    this.RellenarListBox(data);
    this.txtContenido.Text = data;
}

```

Vamos a realizar un ejemplo "parecido" combinando Files (HelperFiles) con Clases,
Es decir, lo que haremos será almacenar Objetos dentro del File (Mascota).



Una mascota está compuesta por su nombre y su raza.

¿Qué tenemos que modificar en nuestra lógica?

ADRIAN,LUCIA,ANTONIA

Tenemos lo siguiente: **MASCOTA**: Garfield,Gato

¿Qué tenemos que almacenar en nuestro fichero?

NOMBRE,RAZA@NOMBRE,RAZA

GARFIELD,GATO@PLUTO,PERRO

Las clases se almacenan dentro de las carpetas **Models**

Sobre el proyecto de clases creamos una nueva carpeta llamada **Models** y una clase
Llamada **Mascota**

MASCOTA

```
public class Mascota
{
    0 references | 0 changes | 0 authors, 0 changes
    public string Nombre { get; set; }
    0 references | 0 changes | 0 authors, 0 changes
    public string Raza { get; set; }
}
```

Sobre el proyecto de Forms creamos un nuevo formulario llamado **Form22ClasesMascota**

Debemos intentar abstraernos en la medida de lo posible del dibujo, de esta forma,
Podremos reutilizar nuestras clases independientemente del entorno de trabajo.

Debemos pensar qué tenemos que hacer y quitar la parte del dibujo.

NOMBRE,RAZA@NOMBRE,RAZA

GARFIELD,GATO@PLUTO,PERRO

HELPERSFILES

- Leer fichero
- Escribir Fichero

Abstraernos del dibujo

Colección de Mascotas

Convertir la colección a String

Convertir el String del Read a colección mascotas

Sobre la carpeta **Helpers** creamos una clase llamada **HelperMascotas**

HELPERMASCOTAS

```
public class HelperMascotas
{
    public List<Mascota> Mascotas { get; set; }
    private string Path;

    public HelperMascotas(string path)
    {
        this.Mascotas = new List<Mascota>();
        this.Path = path;
    }

    //Convertir la colección a String
    private string ConvertirMascotasString()
    {
        //GARFIELD,GATO@PLUTO,PERRO
        string data = "";
        foreach (Mascota mascota in this.Mascotas)
        {
            //GARFIELD,GATO
            string temp = mascota.Nombre + "," + mascota.Raza;
            data += temp + "@";
        }
    }
}
```

```

        data = data.TrimEnd('@');
        return data;
    }

    //Convertir el String del Read a colección mascotas
    private void ConvertirMascotasList(string data)
    {
        //GARFIELD,GATO@PLUTO,PERRO
        //LIMPIAMOS LA COLECCION ACTUAL
        this.Mascotas.Clear();
        //SEPARAMOS EL STRING POR CADA MASCOTA @
        string[] datosMascotas = data.Split("@");
        foreach (string stringMascota in datosMascotas)
        {
            //GARFIELD,GATO
            string[] propiedades = stringMascota.Split(",");
            Mascota mascota = new Mascota();
            mascota.Nombre = propiedades[0];
            mascota.Raza = propiedades[1];
            this.Mascotas.Add(mascota);
        }
    }

    //NECESITAMOS LEER/ESCRIBIR MASCOTAS DE ESTA COLECCION
    public async Task WriteMascotasAsync()
    {
        //CONVERTIMOS LA COLECCION A STRING
        string data = this.ConvertirMascotasString();
        await HelperFiles.WriteFileAsync(this.Path, data);
    }

    public async Task ReadMascotasAsync()
    {
        //LEEMOS EL FICHERO
        string data = await HelperFiles.ReadFileAsync(this.Path);
        this.ConvertirMascotasList(data);
    }
}

```

Por último, hacemos el código del Form

CODIGO DEL FORMULARIO

```

public partial class Form22ClasesMascota : Form
{
    private HelperMascotas helper;

    public Form22ClasesMascota()
    {
        InitializeComponent();
        this.helper = new HelperMascotas("mascotas.txt");
    }

    private void DibujarMascotas()
    {
        this.lstMascotas.Items.Clear();
        foreach (Mascota mascota in this.helper.Mascotas)
        {
            this.lstMascotas.Items.Add(mascota.Nombre);
        }
    }

    private void btnNuevaMascota_Click(object sender, EventArgs e)
    {
        Mascota mascota = new Mascota();
        mascota.Nombre = this.txtNombre.Text;
        mascota.Raza = this.txtRaza.Text;
        this.helper.Mascotas.Add(mascota);
        this.DibujarMascotas();
    }

    private async void btnLeerMascotas_Click(object sender, EventArgs e)
    {
        await this.helper.ReadMascotasAsync();
        this.DibujarMascotas();
    }

    private async void btnGuardarMascotas_Click(object sender, EventArgs e)
    {
        await this.helper.WriteMascotasAsync();
        this.lstMascotas.Items.Clear();
    }

    private void lstMascotas_SelectedIndexChanged(object sender, EventArgs e)
    {
        int index = this.lstMascotas.SelectedIndex;
        if (index != -1)
        {
            Mascota mascota = this.helper.Mascotas[index];
            this.txtNombre.Text = mascota.Nombre;
            this.txtRaza.Text = mascota.Raza;
        }
    }
}

```

SERIALIZACION

Lo que hicimos ayer es exactamente este concepto.
 Serializar es generar un objeto en un punto y recuperar dicho objeto en otro punto con
 La misma forma.
 GARFIELD,GATO@PLUTO,PERRO

Dentro de la serialización, tenemos dos elementos automáticos:

- **JSON/XML:** Esta serialización almacena un texto con un formato ya predefinido y Debemos convertir dicho formato a objetos.
- **BINARY:** Esta serialización almacena directamente el objeto a nivel de bytes y devuelve El mismo objeto, sin necesidad de hacer nada.

Vamos a visualizar un ejemplo de serialización de tipo XML.

Lo que haremos será serializar el objeto Mascota dentro de un fichero con formato XML. La serialización será automática, casi no tenemos que hacer nada.

El namespace para la serialización **System.Xml.Serialization**

Dentro del namespace tenemos una clase llamada **XmlSerializer** que es la encargada de realizar la serialización de objetos a XML

Debemos indicar, en la clase, qué objeto vamos a serializar mediante **Type**

Tenemos dos opciones para almacenar el Type:

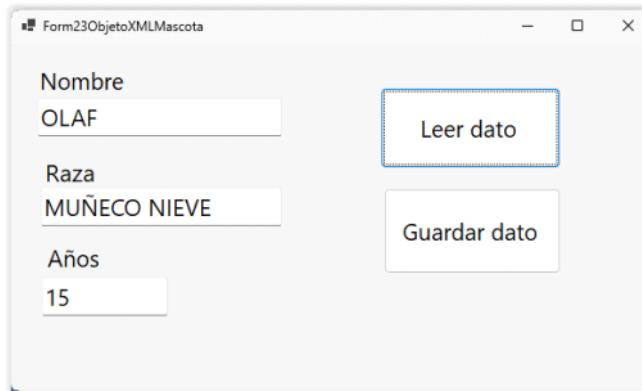
- 1) Desde el objeto: **Objeto.GetType()**
- 2) Si no tenemos objeto: **typeof(clase)**

Nota: Para la serialización solamente podemos utilizar clases propias, no podemos serializar clases de .NET Core.

Vamos a modificar la clase **Mascota** incluyendo una propiedad INT

```
public class Mascota
{
    public string Nombre { get; set; }
    public string Raza { get; set; }
    public int Years { get; set; }
}
```

Creamos un nuevo formulario llamado **Form23ObjetoXMLMascota**



```
<?xml version="1.0" encoding="utf-8"?>
<Mascota xmlns:xsi="http://www.w3.org/2001/XMLSchema"
<Nombre>OLAF</Nombre>
<Raza>MUÑECO NIEVE</Raza>
<Years>15</Years>
</Mascota>
```

CODIGO FORMULARIO

```
using System.Xml.Serialization;

public partial class Form23ObjetoXMLMascota : Form
{
    XmlSerializer serializer;
    public Form23ObjetoXMLMascota()
    {
        InitializeComponent();
        this.serializer =
            new XmlSerializer(typeof(Mascota));
    }

    private async void btnGuardarDatos_Click(object sender, EventArgs e)
    {
        Mascota mascota = new Mascota();
        mascota.Nombre = this.txtNombre.Text;
        mascota.Raza = this.txtRaza.Text;
        mascota.Years = int.Parse(this.txtYears.Text);
        //PARA GUARDAR LOS DATOS DENTRO DE UN FICHERO
        //DEBEMOS UTILIZAR LA CLASE StreamWriter
        using (StreamWriter writer = new StreamWriter("mascota.xml"))
        {
            this.serializer.Serialize(writer, mascota);
            await writer.FlushAsync();
            writer.Close();
        }
        this.txtNombre.Clear();
        this.txtRaza.Clear();
        this.txtYears.Clear();
    }

    private void btnLeerDatos_Click(object sender, EventArgs e)
    {
        Mascota mascota = null;
```

```

        using (StreamReader reader = new StreamReader("mascota.xml"))
    {
        mascota = (Mascota)
            this.serializer.Deserialize(reader);
        reader.Close();
        this.txtNombre.Text = mascota.Nombre;
        this.txtRaza.Text = mascota.Raza;
        this.txtYears.Text = mascota.Years.ToString();
    }
}

```

El siguiente paso será almacenar una colección de Mascotas.

Acabamos de almacenar una Mascota con nuestra propia Clase

```
this.serializer =
new XmlSerializer(typeof(Mascota));
```

Os he comentado que NO podemos almacenar clases de .Net puras.

List<Mascota> es una clase mía???????

```
this.serializer =
new XmlSerializer(typeof(List<Mascota>));
```

Necesitamos una clase que SEA una lista de mascotas

```

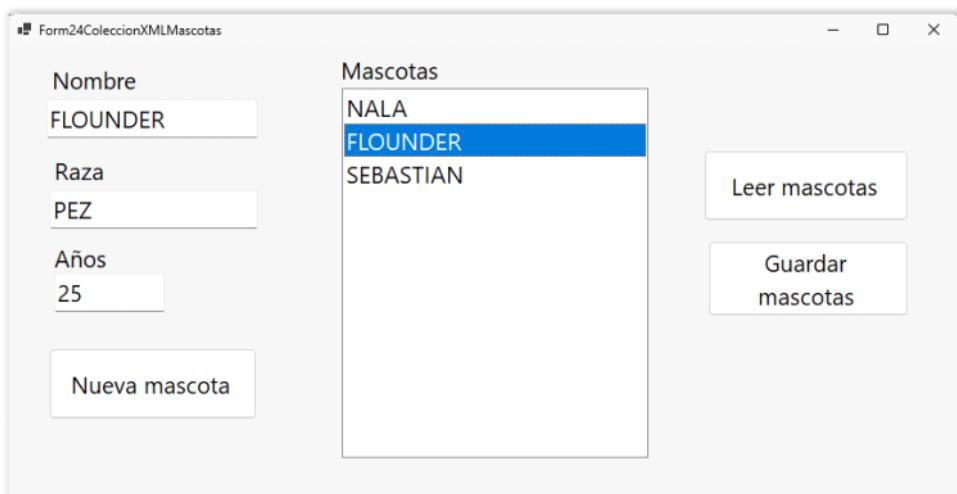
<ArrayOfMascota xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <Mascota>
        <Nombre>NALA</Nombre>
        <Raza>LEONA</Raza>
        <Years>22</Years>
    </Mascota>
    <Mascota>
        <Nombre>FLOUNDER</Nombre>
        <Raza>PEZ</Raza>
        <Years>25</Years>
    </Mascota>
    <Mascota>
        <Nombre>SEBASTIAN</Nombre>
        <Raza>CANGREJO</Raza>
        <Years>25</Years>
    </Mascota>
</ArrayOfMascota>

```

Sobre proyecto de clases **Models** creamos una nueva clase llamada **ColeccionMascotas**

```
public class ColeccionMascotas: List<Mascota>
{
}
```

Creamos un formulario llamado **Form24ColeccionXMLMascotas**



CODIGO FORMULARIO

```

public partial class Form24ColeccionXMLMascotas : Form
{
    XmlSerializer serializer;
    ColeccionMascotas mascotasList;

    public Form24ColeccionXMLMascotas()
    {
        InitializeComponent();
        this.serializer =
            new XmlSerializer(typeof(ColeccionMascotas));
        this.mascotasList = new ColeccionMascotas();
    }

    private void DibujarMascotas()
    {
        this.lstMascotas.Items.Clear();
        foreach (Mascota mascota in this.mascotasList)
        {
            this.lstMascotas.Items.Add(mascota.Nombre);
        }
    }

    private void btnNuevaMascota_Click(object sender, EventArgs e)
    {
        Mascota mascota = new Mascota();
        mascota.Nombre = this.txtNombre.Text;
        mascota.Raza = this.txtRaza.Text;
        mascota.Years = int.Parse(this.txtYears.Text);
        this.mascotasList.Add(mascota);
        this.DibujarMascotas();
        this.txtNombre.Clear();
        this.txtRaza.Clear();
        this.txtYears.Clear();
    }

    private async void btnGuardarMascotas_Click(object sender, EventArgs e)
    {
        using (StreamWriter writer = new StreamWriter("mascotaslist.xml"))
        {
            this.serializer.Serialize(writer, this.mascotasList);
            await writer.FlushAsync();
            writer.Close();
            this.lstMascotas.Items.Clear();
            this.mascotasList.Clear();
        }
    }

    private void btnLeerMascotas_Click(object sender, EventArgs e)
    {
        using (StreamReader reader = new StreamReader("mascotaslist.xml"))
        {
            this.mascotasList = (ColeccionMascotas)
                this.serializer.Deserialize(reader);
            reader.Close();
            this.DibujarMascotas();
        }
    }

    private void lstMascotas_SelectedIndexChanged(object sender, EventArgs e)
    {
        int index = this.lstMascotas.SelectedIndex;
        if (index != -1)
        {
            Mascota mascota = this.mascotasList[index];
            this.txtNombre.Text = mascota.Nombre;
            this.txtRaza.Text = mascota.Raza;
            this.txtYears.Text = mascota.Years.ToString();
        }
    }
}

```

Quiero crear un nuevo formulario llamado **Form25ColeccionMascotasJSON**

En realidad, necesitamos hacer lo mismo, almacenar las mascotas en un fichero y leer las mascotas de un fichero, pero con formato JSON.

<https://learn.microsoft.com/en-us/dotnet/standard/serialization/system-text-json/how-to>

ADO NET

jueves, 9 de enero de 2025 11:25

ADO .NET es como se denomina el acceso a datos dentro de la plataforma Net.
Esto es solamente una arquitectura de acceso a datos, dentro de Net Core existen
Múltiples formas de acceso a datos. Esta es la "tradicional" y las más parecida a los lenguajes
Back.
Conexión/Command/Statement/Reader

En este tipo de tecnología, necesitamos una conexión abierta para el acceso a datos.

Lo primero que necesitamos es ver las consultas que tenemos de SQL Server
Trabajaremos en Local con SQL Server, es decir, nuestro servidor de DESARROLLO

LOCALHOST\DESARROLLO
Password: MCSD2024

Temario ANSI SQL: Es un estándar que contienen todas las bases de datos

- Consultas básicas
- Consultas de agrupación
- Consultas de combinación
- Subconsultas
- Consultas de acción

Temario TRANSACT-SQL: Lenguaje propio de SQL Server

- Lenguaje y sintaxis
- Views, Stored Procedures

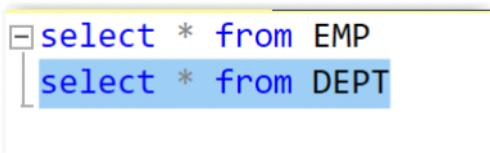
Lo primero, descargamos **HOSPITAL.TXT** de Teams y abrimos Management Studio

En este editor, los comentarios con -- o /* */

Podemos utilizar lo mismo que VS

Para ejecutar, podemos utilizar la tecla **F5** o **Execute**

Se ejecuta la línea que tengamos seleccionada



```
select * from EMP
select * from DEPT
```

CONSULTAS BASICAS

```
--UNA MISMA CONSULTA PUEDE DEVOLVER LOS MISMOS REGISTROS, PERO NO
--SER EFICIENTE (VELOCIDAD)
select * from DEPT
select DEPT_NO, DNOMBRE, LOC from DEPT --MAS EFICIENTE
--ORDENACION DE DATOS order by Y SIEMPRE AL FINAL
select * from EMP order by APELLIDO ASC --POR DEFECTO
select * from EMP order by APELLIDO DESC
--FILTRADO DE DATOS: where
--SOLAMENTE UN where EN TODA LA CONSULTA
--OPERADORES DE COMPARACION
/*
 = IGUAL
 > MAYOR
 >= MAYOR O IGUAL
 < MENOR
 <= MENOR O IGUAL
 <> DISTINTO
*/
--TODO LO QUE NO SEA NUMERO IRA ENTRE COMILLAS SIMPLES
--MOSTRAR TODOS LOS EMPLEADOS DEL DEPARTAMENTO 10
select * from EMP where DEPT_NO=10
--MOSTRAR TODOS LOS EMPLEADOS CON OFICIO ANALISTA
--SQL SERVER (DEFECTO) NO ES CASE SENSITIVE EN LOS DATOS
select * from EMP where OFICIO='ANALISTA'
select * from EMP where OFICIO='analista'
--MOSTRAR TODOS LOS EMPLEADOS QUE NO SON ANALISTAS
select * from EMP where OFICIO <> 'ANALISTA' --ANSI SQL
select * from EMP where OFICIO != 'ANALISTA' --VICIO
/*OPERADORES RELACIONALES
AND TODOS LOS FILTROS SE DEBEN CUMPLIR
OR CUALQUIER FILTRO MUESTRA DATOS
NOT JAMAS LO QUIERO VER
*/
--MOSTRAR LOS ANALISTAS QUE COBREN MAS DE 300000 DE SALARIO
select * from EMP where OFICIO='ANALISTA' and SALARIO>300000
--MOSTRAR TODOS LOS EMPLEADOS DEL DEPARTAMENTO 10 Y DEL 20
select * from EMP where DEPT_NO=10 OR DEPT_NO=20
--MOSTRAR TODOS LOS EMPLEADOS QUE NO SON ANALISTAS
select * from EMP where NOT OFICIO = 'ANALISTA'
select * from EMP where OFICIO <> 'ANALISTA'
--OTROS OPERADORES: BETWEEN
--BUSCA ENTRE DOS DATOS INCLUSIVE
--MOSTRAR TODOS LOS EMPLEADOS QUE TENGAN UN SALARIO ENTRE 104000 Y 300000
select * from EMP where SALARIO between 104000 and 300000
select * from EMP where SALARIO >= 104000 and SALARIO <= 300000
--OPERADOR IN: BUSCA COINCIDENCIAS DENTRO DE UN CAMPO CON
```

```
--MULTIPLES VALORES DE IGUALDAD
--MOSTRAR TODOS LOS EMPLEADOS DEL DEPARTAMENTO 10 Y 20 Y 30, 33, 44,55
select * from EMP where DEPT_NO=10
OR DEPT_NO=20 OR DEPT_NO=30 OR DEPT_NO=33 OR DEPT_NO=44 OR DEPT_NO=55
select * from EMP where DEPT_NO IN (10,20,30,33,44,55)
--OPERADOR NOT IN: IGUAL A IN, PERO BUSCA LOS QUE NO ESTEN
select * from EMP where NOT DEPT_NO IN (10,20) --ESTO ES UNA NEGACION, MAL
select * from EMP where DEPT_NO NOT IN (10,20) --ESTO ES OPERADOR
--OPERADOR LIKE: BUSCA EN NVARCHAR (STRING) COINCIDENCIAS
--UTILIZA UNA SERIE DE CARACTERES CLAVE
-- ? BUSCA UN CARACTER NUMERICO
-- _ BUSCA CUALQUIER CARACTER
-- % BUSCA CUALQUIER CARACTER Y LONGITUD
--MOSTRAR TODOS LOS EMPLEADOS CUYO APELLIDO COMIENZA CON A
select * from EMP where APELLIDO like 'A%'
--MOSTRAR TODOS LOS EMPLEADOS CUYO APELLIDO SEA DE 4 LETRAS
select * from EMP where APELLIDO like '____'
--CAMPOS CALCULADOS
--SON CAMPOS QUE NO EXISTEN EN LAS TABLAS Y QUE SON GENERADOS
--POR LA PROPIA CONSULTA
--TODOS LOS CAMPOS DEBEN TENER UN NAME
--EL NAME NO PUEDE ESTAR REPETIDO
--MOSTRAR EL SALARIO COMPLETO (SALARIO+COMISION) DE LOS EMPLEADOS
select APELLIDO, SALARIO, COMISION, SALARIO + COMISION AS SALARIOTOTAL
from EMP
--NO PODEMOS APLICAR FILTROS A LOS CAMPOS CALCULADOS (where)
--SI NECESITAMOS FILTRAR, DEBEMOS VOLVER A RECREAR LOS MISMOS
--DATOS
--MOSTRAR EL SALARIO COMPLETO (SALARIO+COMISION) DE LOS EMPLEADOS
--QUE COBREN MAS DE 300000
select APELLIDO, SALARIO, COMISION, SALARIO + COMISION AS SALARIOTOTAL
from EMP
where (SALARIO+COMISION) > 300000
--CLAUSULA DISTINCT
--QUITA LOS REPETIDOS DE UNA CONSULTA EN EL SELECT
select distinct OFICIO from EMP
```

CONSULTAS DE AGRUPACION

Utilizan funciones de agrupación para mostrar los resultados

Sirven para mostrar algún resumen sobre los datos.

Dicho resumen, nunca tendrá los datos de donde se ha realizado la agrupación.

Por ejemplo, contar el número de registros de una tabla.

Permiten agrupar por columnas.

Por ejemplo, contar el número de registros con oficio ANALISTA.

Nota: Campo es cualquier tipo de datos y número es algo numérico

- COUNT(*): Cuenta el número de registros, incluyendo **NULOS**
- COUNT(CAMPO): Cuenta el número de registros sin incluir nulos.
- SUM(NUMERO): Suma el total de un campo
- AVG(NUMERO): Muestra la media de un campo
- MAX(CAMPO): El valor máximo de un campo
- MIN(CAMPO): El valor mínimo de un campo

Todas las funciones de agrupación en una consulta deben tener un ALIAS

```
--CONSULTAS DE AGRUPACION
--CONTAR EL NUMERO DE REGISTROS DE LA TABLA DEPARTAMENTO
select COUNT(*) as REGISTROS from DEPT
--PODEMOS COMBINAR MULTIPLES FUNCIONES DENTRO DE UNA MISMA CONSULTA
--MOSTRAR EL SALARIO MAXIMO Y MINIMO DE LOS EMPLEADOS
select MAX(SALARIO) as SALARIOMAXIMO,
MIN(SALARIO) as SALARIOMINIMO from EMP
--LAS CONSULTAS PUEDEN REALIZARSE BASANDOSE EN UN CAMPO/S
--SE UTILIZA group by
--TRUCO: INCLUIR EN group by CADA CAMPO QUE NO SEA UNA FUNCION DE
--AGRUPACION
--QUEREMOS SABER EL NUMERO DE PERSONAS QUE TENEMOS EN CADA OFICIO
select COUNT(*) as PERSONAS, OFICIO
from EMP
group by OFICIO
--MOSTRAR EL SALARIO MAXIMO POR CADA DEPARTAMENTO Y CADA OFICIO
select MAX(SALARIO) as SALARIOMAXIMO, DEPT_NO, OFICIO
from EMP
group by DEPT_NO, OFICIO
--FILTROS EN AGRUPACIONES
--where: SE APlica SOBRE LOS CAMPOS DE LA TABLA (antes group by)
--having: SE APlica SOBRE LOS ELEMENTOS DEL SELECT (despues group by)
--MOSTRAR EL NUMERO DE PERSONAS POR DEPARTAMENTO QUE SEAN DIRECTORES
select COUNT(*) as PERSONAS, DEPT_NO
from EMP
where OFICIO='DIRECTOR'
group by DEPT_NO
--MOSTRAR EL NUMERO DE PERSONAS POR DEPARTAMENTO
--SOLAMENTE DEL 10 Y 20
select COUNT(*) as PERSONAS, DEPT_NO
from EMP
group by DEPT_NO
having DEPT_NO IN (10, 20)
--EXISTEN OPCIONES DONDE SOLAMENTE PODEMOS UTILIZA HAVING
--MOSTRAR LOS EMPLEADOS POR DEPARTAMENTO, PERO SOLAMENTE
--LOS QUE TENGAN 3 O MAS PERSONAS TRABAJANDO
select COUNT(*) as PERSONAS, DEPT_NO
from EMP
group by DEPT_NO
having COUNT(*) > 3
```

CONSULTAS DE COMBINACION (IMPORTANTES)

Las consultas de combinación nos permiten representar datos de 2 o más tablas dentro
De un mismo cursor siempre que estén relacionadas.

Dependiendo del tipo de consulta, nos mostrará unos datos u otros.

Tenemos dos tipos de sintaxis:

1) MENOS EFICIENTE

```
select TABLA1.CAMPO1, TABLA1.CAMPO2  
, TABLA2.CAMPO1, TABLA2.CAMPO2  
from TABLA1, TABLA2  
where TABLA1.CAMPORELACION=TABLA2.CAMPORELACION
```

2) JOIN LO MAS EFICIENTE

```
select TABLA1.CAMPO1, TABLA1.CAMPO2  
, TABLA2.CAMPO1, TABLA2.CAMPO2  
from TABLA1  
inner join TABLA2  
on TABLA1.CAMPORELACION=TABLA2.CAMPORELACION
```

Vamos a visualizar los datos de EMP y DEPT

El campo de relación es DEPT_NO

	EMP_NO	APELLIDO	OFICIO	DIR	FECHA_ALT	SALARIO	COMISION	DEPT_NO
1	7369	SANCHA	EMPLEADO	7902	2010-12-17 00:00:00.000	104000	0	20
2	7499	ARROYO	VENDEDOR	7698	2011-02-22 00:00:00.000	208000	39000	30
3	7521	SALA	VENDEDOR	7698	2011-02-22 00:00:00.000	162500	65000	30
4	7566	JIMENEZ	DIRECTOR	7839	2011-04-02 00:00:00.000	386750	0	20
5	7654	MARTINEZ	VENDEDOR	7698	2011-09-28 00:00:00.000	182000	182000	30

	DEPT_NO	DNOMBRE	LOC
1	10	CONTABILIDAD	ELCHE
2	20	INVESTIGACION	MADRID
3	30	VENTAS	BARCELONA
4	40	PRODUCCION	SALAMANCA

Deseamos mostrar el Apellido, oficio, nombre de departamento y localidad de los empleados

```
select EMP.APELLIDO, EMP.OFICIO  
, DEPT.DNOMBRE, DEPT.LOC  
from EMP, DEPT  
where EMP.DEPT_NO=DEPT.DEPT_NO
```

```
select EMP.APELLIDO, EMP.OFICIO  
, DEPT.DNOMBRE, DEPT.LOC  
from EMP  
inner join DEPT  
on EMP.DEPT_NO=DEPT.DEPT_NO
```

Podemos utilizar ALIAS para las tablas, por comodidad.

Si utilizamos ALIAS, se hará para TODA la consulta

```
select e.APELLIDO, e.OFICIO  
, d.DNOMBRE, d.LOC  
from EMP e  
inner join DEPT d  
on e.DEPT_NO=d.DEPT_NO
```

No es obligatorio (recomendable) incluir el nombre de la tabla precediendo al campo.

```

]select APELLIDO, OFICIO
, DEPT.DNOMBRE, DEPT.LOC
from EMP
inner join DEPT
on EMP.DEPT_NO=DEPT.DEPT_NO
|

```

Si es obligatorio si existen campos comunes entre las dos tablas dentro del select

Podemos utilizar **where** o **group by** sin problemas, siendo lo mismo.

```

--MOSTRAR EL APELLIDO, OFICIO
--Y EL NOMBRE DEL DEPARTAMENTO DE LOS EMPLEADOS
--DE VENTAS
]select EMP.APELLIDO, EMP.OFICIO
, DEPT.DNOMBRE
from EMP
inner join DEPT
on EMP.DEPT_NO=DEPT.DEPT_NO
where DEPT.DNOMBRE='VENTAS'
|

```

EJEMPLO DE GROUP BY

```

--MOSTRAR LA SUMA SALARIAL DE LOS EMPLEADOS
--POR CADA CIUDAD
]select SUM(EMP.SALARIO) as SUMASALARIAL, DEPT.LOC
from EMP
inner join DEPT
on EMP.DEPT_NO=DEPT.DEPT_NO
group by DEPT.LOC
|

```

Hemos visto el concepto de **INNER JOIN** que muestra los datos que **combinan** entre las Dos tablas.

Tenemos más tipos de JOIN dependiendo de lo que necesitemos recuperar de la consulta.

- **INNER JOIN:** Recupera los datos que combinan entre las dos tablas
- **LEFT JOIN:** Devuelve los datos que combinan entre las tablas y también los datos Que no combinan de la tabla de la izquierda
- **RIGHT JOIN:** Devuelve los datos que combinan entre las tablas y también los datos Que no combinan de la tabla de la derecha
- **FULL JOIN:** Devuelve los datos que combinan y los que no de las tablas
- **CROSS JOIN:** Devuelve el producto cartesiano, que es cada fila de una tabla con todas Las posibilidades de combinación con la otra tabla

Vamos a realizar la siguiente consulta:

SQLQuery1.sql - LO...HOSPITAL (SA (70))*

```

1 | select * from DEPT
2 | select distinct DEPT_NO from EMP

```

214 %

Results Messages

	DEPT_NO	DNOMBRE	LOC
1	10	CONTABILIDAD	ELCHE
2	20	INVESTIGACION	MADRID
3	30	VENTAS	BARCELONA
4	40	PRODUCCION	SALAMANCA

	DEPT_NO
1	10
2	20
3	30

Estamos viendo una relación normal entre dos tablas.

Tenemos 4 departamentos y existen empleados en 3 de ellos.

El departamento de **SALAMANCA** no tiene empleados.

Para probar nuestras consultas vamos a romper la relación y agregamos un Empleado que no tenga Departamento: **ALUMNO**

1111	ALUMNO	ESTUDIANTE	7839	2025-01-10 00:00:00.000	250000	0	50
------	--------	------------	------	-------------------------	--------	---	----

Ahora mismo tenemos un Departamento (SALAMANCA 40) sin empleados

Tenemos un empleado (ALUMNO) sin departamento 50

	DEPT_NO	DNOMBRE	LOC
1	10	CONTABILIDAD	ELCHE
2	20	INVESTIGACION	MADRID
3	30	VENTAS	BARCELONA
4	40	PRODUCCION	SALAMANCA

	DEPT_NO
1	10
2	20
3	30
4	50

- **RIGHT:** La tabla de la derecha es la tabla **DESPUES** del join
- **LEFT:** La tabla de la izquierda es la tabla **ANTES** del join

Mostrar el apellido, oficio y la localidad de los empleados

```

1 | --mostrar los empleados aunque no
2 | --tengan departamento
3 | select EMP.APELLIDO, EMP.OFICIO
4 | , DEPT.LOC
5 | from EMP
6 | left join DEPT
7 | on EMP.DEPT_NO = DEPT.DEPT_NO
8 |

```

214 %

Results Messages

	APELLIDO	OFICIO	LOC
14	MUÑOZ	EMPLEADO	ELCHE
15	SERRA	DIRECTOR	MADRID
16	SANTIUS...	ANALISTA	MADRID
17	FORD	VENDEDOR	MADRID
18	GUTIERR...	ANALISTA	MADRID
19	CASALES	EMPLEADO	ELCHE
20	TORMO	VENDEDOR	BARCELONA
21	ALCALA	EMPLEADO	ELCHE
22	ALUMNO	ESTUDIANTE	NULL

```

1 --mostrar los departamentos aunque no
2   --tengan empleados
3 select EMP.APELLIDO, EMP.OFICIO
4   , DEPT.LOC
5   from EMP
6   right join DEPT
7   on EMP.DEPT_NO = DEPT.DEPT_NO
8

```

Results		
APELLIDO	OFICIO	LOC
GUTIERREZ	ANALISTA	MADRID
ARROYO	VENDEDOR	BARCELONA
SALA	VENDEDOR	BARCELONA
MARTINEZ	VENDEDOR	BARCELONA
NEGRO	DIRECTOR	BARCELONA
TOVAR	VENDEDOR	BARCELONA
JIMENO	EMPLEADO	BARCELONA
TORMO	VENDEDOR	BARCELONA
NULL	NULL	SALAMANCA

Mostrar todos los datos de las tablas aunque no combinen entre ellas

```

1 --mostrar los departamentos aunque no
2   --tengan empleados
3 select EMP.APELLIDO, EMP.OFICIO
4   , DEPT.LOC
5   from EMP
6   full join DEPT
7   on EMP.DEPT_NO = DEPT.DEPT_NO
8

```

Results		
APELLIDO	OFICIO	LOC
SERRA	DIRECTOR	MADRID
SANTIUS...	ANALISTA	MADRID
FORD	VENDEDOR	MADRID
GUTIERR...	ANALISTA	MADRID
CASALES	EMPLEADO	ELCHE
TORMO	VENDEDOR	BARCELONA
ALCALA	EMPLEADO	ELCHE
ALUMNO	ESTUDIANTE	NULL
NULL	NULL	SALAMANCA

Por último, tenemos CROSS JOIN

```

select EMP.APELLIDO, EMP.OFICIO
, DEPT.LOC
from EMP
cross join DEPT

```

CONSULTA

Necesito un informe que indique cuantos empleados trabajan en cada departamento
Por su nombre de departamento.

```

4  select COUNT(EMP.EMP_NO) as EMPLEADOS
5    , DEPT.DNOMBRE
6  from EMP
7  right join DEPT
8  on EMP.DEPT_NO = DEPT.DEPT_NO
9  group by DEPT.DNOMBRE
10

```

Results

EMPLEADOS	DNOMBRE
5	CONTABILIDAD
9	INVESTIGACION
0	PRODUCCION
7	VENTAS

COMBINACION DE MAS TABLAS (3 TABLAS)

Lo único que tenemos que tener en cuenta son dos características:

- 1) Incluir en cada JOIN su correspondiente ON
- 2) El orden SI que importa en los JOIN, todo depende de las relaciones de las tablas

```

select TABLA1.CAMPO1, TABLA1.CAMPO2
, TABLA2.CAMPO1
, TABLA3.CAMPO1
from TABLA1
inner join TABLA2
on TABLA1.CAMPORELACION = TABLA2.CAMPORELACION
inner join TABLA3
on TABLA1.CAMPORELACION = TABLA3.CAMPORELACION

```

En esta base de datos tenemos tres tablas combinadas

- HOSPITAL, PLANTILLA, SALA

Mostrar el apellido, la función, nombre de hospital, dirección y nombre
De sala de los empleados de la plantilla

```

select PLANTILLA.APELLIDO, PLANTILLA.FUNCION
, HOSPITAL.NOMBRE, HOSPITAL.DIRECCION
, SALA.NOMBRE
from HOSPITAL
inner join PLANTILLA
on HOSPITAL.HOSPITAL_COD = PLANTILLA.HOSPITAL_COD
inner join SALA
on HOSPITAL.HOSPITAL_COD = SALA.HOSPITAL_COD
and PLANTILLA.SALA_COD = SALA.SALA_COD

```

EJEMPLOS

1. Seleccionar el apellido, oficio, salario, numero de departamento y su nombre de todos los empleados cuyo salario sea mayor de 300000

Apellido	Oficio	Salario	Nº DEPT	DEPARTAMENTO
JIMENEZ	DIRECTOR	386750	20	INVESTIGACION

NEGRO	DIRECTOR	370500	30	VENTAS
CEREZO	DIRECTOR	318500	10	CONTABILIDAD
GIL	ANALISTA	390000	20	INVESTIGACION
REY	PRESIDENTE	650000	10	CONTABILIDAD
FERNANDEZ	ANALISTA	390000	20	INVESTIGACION
SERRA	DIRECTOR	390000	20	INVESTIGACION

2. Mostrar todos los nombres de Hospital con sus nombres de salas correspondientes.

Nombre Sala	Nombre Hospital
Cuidados Intensivos	Provincial
Psiquiátricos	Provincial
Cuidados Intensivos	General
Cardiología	General
Recuperación	La Paz
Psiquiátricos	La Paz
Maternidad	La Paz
Cardiología	San Carlos
Recuperación	San Carlos
Maternidad	San Carlos

3. Visualizar cuantas personas realizan cada oficio en cada departamento mostrando el nombre del departamento.

Nº Departamento	Nº de personas	oficio
PRODUCCION	0	NULL
CONTABILIDAD	1	DIRECTOR
CONTABILIDAD	1	EMPLEADO
CONTABILIDAD	1	PRESIDENTE
INVESTIGACION	3	ANALISTA
INVESTIGACION	2	DIRECTOR
INVESTIGACION	2	EMPLEADO
INVESTIGACION	1	VENDEDOR
VENTAS	1	DIRECTOR
VENTAS	1	EMPLEADO
VENTAS	4	VENDEDOR

4. Contar cuantas salas hay en cada hospital, mostrando el nombre de las salas y el nombre del hospital.

Numero de salas	Sala	Hospital
1	Cardiología	General
1	Cardiología	San Carlos
1	Cuidados Intensivos	General
1	Cuidados Intensivos	Provincial
1	Maternidad	La Paz
1	Maternidad	San Carlos
1	Psiquiátricos	La Paz
1	Psiquiátricos	Provincial
1	Recuperación	La Paz
1	Recuperación	San Carlos

CONSULTAS UNION, SUBCONSULTAS, SELECT TO SELECT Y CONSULTAS A NIVEL DE FILA

Una subconsulta es una consulta que necesita de otra consulta para poder Mostrar sus resultados.

En las consultas de selección NO se suelen utilizar, pero en las consultas de Acción son muy utilizadas.

Debemos intentar evitarlas en las consultas SELECT porque generan bloqueos En el motor de la base de datos.

Bloquea las consultas hasta que devuelve los resultados de las otras consultas

No importa el número de subconsultas ni su anidamiento

```
--QUEREMOS SABER LOS DATOS DEL EMPLEADO QUE MAS COBRA DE LA EMPRESA
--NECESITAMOS SABER????? MAX(SALARIO)
select MAX(SALARIO) from EMP --650000
select * from EMP where SALARIO=650000
--SE ANIDAN LAS CONSULTAS MEDIANTE PARENTESIS PARA ALMACENAR EL RESULTADO
--QUE DEVUELVE UNA PARA CON OTRA
select * from EMP where SALARIO =
(select MAX(SALARIO) from EMP)
--MOSTRAR LOS EMPLEADOS QUE TENGAN EL MISMO OFICIO QUE SANCH
select * from EMP where OFICIO=
```

```

(select OFICIO from EMP where APELLIDO='SANCHAS')
--SI LA SUBCONSULTA DEVUELVE MAS DE UN VALOR NO PODEMOS
--UTILIZAR LA IGUALDAD
--EN SU LUGAR SE UTILIZA EL OPERADOR IN
--MOSTRAR LOS EMPLEADOS QUE TENGAN EL MISMO OFICIO QUE SANCHAS Y
--QUE ARROYO
select * from EMP where OFICIO IN
(select OFICIO from EMP where APELLIDO='SANCHAS' OR APELLIDO='ARROYO')

```

CONSULTAS UNION

Una consulta UNION muestra, en un mismo cursor, los datos de dos o más tablas Aunque no tengan relación entre sí.

Debemos tener en cuenta tres reglas:

- 1) La primera consulta es la JEFA, manda sobre el resto
- 2) Todas las consultas deben tener el mismo número de argumentos (select)
- 3) Todas las consultas deben tener el mismo tipo de dato en cada columna Correspondiente del select

En nuestra base de datos tenemos tres tablas distintas que representan a personas /trabajadores: EMP, PLANTILLA, DOCTOR

Pongamos que necesitamos representar los datos de TODAS las personas que Están en nuestra base de datos.

```

select APELLIDO, OFICIO, SALARIO from EMP
UNION
select APELLIDO, FUNCION, SALARIO from PLANTILLA
UNION
select APELLIDO, ESPECIALIDAD, SALARIO from DOCTOR

```

A continuación, queremos filtrar y mostrar solamente las personas que cobren más de 220.000

--CADA CONSULTA ES INDEPENDIENTE

```

select APELLIDO, OFICIO, SALARIO from EMP
where SALARIO > 220000
UNION
select APELLIDO, FUNCION, SALARIO from PLANTILLA
where SALARIO > 220000
UNION
select APELLIDO, ESPECIALIDAD, SALARIO from DOCTOR
where SALARIO > 220000

```

```

select APELLIDO, OFICIO, SALARIO, 'EMPLEADOS' as TABLA from EMP
UNION
select APELLIDO, FUNCION, SALARIO, 'PLANTILLA' from PLANTILLA
UNION
select APELLIDO, ESPECIALIDAD, SALARIO, 'DOCTORES' from DOCTOR

```

--LA CLAUSULA UNION ELIMINA REPETIDOS
--SI DESEAMOS MOSTRAR REPETIDOS SE UTILIZA UNION ALL

```

select APELLIDO, SALARIO from EMP
UNION ALL
select APELLIDO, SALARIO from EMP

```

SELECT TO SELECT

Sirven para realizar consultas sobre un cursor.

En ocasiones, es necesario hacer una consulta sobre otra consulta que tengamos hecha. Por ejemplo, esto es algo básico en la Paginación.

Un Select To Select parece una subconsulta pero no lo es.

Debemos incluir un ALIAS a nuestro CURSOR

```
select CAMPO1 from  
(select campo1, campo2, campo3 from tabla) QUERY  
where QUERY.CAMPO2=valor
```

```
select * from  
(select APELLIDO, OFICIO, SALARIO as SUELDO from EMP  
UNION  
select APELLIDO, FUNCION, SALARIO from PLANTILLA  
UNION  
select APELLIDO, ESPECIALIDAD, SALARIO from DOCTOR) QUERY  
where QUERY.SUELDO > 220000
```

CONSULTAS A NIVEL DE FILA

Se utilizan para modificar los datos del resultado de un cursor a nivel de select. Nunca se modifican los datos de la tabla, sino el resultado que estamos viendo.

Sirve para poder representar, desde la base de datos, los valores ya formateados.

```
select CAMPO1, CAMPO2  
case CAMPO_A_EVALUAR  
when valor1 then 'resultado1'  
when valor2 then 'resultado2'  
end as ALIAS  
from TABLA
```

```
select APELLIDO, FUNCION  
, case T  
when 'T' then 'TARDE'  
when 'N' then 'NOCHE'  
else 'MORNING'  
end as TURNO  
from PLANTILLA
```

CONSULTAS DE ACCIÓN

Modifican datos de las tablas de base de datos.

Tenemos tres tipos de instrucciones:

- INSERT: Crea nuevos registros
- UPDATE: Modifica registros
- DELETE: Elimina registros

Las consultas son las mismas en todas las bases de datos.

En SQL Server, las consultas de acción NO son transaccionales, es decir, lo que hago, Se queda hecho.

INSERT

Inserta un solo registro en la tabla. Si necesitamos insertar 5 registros, son 5 insert. Tenemos dos tipos de consultas insert:

- 1) **Todas las columnas de la tabla.** El orden de las columnas será el mismo que El de la tabla y debemos indicar solamente los valores necesarios (todos)

Sintaxis:

```
insert into TABLA values (valor1, valor2, valor3,...)
```

```
insert into DEPT values  
(50, 'INFORMATICA', 'GIJON')
```

- 1) Solamente algunos valores de la tabla. Debemos indicar el nombre de las columnas que deseamos incluir valores y los valores irán en dicho orden.

Sintaxis:

```
insert into TABLA (campo1, campo2) values (valor1, valor2)
```

```
insert into DEPT (DEPT_NO, LOC)  
values (60, 'OVIEDO')
```

Por supuesto, las SUBCONSULTAS son muy útiles dentro de las consultas de acción.
No generan bloqueos en este tipo de consultas.

Queremos insertar un nuevo departamento en TOLEDO de I+D.
En la consulta, recuperamos el máximo ID del departamento para insertar

```
insert into DEPT values  
((select MAX(DEPT_NO) + 1 from DEPT)  
, 'I+D', 'TOLEDO')
```

Tenemos unas consultas ANSI SQL para insertar múltiples registros con un solo insert.

Dichas consultas necesitan los datos de otras tablas para poder trabajar.

- 1) INSERT..INTO

```
INSERT INTO DESTINO  
SELECT * FROM ORIGEN
```

- 1) SELECT..INTO

```
SELECT * INTO DESTINO FROM ORIGEN
```

```
--1)INSERT INTO  
--LA TABLA DESTINO DEBE EXISTIR Y DEBE TENER LAS MISMAS  
--COLUMNAS QUE EL SELECT DE ORIGEN  
insert into DEPARTAMENTOS  
select * from DEPT  
  
--2) SELECT INTO  
--CREA UNA TABLA DESTINO CON LA CONSULTA DEL SELECT  
--SI LA TABLA EXISTE, NO PODEMOS EJECUTAR MAS  
select * into DEPARTAMENTOS  
from DEPT
```

DELETE

Elimina uno, varios o ningún registro con una sola instrucción.
La sintaxis es super simple:

```
DELETE FROM TABLA
```

Si la instrucción no tiene **where** elimina todos los registros de la tabla.
Es opcional, pero conveniente incluir un where.

```
DELETE FROM TABLA WHERE CONDICION
```

```
delete from DEPARTAMENTOS  
where LOC='TOLEDO'
```

```
--SI NECESITAMOS ACCEDER A UN NULL  
--SE REALIZA CON is  
delete from DEPARTAMENTOS  
where DNOMBRE is null
```

Las subconsultas son importantes dentro de las consultas de acción.

```
--DELETE LOS EMPLEADOS DEL DEPARTAMENTO  
--DE GIJON  
delete from EMP where DEPT_NO=  
(select DEPT_NO from DEPT where LOC='GIJON')
```

UPDATE

Modifica uno o varios registros con una sola consulta.
Puede modificar varios campos/columnas a la vez.

Sintaxis:

```
update TABLA set campo1=valor1, campo2=valor2
```

Esta consulta modifica todos los datos de la tabla.
Es conveniente incluir un **where**

```
--EL DEPARTAMENTO DE GIJON SE TRASLADA A SORIA  
update DEPT set LOC='SORIA'  
where LOC='GIJON'
```

```
--PODEMOS UTILIZAR EL PROPIO VALOR DE UN CAMPO PARA  
--LA MODIFICACION  
--INCREMENTAMOS EN 1 EL SALARIO DE LOS DIRECTORES  
update EMP set SALARIO=SALARIO + 1  
where OFICIO='DIRECTOR'
```

Podemos utilizar subconsultas tanto en el Set como en where.
En el set solamente pueden devolver un valor.

```
--INCREMENTAMOS EN 1 EL SALARIO DE LOS EMPLEADOS  
--DE SORIA  
update EMP set SALARIO=SALARIO + 1  
where DEPT_NO=  
(select DEPT_NO from DEPT where LOC='SORIA')
```

```
--PONER EL MISMO SALARIO QUE EL EMPLEADO ARROYO  
--A LOS EMPLEADOS DE SORIA  
update EMP set  
SALARIO=(select SALARIO from EMP where APELLIDO='ARROYO')  
where DEPT_NO IN  
(select DEPT_NO from DEPT where LOC='SORIA' OR loc='OVIEDO')
```

ADO NET

Es la tecnología de acceso a datos dentro de Net.

Dentro de .Net tenemos múltiples tecnologías para acceder a datos:

- **ADO .NET:** Es la tecnología más tradicional y la más parecida al resto de lenguajes.
- **ENTITY FRAMEWORK:** Es la última tecnología que tiene Net Core. Es una forma de conectarse "transparente", es decir, no utilizamos ni comandos ni conexiones ni nada. Directamente con las consultas y montando una estructura en nuestros proyectos.

Ado Net trabaja con proveedores, es decir, dependiendo de la base de datos, necesitamos unas librerías u otras. Dichas librerías son **nuget** que son librerías que están en un repositorio de Visual Studio y nos permiten acceder a funcionalidades.

- System.Data.SqlClient --> SQL SERVER
- System.Data.OleDb --> ACCESS
- System.Data.OracleClient --> ORACLE

Estos **namespace** son proporcionados por los **Nuget**

Objetos de ADO NET

CONNECTION

Nos permite conectar/desconectar con la base de datos.

Necesitamos una cadena de conexión para acceder a dicha base de datos. Cada base de datos tendrá una cadena de conexión diferente.

Propiedades y métodos

- ConnectionString: La cadena de conexión de acceso a datos
- Open(): Abrir la conexión con la base de datos
- Close(): Cerrar la conexión con la base de datos
- State: Enumeración con los diferentes estados de la conexión.

Eventos:

- StateChanged: Este evento se ejecuta cuando cambia el estado de la conexión
- InfoMessage: Se ejecuta cuando el servidor nos manda mensajes (PRINT)

COMMAND

El objeto Command se encarga de ejecutar las consultas SQL. Las mismas que hemos estado haciendo hoy mismo.

Propiedades y métodos

- Connection: La conexión que utilizará el comando
- CommandType: El tipo de consulta que vamos a hacer
- CommandText: El texto de la propia consulta (select..., update...)
- ExecuteReader(): Ejecuta una consulta de tipo SELECT
- ExecuteNonQuery(): Ejecuta una consulta de acción (INSERT, UPDATE, DELETE)
- Close(): Cierra el comando
- Parameters: Es una propiedad de tipo colección que contiene todos los parámetros que podemos tener en una consulta

DATAREADER

Es un lector de datos. Contendrá los datos de la consulta SELECT que hayamos realizado. Los datos de un cursor.

Es un objeto que solamente se puede recorrer una vez, una vez que los datos han sido extraídos, no podemos volver a recuperarlos a no ser que ejecutemos otra vez la consulta.

Propiedades y métodos:

- **Read():** Lee una fila del cursor. Devuelve un boolean (true/false). Cada vez que ejecutemos este método, leerá una fila de la consulta.
- **DataReader[INDICE/COLUMNA]:** Nos devuelve el dato que corresponde a la posición o el nombre de columna. Recupera siempre un string.
- **GetName(indice):** Nos devuelve el nombre de la columna en la posición que le digamos.
- **GetDataType(indice):** Nos devuelve el tipo de dato de la columna.
- **FieldCount:** Nos devuelve el número de columnas que contiene la consulta SELECT.
- **HasRows:** Indica si el cursor contiene registros o no.
- **Close():** Cierra el lector y lo libera para poder reutilizarlo.

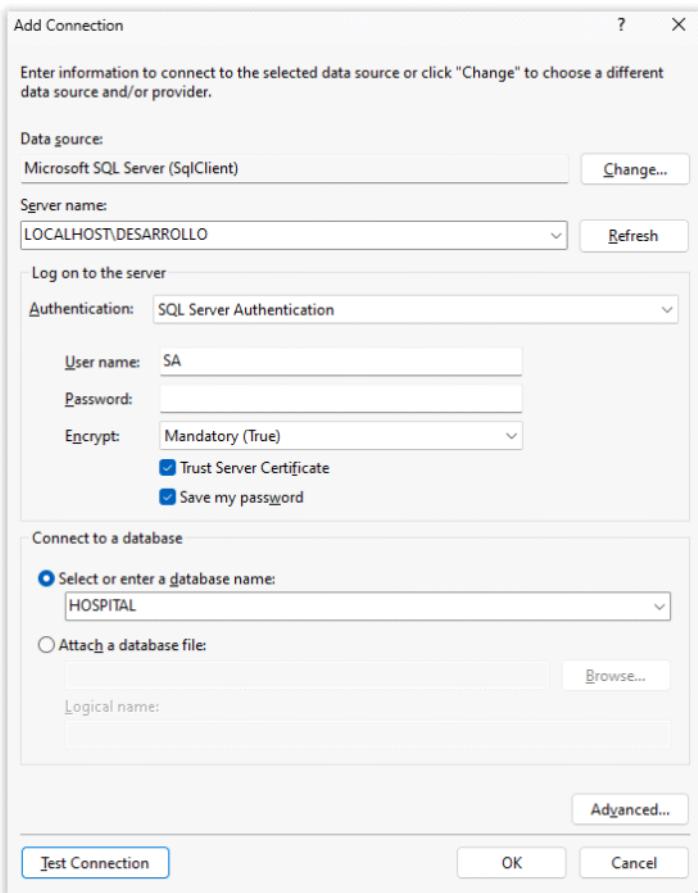
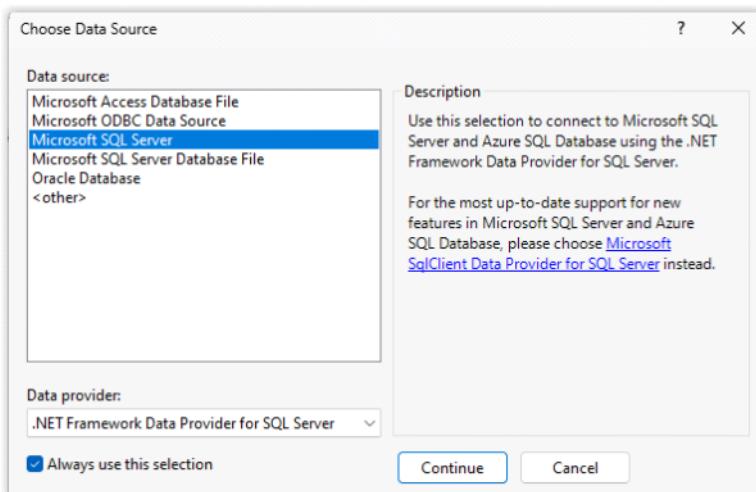
Comenzamos creando un nuevo proyecto Forms llamado **AdoNetCore**

Vamos a utilizar una herramienta de Visual Studio para tener siempre accesible la cadena de conexión.

Mediante el **Explorador de servidores** siempre tendremos ahí la cadena de conexión independientemente al proyecto sobre el que estemos trabajando

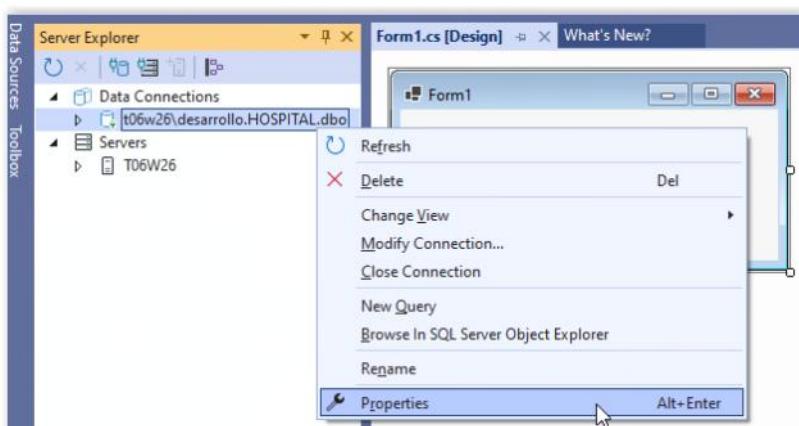
Abrimos **Server Explorer** mediante el menú **View**

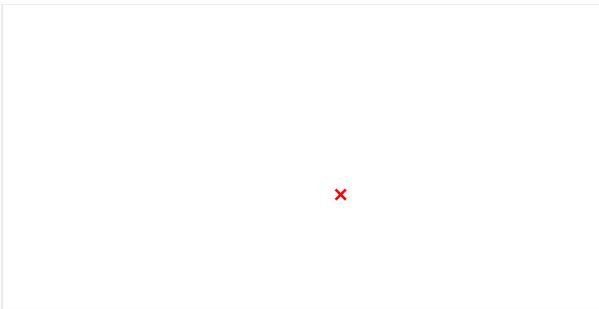
Sobre **Data Connections** agregamos una nueva conexión



En esta ventana podemos visualizar la cadena de conexión.

Botón derecho, Propiedades y ahí estará nuestroConnectionString





Para poder consumir nuestra base de datos de SQL Server necesitamos el Namespace **System.Data.SqlClient**

Cuando hacemos referencia a clases que no son nativas de Net Core se hace mediante **Nuget**

Vamos a instalar el Nuget de **Microsoft.Data.SqlClient**



Creamos un nuevo Form llamado **Form01PrimerAdo**



Si nos conectamos más de una vez cambiando la cadena de conexión nos dará una excepción



Una conexión siempre utilizará la misma cadena de conexión.

Si quiero conectar a otro servidor o base de datos, será otro objeto distinto.

La cadena de conexión se indica en el momento de instanciar el objeto **SqlConnection**

```
this.cn = new SqlConnection(this.connectionString);
```

El siguiente error es ofrecido porque la conexión ya está abierta

Debemos verificar el estado de la conexión antes de abrirla

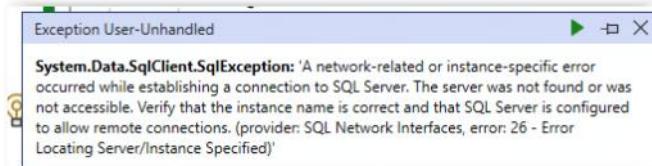


```

private void btnConectar_Click(object sender, EventArgs e)
{
    if (this.cn.State == ConnectionState.Closed)
    {
        this.cn.Open();
    }
    this.lblMensaje.BackColor = Color.LightGreen;
}

```

El siguiente error que podríamos tener es que el servidor NO esté disponible o esté caído.



La única forma de controlar este error es con Control de Excepciones

CONTROL DE EXCEPTIONS: try..catch

```

try {
    //CODIGO A EJECUTAR
}catch (Exception ex){
    //CODIGO SI SUCEDE ALGUNA EXCEPCION
}

try {
    string dato = "aaaa";
    int numero = int.Parse(dato);
}catch (Exception ex)
{
    this.lblMensaje.Text = ex.ToString();
}

```

Podemos capturar una excepción personalizada

```

try {
    string dato = "aaaa";
    int numero = int.Parse(dato);
    int num2 = 88;
    int division = num2 / 0;
}catch (DivideByZeroException ex)
{
    this.lblMensaje.Text = ex.ToString();
}

catch (Exception ex)
{
    this.lblMensaje.Text = ex.ToString();
}

```

```

try {
    if (this.cn.State == ConnectionState.Closed)
    {
        this.cn.Open();
    }
    this.lblMensaje.BackColor = Color.LightGreen;
}
catch (SqlException ex)
{
    this.lblMensaje.Text = ex.ToString();
}

```

En el aula nunca voy a utilizar try..catch porque siempre quiero que nos de excepción

Vamos a recuperar un evento de la conexión para cuando cambia el estado de la conexión.

```

    this.cn.StateChange += Cn_StateChange;
}

1 reference
private void Cn_StateChange(object sender, StateChangeEventArgs e)
{
    this.lblMensaje.Text = "La conexión está pasando de "
        + e.OriginalState + " a " + e.CurrentState;
}

```

Leer datos siempre será lo mismo.

- 1) Necesitamos una consulta.
- 2) Necesitamos la conexión
- 3) Indicar el tipo de consulta
- 4) La consulta en sí
- 5) Conectar

CODIGO FORMULARIO

```

using Microsoft.Data.SqlClient;

public partial class Form01PrimerAdo : Form
{
    SqlConnection cn;
    SqlCommand com;
    SqlDataReader reader;
    string connectionString;

    public Form01PrimerAdo()
    {
        InitializeComponent();
        this.connectionString = @"Data Source=LOCALHOST
\DESARROLLO;Initial Catalog=HOSPITAL;Persist Security Info=True;User ID=SA";
        this.cn = new SqlConnection(this.connectionString);
        this.com = new SqlCommand();
        this.cn.StateChange += Cn_StateChange;
    }

    private void Cn_StateChange(object sender, StateChangeEventArgs e)
    {
        this.lblMensaje.Text = "La conexión está pasando de "
            + e.OriginalState + " a " + e.CurrentState;
    }

    private void btnConectar_Click(object sender, EventArgs e)
    {
        try
        {
            if (this.cn.State == ConnectionState.Closed)
            {
                this.cn.Open();
            }
            this.lblMensaje.BackColor = Color.LightGreen;
        }
        catch (SqlException ex)
        {
            this.lblMensaje.Text = ex.ToString();
        }
    }

    private void btnDesconectar_Click(object sender, EventArgs e)

```

```

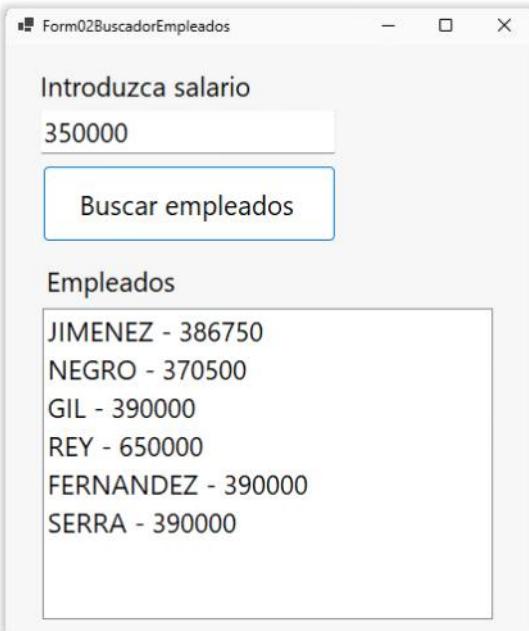
{
    this.cn.Close();
    this.lblMensaje.BackColor = Color.Red;
}

private void btnLeerDatos_Click(object sender, EventArgs e)
{
    //CONSULTA A REALIZAR
    string sql = "select * from EMP";
    //CONFIGURAMOS NUESTRO COMMAND
    //INDICAMOS LA CONEXION DEL COMMAND
    this.com.Connection = this.cn;
    //TIPO DE CONSULTA A REALIZAR
    this.com.CommandType = CommandType.Text;
    //LA PROPIA CONSULTA
    this.com.CommandText = sql;
    //AQUI DEBERIAMOS ABRIR LA CONEXION
    //ES UNA CONSULTA DE SELECCION POR LO QUE DEBEMOS
    //UTILIZAR EL METODO ExecuteReader() QUE NOS DEVUELVE
    //UN DataReader
    this.reader = this.com.ExecuteReader();
    //YA PODEMOS LEER DATOS
    //LEEMOS LA PRIMERA COLUMNA
    for (int i = 0; i < this.reader.FieldCount; i++)
    {
        string columna = this.reader.GetName(i);
        string tipoData = this.reader.GetDataTypeName(i);
        this.lstColumnas.Items.Add(columna);
        this.lstTiposData.Items.Add(tipoData);
    }
    //LEEMOS EL PRIMER REGISTRO
    //PARA LEER DATOS DEBEMOS UTILIZAR EL METODO Read()
    //DEL LECTOR
    while (this.reader.Read())
    {
        string apellido = this.reader["APELIDO"].ToString();
        this.lstApellidos.Items.Add(apellido);
    }
    //SIEMPRE DEBEMOS CERRAR TODO PARA PODER REUTILIZARLO
    this.reader.Close();
}
}

```

El siguiente ejemplo será un buscador de empleados por su Salario.
Haremos todo a la vez, conectar y leer datos.

Creamos un nuevo formulario llamado **Form02BuscadorEmpleados**



CODIGO FORMULARIO

```

public partial class Form02BuscadorEmpleados : Form
{
    SqlConnection cn;
    SqlCommand com;
    SqlDataReader reader;

    public Form02BuscadorEmpleados()
    {
        InitializeComponent();
        string connectionString = @"Data Source=LOCALHOST
\DESARROLLO;Initial Catalog=HOSPITAL;Persist Security Info=True;User ID=SA;Tru
st Server Certificate=True";
        this.cn = new SqlConnection(connectionString);
        this.com = new SqlCommand();
    }

    private void btnBuscarEmpleados_Click(object sender, EventArgs e)
    {
        string salario = this.txtSalario.Text;

```

```

string sql =
    "select * from EMP where salario >= " + salario;
//CONEXION
this.com.Connection = this.cn;
//TIPO DE CONSULTA
this.com.CommandType = CommandType.Text;
//LA CONSULTA
this.com.CommandText = sql;
//ABRIMOS LA CONEXION
//ENTRAR Y SALIR
this.cn.Open();
this.reader = this.com.ExecuteReader();
//DIBUJAMOS LOS DATOS
this.lstEmpleados.Items.Clear();
while (this.reader.Read())
{
    string ape = this.reader["APELIDO"].ToString();
    string sal = this.reader["SALARIO"].ToString();
    this.lstEmpleados.Items.Add(ape + " - " + sal);
}
//LIBERAMOS TODOS LOS RECURSOS UTILIZADOS
//SALIR
this.reader.Close();
this.cn.Close();
}
}

```

Vamos a realizar lo mismo, sobre el mismo formulario pero buscando por Oficio.

CODIGO BOTON BUSCAR

```

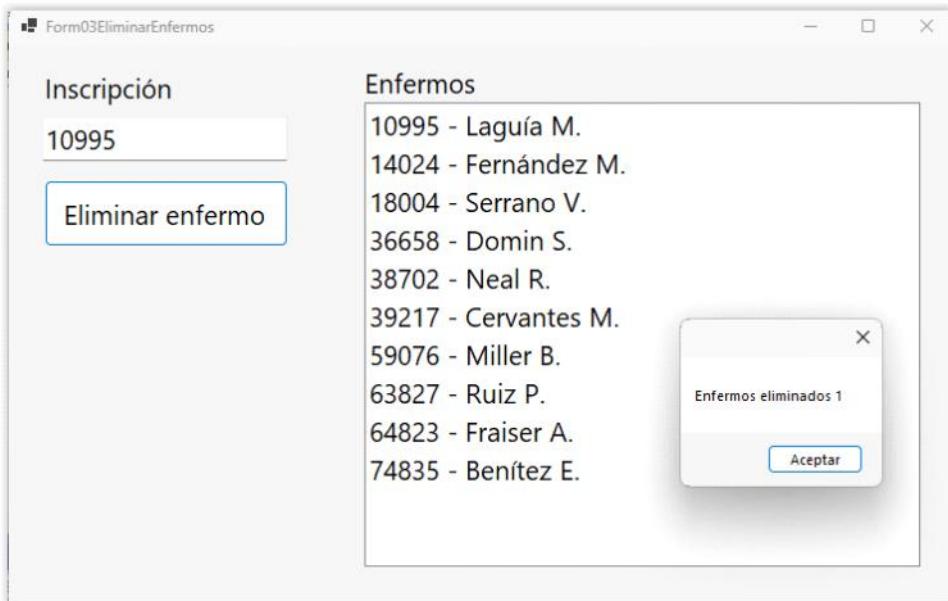
private void btnBuscarOficio_Click(object sender, EventArgs e)
{
    string sql =
        "select * from EMP where OFICIO="
        + this.txtOficio.Text + "'";
    this.com.Connection = this.cn;
    this.com.CommandType = CommandType.Text;
    this.com.CommandText = sql;
    this.cn.Open();
    this.reader = this.com.ExecuteReader();
    this.lstEmpleados.Items.Clear();
    while (this.reader.Read())
    {
        string ape = this.reader["APELIDO"].ToString();
        string ofi = this.reader["OFICIO"].ToString();
        this.lstEmpleados.Items.Add(ape + " - " + ofi);
    }
    this.reader.Close();
    this.cn.Close();
}

```

A continuación, vamos a realizar un ejemplo para consultas de acción.
Las consultas de acción se ejecutan mediante el método `ExecuteNonQuery()`
Que nos devuelve el número de registros que han sido afectados por la consulta

Vamos a realizar un ejemplo para eliminar Enfermos por su inscripción.

Creamos un nuevo formulario llamado **Form03EliminarEnfermos**



CODIGO FORMULARIO

```

public partial class Form03EliminarEnfermos : Form
{
    SqlConnection cn;
    SqlCommand com;
    SqlDataReader reader;

    public Form03EliminarEnfermos()
    {

```

```

{
    InitializeComponent();
    string connectionString = @"Data Source=LOCALHOST
\DESARROLLO;Initial Catalog=HOSPITAL;Persist Security Info=True;User ID=SA;Tru
st Server Certificate=True";
    this.cn = new SqlConnection(connectionString);
    this.com = new SqlCommand();
    this.CargarEnfermos();
}

private void CargarEnfermos()
{
    string sql = "select * from ENFERMO";
    this.com.Connection = this.cn;
    this.com.CommandType = CommandType.Text;
    this.com.CommandText = sql;
    this.cn.Open();
    this.reader = this.com.ExecuteReader();
    this.lstEnfermos.Items.Clear();
    while (this.reader.Read())
    {
        string inscripcion = this.reader["INSCRIPCION"].ToString();
        string apellido = this.reader["APELIDO"].ToString();
        this.lstEnfermos.Items.Add(inscripcion + " - " + apellido);
    }
    this.reader.Close();
    this.cn.Close();
}

private void btnEliminar_Click(object sender, EventArgs e)
{
    string sql =
        "delete from ENFERMO where INSCRIPCION="
        + this.txtInscripcion.Text;
    this.com.Connection = this.cn;
    this.com.CommandType = CommandType.Text;
    this.com.CommandText = sql;
    this.cn.Open();
    int eliminados = this.com.ExecuteNonQuery();
    this.cn.Close();
    this.CargarEnfermos();
    MessageBox.Show("Enfermos eliminados " + eliminados);
}
}

```

Sin copiar ni pegar, necesito un formulario que elimine PLANTILLA por el ID de Empleado.

Creamos un nuevo formulario llamado **Form04EliminarPlantilla**

Nota: Nunca debemos utilizar parámetros concatenados en las consultas SQL.
Esto puede llevar a un error de seguridad grave llamado **SQL Injection**

Para solucionar este problema, se utilizan **Parámetros** en las consultas.

En las consultas de tipo **Text** que son las que estamos utilizando es algo opcional
Utilizar parámetros.

En las consultas de tipo **Stored Procedures** es algo obligatorio.

CLASE PARAMETER

La clase Parameter sustituye los parámetros de una consulta por su valor.
La gran ventaja es evitar SQL Injection, pero otra gran ventaja es no estar utilizando Comillas simples ni nada, ya que el propio parámetro utiliza las comillas simples o no Dependiendo de su tipo de dato.

Podemos tener todos los parámetros que deseemos dentro de una misma consulta, pero No podemos repetir el **NAME** de los parámetros para una consulta, debido a que Los parámetros son buscados mediante su **name** dentro de la consulta y si tenemos dos **Name** no sabe cual debe sustituir.

Los nombres de los parámetros también cambiarán dependiendo del origen de datos, es decir, no será lo mismo el nombre de los parámetros en Oracle que en SQL Server.

Los parámetros entran en el juego de **ENTRAR/SALIR**, es decir, siempre los creamos y los Eliminamos en cada consulta.

Métodos y propiedades:

- **ParameterName:** El nombre del parámetro en la consulta
- **Value:** El valor del parámetro para ser sustituido
- **DbType:** Son opcionales, no son necesarios si "reconoce" el tipo, es decir, si Es un string, pondrá comillas simples aunque yo no se lo digo.
El tipado de Net Core
- **SqlDbType:** El tipado de Sql.
- **Direction:** Dirección del parámetro. Por defecto, todos los parámetros son de entrada , pero podrían ser de salida o de los dos.

Utilización de los parámetros

Se incluyen los parámetros en la consulta string.

No diferencia mayúsculas de minúsculas.

SQL SERVER

string sql = "select * from EMP where OFICIO=@oficio";

ORACLE

string sql = "select * from EMP where OFICIO=:oficio";

```
ParameterName = @oficio  
DbType = string  
SqlDbType = nvarchar  
Value=ANALISTA
```

Automáticamente, en la consultas e internamente se sustituye:

```
select * from EMP where OFICIO=@ANALISTA
```

Los parámetros están incluidos dentro de una colección en el objeto **Command** y mediante la propiedad **Parameters** se van incluyendo.

Vamos a modificar el código que tenemos de Eliminar enfermos para incluir un parámetro Con la inscripción.

BOTON ELIMINAR

```
//DEBEMOS INDICAR EL TIPO DE DATOS A ENVIAR COMO PARAMETRO  
//INSCRIPCION ES UN NUMERO, POR LO QUE DEBEMOS HACER UN  
//CASTING PARA int  
int inscripcion = int.Parse(this.txtInscripcion.Text);  
string sql =  
    "delete from ENFERMO where INSCRIPCION=@inscripcion";  
//CREAMOS EL PARAMETRO PARA LA INSCRIPCION  
SqlParameter pamInscripcion = new SqlParameter();  
pamInscripcion.ParameterName = "@inscripcion";  
//Value DEBE SER DEL MISMO TIPO QUE EL PARAMETRO (int)  
pamInscripcion.Value = inscripcion;  
pamInscripcion.DbType = DbType.Int32;  
//Direction INDICA SI EL PARAMETRO ES ENTRADA O SALIDA  
//POR DEFECTO, ES Input.  
pamInscripcion.Direction = ParameterDirection.Input;  
//AÑADIMOS EL PARAMETRO AL COMMAND  
this.com.Parameters.Add(pamInscripcion);
```

Actualmente, si pulsamos dos veces para Eliminar...



Siempre debemos eliminar los parámetros al finalizar las acciones. (Entrar/Salir)

```
this.cn.Open();  
int eliminados = this.com.ExecuteNonQuery();  
this.cn.Close();  
this.com.Parameters.Clear();
```

Por último, podemos indicar en el constructor de **Parameter** el **name** y el **value** directamente

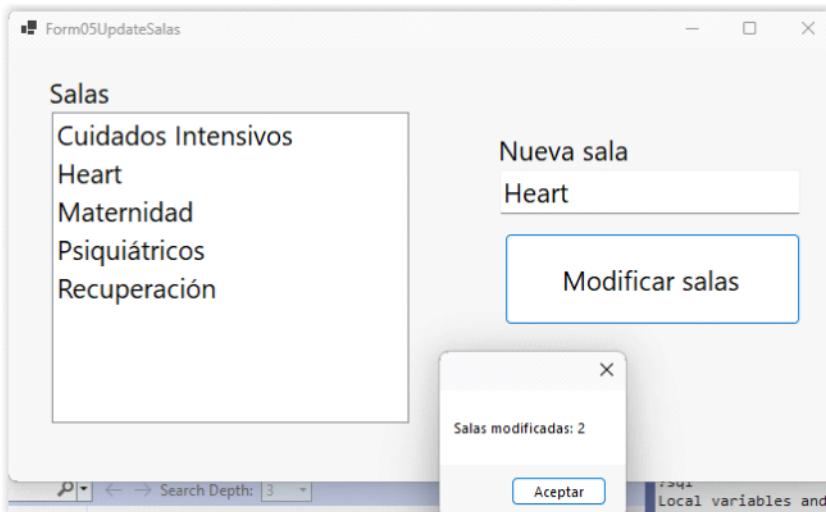
```
SqlParameter pamInscripcion =  
    new SqlParameter("@inscripcion", inscripcion);  
this.com.Parameters.Add(pamInscripcion);
```

Vamos a crear un nuevo formulario llamado **Form05UpdateSalas**

Al iniciar la aplicación, cargamos todas las salas.

Modificaremos la sala seleccionada con un nuevo nombre que escribamos en la caja

Volvemos a cargar las salas al realizar el cambio.



CODIGO FORMULARIO

```

public partial class Form05UpdateSalas : Form
{
    SqlConnection cn;
    SqlCommand com;
    SqlDataReader reader;

    public Form05UpdateSalas()
    {
        InitializeComponent();
        string connectionString = @"Data Source=LOCALHOST
\DESARROLLO;Initial Catalog=HOSPITAL;Persist Security Info=True;User ID=SA;Tru
st Server Certificate=True";
        this.cn = new SqlConnection(connectionString);
        this.com = new SqlCommand();
        this.LoadSalas();
    }

    private async void LoadSalas()
    {
        string sql = "select distinct NOMBRE from SALA";
        this.com.Connection = this.cn;
        this.com.CommandType = CommandType.Text;
        this.com.CommandText = sql;
        await this.cn.OpenAsync();
        this.reader = await this.com.ExecuteReaderAsync();
        this.lstSalas.Items.Clear();
        while (await this.reader.ReadAsync())
        {
            string nombre = this.reader["NOMBRE"].ToString();
            this.lstSalas.Items.Add(nombre);
        }
        await this.reader.CloseAsync();
        await this.cn.CloseAsync();
    }

    private async void btnModificarSalas_Click(object sender, EventArgs e)
    {
        string sql =
            "update SALA set NOMBRE=@nuevonombre where NOMBRE=@antiguonombre";
        string newName = this.txtNuevaSala.Text;
        string oldName = this.lstSalas.SelectedItem.ToString();
        SqlParameter pamOld =
            new SqlParameter("@antiguonombre", oldName);
        this.com.Parameters.Add(pamOld);
        SqlParameter pamNew =
            new SqlParameter("@nuevonombre", newName);
        this.com.Parameters.Add(pamNew);
        this.com.Connection = this.cn;
        this.com.CommandType = CommandType.Text;
        this.com.CommandText = sql;
        await this.cn.OpenAsync();
        int afectados =
            await this.com.ExecuteNonQueryAsync();
        await this.cn.CloseAsync();
        this.com.Parameters.Clear();
        this.LoadSalas();
        MessageBox.Show("Salas modificadas: " + afectados);
    }
}

```

CLASES DE ACCESO A DATOS

Esto es exactamente lo mismo que hicimos ayer.
Pero debemos abstraernos de la parte gráfica para poder trabajar con acceso a datos.
Es lo mismo que hicimos con los ficheros, llevarnos toda la parte lógica a clases.

Debemos tener la siguiente estructura de elementos en cualquier proyecto:

- **Models:** Son las clases que representan datos. Algunas clases tendremos que representarlas y otras no, por ejemplo, si vamos a dibujar apellidos y solamente esto, no creamos un Model, ya tenemos un List<string>

- **Repositories:** Son las clases que contienen la ejecución de los datos. Lo que harán Será devolver a las partes gráficas los datos o modificar datos.

Por no estar mareando, vamos a realizar las clases dentro del mismo proyecto.

Vamos a realizar el ejemplo de Modificar salas, pero con acceso a clases.

La clase Repository es la encargada de ejecutar las consultas, por lo que, la clase Form Ni siquiera tendrá nada de **Microsoft.Data.SqlClient**

¿Qué necesitamos?

- En este ejemplo, simplemente un Repository

- `GetNombreSalas() -> List<string>`
- `UpdateNombreSala(string oldname, string newname)`

Sobre el proyecto, creamos una nueva carpeta llamada **Repositories** y una clase llamada **RepositorySalas**

REPOSITORYSALAS

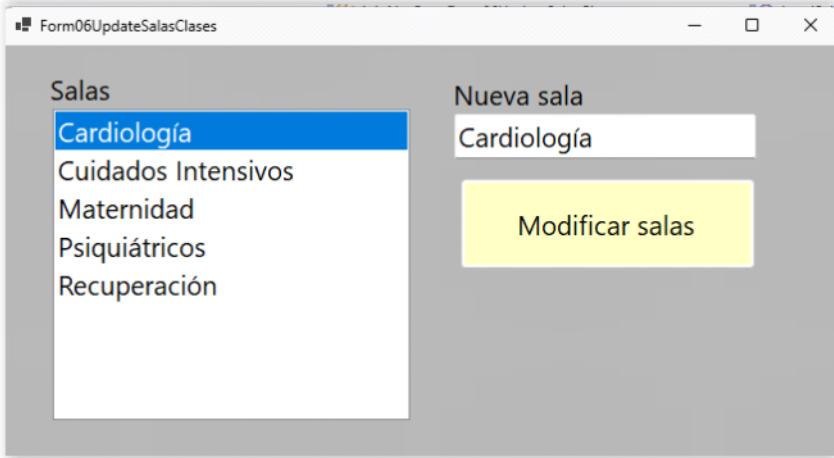
```
public class RepositorySalas
{
    private SqlConnection cn;
    private SqlCommand com;
    private SqlDataReader reader;

    public RepositorySalas()
    {
        string connectionString = @"Data Source=LOCALHOST
\DESARROLLO;Initial Catalog=HOSPITAL;Persist Security Info=True;User ID=SA;Tru
st Server Certificate=True";
        this.cn = new SqlConnection(connectionString);
        this.com = new SqlCommand();
    }

    public async Task<List<string>> GetNombresSalaAsync()
    {
        string sql = "select distinct NOMBRE from SALA";
        this.com.Connection = this.cn;
        this.com.CommandType = CommandType.Text;
        this.com.CommandText = sql;
        await this.cn.OpenAsync();
        this.reader = await this.com.ExecuteReaderAsync();
        //NECESITAMOS CREAR EL OBJETO QUE VAYAMOS A DEVOLVER
        List<string> salas = new List<string>();
        while (await this.reader.ReadAsync())
        {
            string nombre = this.reader["NOMBRE"].ToString();
            salas.Add(nombre);
        }
        await this.reader.CloseAsync();
        await this.cn.CloseAsync();
        return salas;
    }

    public async Task UpdateNombreSalaAsync
        (string oldName, string newName)
    {
        string sql = "update SALA set NOMBRE=@nuevonombre "
            + " where NOMBRE=@antiguonombre";
        SqlParameter pamNewName =
            new SqlParameter("@nuevonombre", newName);
        this.com.Parameters.Add(pamNewName);
        SqlParameter pamOldName =
            new SqlParameter("@antiguonombre", oldName);
        this.com.Parameters.Add(pamOldName);
        this.com.Connection = this.cn;
        this.com.CommandType = CommandType.Text;
        this.com.CommandText = sql;
        await this.cn.OpenAsync();
        await this.com.ExecuteNonQueryAsync();
        await this.cn.CloseAsync();
        this.com.Parameters.Clear();
    }
}
```

Creamos un nuevo formulario llamado **Form06UpdateSalasClases** con el mismo diseño Que el Form05



CODIGO FORMULARIO

```
public partial class Form06UpdateSalasClases : Form
{
    RepositorySalas repo;

    public Form06UpdateSalasClases()
    {
        InitializeComponent();
        this.repo = new RepositorySalas();
        this.LoadSalas();
    }

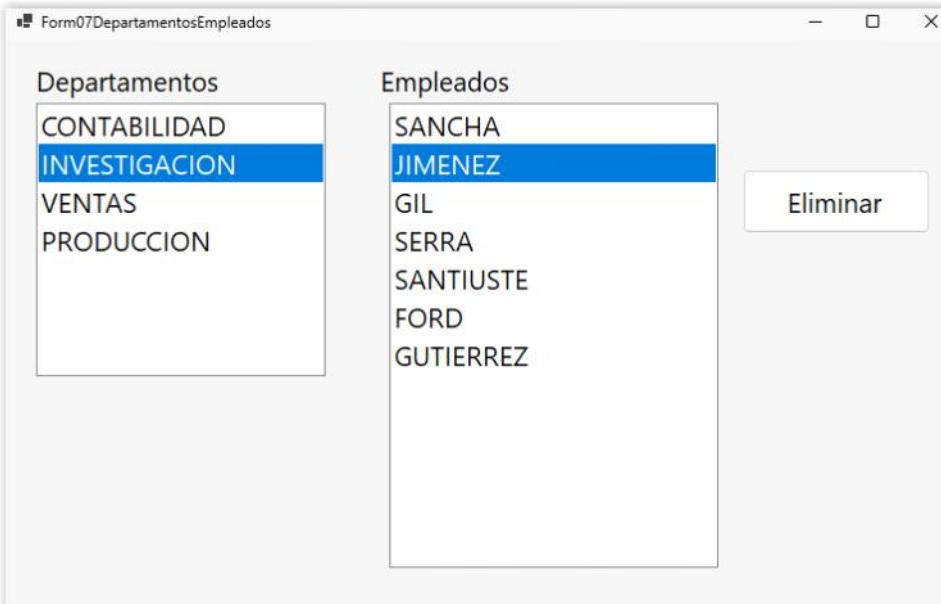
    private async void LoadSalas()
    {
        List<string> salas =
            await this.repo.GetNombresSalaAsync();
        this.lstSalas.Items.Clear();
        foreach (string nombre in salas)
        {
            this.lstSalas.Items.Add(nombre);
        }
    }

    private async void btnModificarSalas_Click(object sender, EventArgs e)
    {
        string oldName = this.lstSalas.SelectedItem.ToString();
        string newName = this.txtNuevaSala.Text;
        await this.repo.UpdateNombreSalaAsync(oldName, newName);
        this.LoadSalas();
    }
}
```

Creamos un nuevo formulario llamado **Form07DepartamentosEmpleados**

Al cargar el formulario, mostraremos los NOMBRES de departamento en la lista
Al seleccionar un nombre de departamento, mostraremos los apellidos de los
Empleados
Podremos seleccionar el apellido de un empleado y eliminarlo.

Creamos un nuevo Repo llamado **RepositoryDepartamentosEmpleados**



REPOSITORYDEPARTAMENTOSEMPLEADOS

```

public class RepositoryDepartamentosEmpleados
{
    private SqlConnection cn;
    private SqlCommand com;
    private SqlDataReader reader;

    public RepositoryDepartamentosEmpleados()
    {
        string connectionString = @"Data Source=LOCALHOST
\DESARROLLO;Initial Catalog=HOSPITAL;Persist Security Info=True;User ID=SA;Tru
st Server Certificate=True";
        this.cn = new SqlConnection(connectionString);
        this.com = new SqlCommand();
    }

    public async Task<List<string>> GetNombresDepartamentoAsync()
    {
        string sql = "select DNOMBRE from DEPT";
        this.com.Connection = this.cn;
        this.com.CommandType = CommandType.Text;
        this.com.CommandText = sql;
        await this.cn.OpenAsync();
        this.reader = await this.com.ExecuteReaderAsync();
        List<string> nombres = new List<string>();
        while (await this.reader.ReadAsync())
        {
            string nombre = this.reader["DNOMBRE"].ToString();
            nombres.Add(nombre);
        }
        await this.reader.CloseAsync();
        await this.cn.CloseAsync();
        return nombres;
    }

    public async Task<List<string>> GetEmpleadosDepartamentoAsync(
        string nombreDepartamento)
    {
        string sql = "select EMP.APELLIDO from EMP "
            + " inner join DEPT "
            + " on EMP.DEPT_NO = DEPT.DEPT_NO "
            + " where DEPT.DNOMBRE=@departamento";
        SqlParameter pamDept =
            new SqlParameter("@departamento", nombreDepartamento);
        this.com.Parameters.Add(pamDept);
        this.com.Connection = this.cn;
        this.com.CommandType = CommandType.Text;
        this.com.CommandText = sql;
        await this.cn.OpenAsync();
        this.reader = await this.com.ExecuteReaderAsync();
        List<string> empleados = new List<string>();
        while (await this.reader.ReadAsync())
        {
            string apellido = this.reader["APELLIDO"].ToString();
            empleados.Add(apellido);
        }
        await this.reader.CloseAsync();
        await this.cn.CloseAsync();
        this.com.Parameters.Clear();
        return empleados;
    }

    public async Task DeleteEmpleadoAsync(string apellido)
    {
        string sql = "delete from EMP where APELLIDO=@apellido";
        SqlParameter pamApellido =
            new SqlParameter("@apellido", apellido);
        this.com.Parameters.Add(pamApellido);
        this.com.Connection = this.cn;
        this.com.CommandType = CommandType.Text;
        this.com.CommandText = sql;
        await this.cn.OpenAsync();
        await this.com.ExecuteNonQueryAsync();
        await this.cn.CloseAsync();
        this.com.Parameters.Clear();
    }
}

```

CODIGO FORMULARIO

```

public partial class Form07DepartamentosEmpleados : Form
{
    RepositoryDepartamentosEmpleados repo;

    public Form07DepartamentosEmpleados()
    {
        InitializeComponent();
        this.repo = new RepositoryDepartamentosEmpleados();
        this.LoadDepartamentos();
    }

    private async void LoadDepartamentos()
    {
        List<string> departamentos =
            await this.repo.GetNombresDepartamentoAsync();
        this.lstDepartamentos.Items.Clear();
        foreach (string nombre in departamentos)
        {
            this.lstDepartamentos.Items.Add(nombre);
        }
    }

    public async void LoadEmpleados()
    {
        string nombreDepartamento =

```

```

        this.lstDepartamentos.SelectedItem.ToString();
    List<string> empleados =
        await this.repo
            .GetEmpleadosDepartamentoAsync(nombreDepartamento);
    this.lstEmpleados.Items.Clear();
    foreach (string apellido in empleados)
    {
        this.lstEmpleados.Items.Add(apellido);
    }
}

private void lstDepartamentos_SelectedIndexChanged(object sender, EventArgs e)
{
    this.LoadEmpleados();
}

private async void btnEliminar_Click(object sender, EventArgs e)
{
    string apellido =
        this.lstEmpleados.SelectedItem.ToString();
    await this.repo.DeleteEmpleadoAsync(apellido);
    this.LoadEmpleados();
}
}

```

El siguiente ejemplo será un **CRUD** con un modelo.

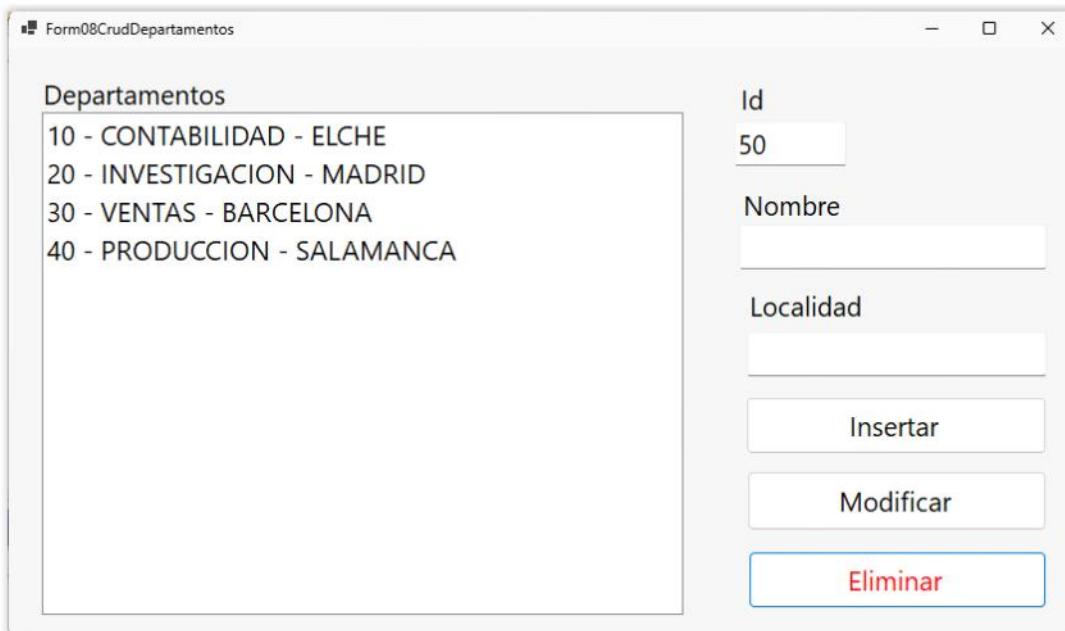
En este ejemplo, lo que queremos es dibujar todos los datos de los Departamentos

En una lista (listbox).

Al seleccionar un dato en la lista, quiero dibujar sus datos en cajas de texto.

Podremos modificar, insertar y eliminar departamentos.

Creamos un nuevo form llamado **Form08CrudDepartamentos**



En este ejemplo SI que necesitamos un Model para comunicar todos los datos
De cada Departamento con el dibujo.

Sobre el proyecto, creamos una nueva carpeta llamada **Models** y una clase llamada **Departamento**

```

public class Departamento
{
    0 references | 0 changes | 0 authors, 0 changes
    public int IdDepartamento { get; set; }
    0 references | 0 changes | 0 authors, 0 changes
    public string Nombre { get; set; }
    0 references | 0 changes | 0 authors, 0 changes
    public string Localidad { get; set; }
}

```

Sobre **Repositories** creamos una nueva clase llamada **RepositoryDepartamentos**

REPOSITORYDEPARTAMENTOS

```

public class RepositoryDepartamentos
{
    private SqlConnection cn;
    private SqlCommand com;
    private SqlDataReader reader;

```

```

public RepositoryDepartamentos()
{
    string connectionString = @"Data Source=LOCALHOST
\DESARROLLO;Initial Catalog=HOSPITAL;Persist Security Info=True;User ID=SA;Tru
st Server Certificate=True";
    this.cn = new SqlConnection(connectionString);
    this.com = new SqlCommand();
    this.com.Connection = this.cn;
}

//CREATE, READ, UPDATE, DELETE
//DEVOLVER TODOS LOS DEPARTAMENTOS
public async Task<List<Departamento>> GetDepartamentosAsync()
{
    string sql = "select * from DEPT";
    this.com.CommandType = CommandType.Text;
    this.com.CommandText = sql;
    await this.cn.OpenAsync();
    this.reader = await this.com.ExecuteReaderAsync();
    List<Departamento> departamentos = new List<Departamento>();
    while (await this.reader.ReadAsync())
    {
        int id = int.Parse(this.reader["DEPT_NO"].ToString());
        string nombre = this.reader["DNOMBRE"].ToString();
        string localidad = this.reader["LOC"].ToString();
        Departamento dept = new Departamento();
        dept.IdDepartamento = id;
        dept.Nombre = nombre;
        dept.Localidad = localidad;
        departamentos.Add(dept);
    }
    await this.reader.CloseAsync();
    await this.cn.CloseAsync();
    return departamentos;
}

public async Task InsertDepartamentoAsync
    (int id, string nombre, string localidad)
{
    string sql =
        "insert into DEPT values (@id, @nombre, @localidad)";
    SqlParameter pamId =
        new SqlParameter("@id", id);
    this.com.Parameters.Add(pamId);
    SqlParameter pamNombre =
        new SqlParameter("@nombre", nombre);
    this.com.Parameters.Add(pamNombre);
    SqlParameter pamLocalidad =
        new SqlParameter("@localidad", localidad);
    this.com.Parameters.Add(pamLocalidad);
    this.com.CommandType = CommandType.Text;
    this.com.CommandText = sql;
    await this.cn.OpenAsync();
    await this.com.ExecuteNonQueryAsync();
    await this.cn.CloseAsync();
    this.com.Parameters.Clear();
}

public async Task UpdateDepartamentoAsync
    (int id, string nombre, string localidad)
{
    string sql =
        "update DEPT set DNOMBRE=@nombre, LOC=@localidad "
        + " where DEPT_NO=@id";
    //TENEMOS UN METODO PARA ALMACENAR PARAMETROS
    //DIRECTAMENTE EN EL COMMAND SIN CREAR OBJETOS
    //ESTE METODO SOLAMENTE LO UTILIZAMOS CUANDO
    //LOS PARAMETROS SEAN TIPODOS PRIMITIVOS
    this.com.Parameters.AddWithValue("@id", id);
    this.com.Parameters.AddWithValue("@nombre", nombre);
    this.com.Parameters.AddWithValue("@localidad", localidad);
    this.com.CommandType = CommandType.Text;
    this.com.CommandText = sql;
    await this.cn.OpenAsync();
    await this.com.ExecuteNonQueryAsync();
    await this.cn.CloseAsync();
    this.com.Parameters.Clear();
}

public async Task DeleteDepartamentoAsync(int id)
{
    string sql = "delete from DEPT where DEPT_NO=@id";
    this.com.Parameters.AddWithValue("@id", id);
    this.com.CommandType = CommandType.Text;
    this.com.CommandText = sql;
    await this.cn.OpenAsync();
    await this.com.ExecuteNonQueryAsync();
    await this.cn.CloseAsync();
    this.com.Parameters.Clear();
}
}

```

CODIGO FORMULARIO

```

public partial class Form08CrudDepartamentos : Form
{
    private RepositoryDepartamentos repo;

    public Form08CrudDepartamentos()
    {
        InitializeComponent();
        this.repo = new RepositoryDepartamentos();
        this.LoadDepartamentos();
    }
}

```

```

private async void LoadDepartamentos()
{
    List<Departamento> departamentos =
        await this.repo.GetDepartamentosAsync();
    this.lstDepartamentos.Items.Clear();
    foreach (Departamento dept in departamentos)
    {
        this.lstDepartamentos.Items.Add(dept.IdDepartamento
            + " - " + dept.Nombre + " - " + dept.Localidad);
    }
}

private async void btnInsertar_Click(object sender, EventArgs e)
{
    int id = int.Parse(this.txtIdDepartamento.Text);
    string nombre = this.txtNombre.Text;
    string localidad = this.txtLocalidad.Text;
    await this.repo.InsertDepartamentoAsync(id, nombre, localidad);
    this.LoadDepartamentos();
}

private async void btnModificar_Click(object sender, EventArgs e)
{
    int id = int.Parse(this.txtIdDepartamento.Text);
    string nombre = this.txtNombre.Text;
    string localidad = this.txtLocalidad.Text;
    await this.repo.UpdateDepartamentoAsync(id, nombre, localidad);
    this.LoadDepartamentos();
}

private async void btnEliminar_Click(object sender, EventArgs e)
{
    int id = int.Parse(this.txtIdDepartamento.Text);
    await this.repo.DeleteDepartamentoAsync(id);
    this.LoadDepartamentos();
}

```

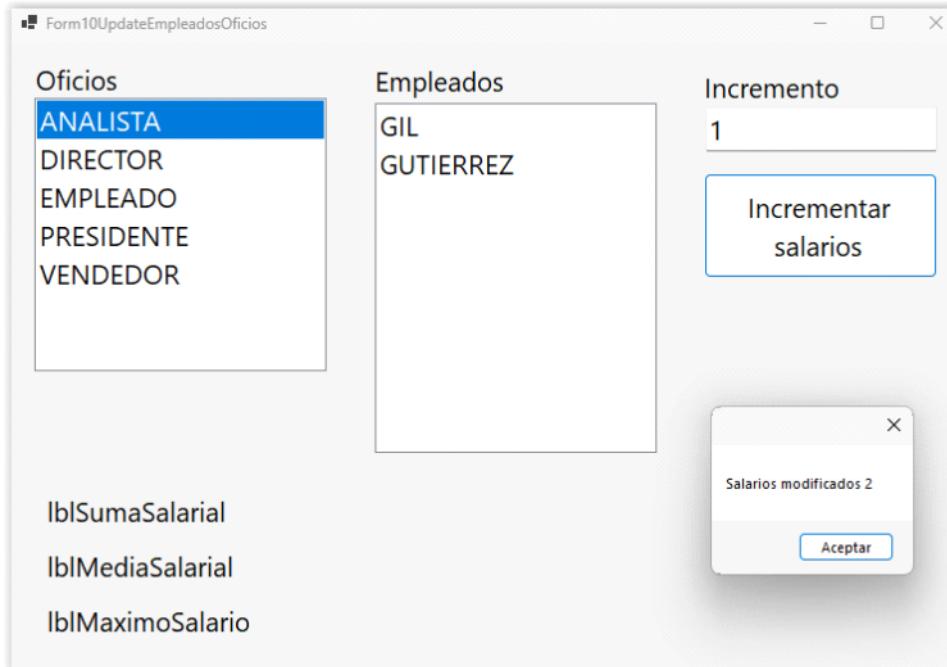
REALIZAMOS LA MISMA PRACTICA PERO CON HOSPITAL

Creamos un formulario nuevo llamado **Form09CrudHospitales**

Vamos a comenzar realizando una práctica en la que cargaremos los oficios
En una lista, al seleccionar un oficio, mostramos sus empleados y también
Mostraremos la media, el salario mínimo y el salario máximo de dicho oficio.

También podremos incrementar los salarios de los empleados por oficio.

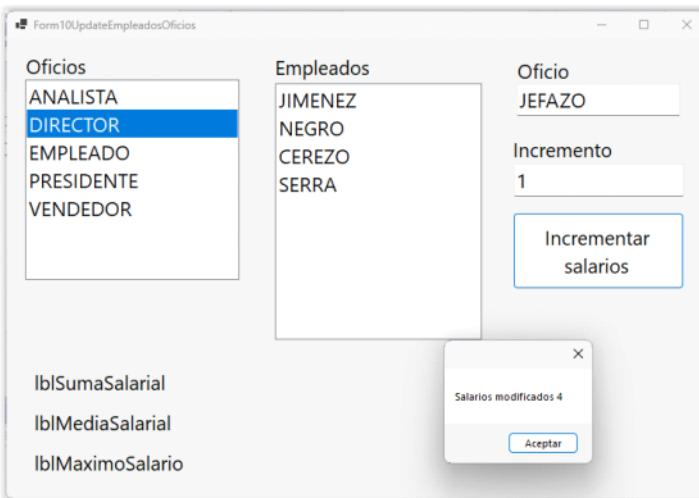
Creamos un nuevo formulario llamado **Form10UpdateEmpleadosOficios**



Creamos un nuevo Repositorio llamado **RepositoryUpdateEmpleados**

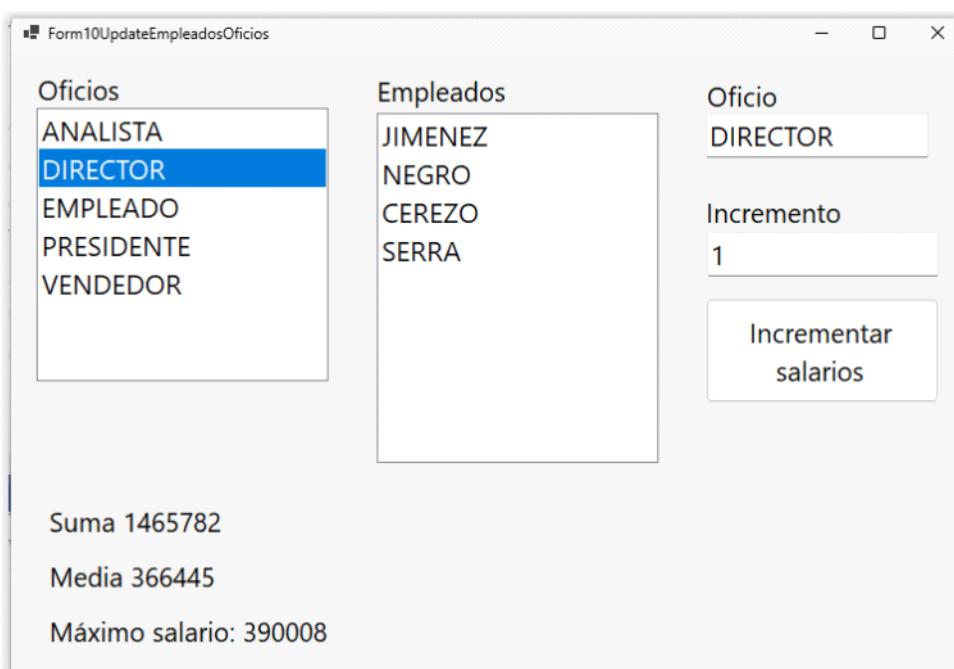
Versión 2

Modificamos también el Oficio de los empleados además de su salario
Debemos mostrar la media, el máximo salario y la suma salarial al seleccionar un Oficio.
Al modificar, debemos visualizar los cambios.



Debemos crear un modelo para devolver la suma, media y mínimo salario.

Sobre **Models** creamos una nueva clase llamada **DatosEmpleadosOficio**



REPOSITORYUPDATEEMPLEADOS

```
public class RepositoryUpdateEmpleados
{
    private SqlConnection cn;
    private SqlCommand com;
    private SqlDataReader reader;

    public RepositoryUpdateEmpleados()
    {
        string connectionString = @"Data Source=LOCALHOST
\DESARROLLO;Initial Catalog=HOSPITAL;Persist Security Info=True;User ID=SA;Trusted Server Certificate=True";
        this.cn = new SqlConnection(connectionString);
        this.com = new SqlCommand();
        this.com.Connection = this.cn;
    }

    public async Task<List<string>> GetOficiosAsync()
    {
        string sql = "select distinct OFICIO from EMP";
        this.com.CommandType = CommandType.Text;
        this.com.CommandText = sql;
        await this.cn.OpenAsync();
        this.reader = await this.com.ExecuteReaderAsync();
        List<string> oficios = new List<string>();
        while (await this.reader.ReadAsync())
        {
            string oficio = this.reader["OFICIO"].ToString();
            oficios.Add(oficio);
        }
        await this.reader.CloseAsync();
        await this.cn.CloseAsync();
        return oficios;
    }

    public async Task<List<string>> GetEmpleadosOficioAsync(string oficio)
```

```

{
    string sql = "select * from EMP where OFICIO=@oficio";
    this.com.Parameters.AddWithValue("@oficio", oficio);
    this.com.CommandType = CommandType.Text;
    this.com.CommandText = sql;
    await this.cn.OpenAsync();
    this.reader = await this.com.ExecuteReaderAsync();
    List<string> empleados = new List<string>();
    while (await this.reader.ReadAsync())
    {
        string apellido = this.reader["APELLIDO"].ToString();
        empleados.Add(apellido);
    }
    await this.reader.CloseAsync();
    await this.cn.CloseAsync();
    this.com.Parameters.Clear();
    return empleados;
}

public async Task<int> UpdateSalarioEmpleadosOficio
    (string oldOficio, string newOficio, int incremento)
{
    string sql = "update EMP set SALARIO=SALARIO+@incremento "
        + ", oficio=@newoficio "
        + " where OFICIO=@oldoficio";

    this.com.Parameters.AddWithValue("@incremento", incremento);
    this.com.Parameters.AddWithValue("@newoficio", newOficio);
    this.com.Parameters.AddWithValue("@oldoficio", oldOficio);
    this.com.CommandType = CommandType.Text;
    this.com.CommandText = sql;
    await this.cn.OpenAsync();
    int afectados = await this.com.ExecuteNonQueryAsync();
    await this.cn.CloseAsync();
    this.com.Parameters.Clear();
    return afectados;
}

public async Task<DatosEmpleadosOficio>
    GetDatosEmpleadosOficiosAsync(string oficio)
{
    string sql = "select sum(SALARIO) as SUMASALARIAL "
        + ", avg(SALARIO) as MEDIASALARIAL"
        + ", max(SALARIO) as MAXIMOSALARIO"
        + " from EMP where OFICIO=@oficio";
    this.com.Parameters.AddWithValue("@oficio", oficio);
    this.com.CommandType = CommandType.Text;
    this.com.CommandText = sql;
    await this.cn.OpenAsync();
    this.reader = await this.com.ExecuteReaderAsync();
    DatosEmpleadosOficio datos = new DatosEmpleadosOficio();
    await this.reader.ReadAsync();
    datos.SumaSalarial =
        int.Parse(this.reader["SUMASALARIAL"].ToString());
    datos.MediaSalarial =
        int.Parse(this.reader["MEDIASALARIAL"].ToString());
    datos.MaximoSalario =
        int.Parse(this.reader["MAXIMOSALARIO"].ToString());
    await this.reader.CloseAsync();
    await this.cn.CloseAsync();
    this.com.Parameters.Clear();
    return datos;
}
}

```

CODIGO DEL FORMULARIO

```

public partial class Form10UpdateEmpleadosOficios : Form
{
    RepositoryUpdateEmpleados repo;

    public Form10UpdateEmpleadosOficios()
    {
        InitializeComponent();
        this.repo = new RepositoryUpdateEmpleados();
        this.LoadOficios();
    }

    private async Task LoadOficios()
    {
        List<string> oficios = await this.repo.GetOficiosAsync();
        this.lstOficios.Items.Clear();
        foreach (string ofi in oficios)
        {
            this.lstOficios.Items.Add(ofi);
        }
    }

    private async void lstOficios_SelectedIndexChanged(object sender, EventArgs e)
    {
        if (this.lstOficios.SelectedIndex != -1)
        {
            string oficio = this.lstOficios.SelectedItem.ToString();
            await this.LoadEmpleados(oficio);

            DatosEmpleadosOficio datos =
                await this.repo.GetDatosEmpleadosOficiosAsync(oficio);
            this.lblSumaSalarial.Text = "Suma " + datos.SumaSalarial;
            this.lblMediaSalarial.Text = "Media " + datos.MediaSalarial;
            this.lblMaximoSalario.Text = "Máximo salario: "
                + datos.MaximoSalario;
        }
    }

    private async Task LoadEmpleados(string oficio)

```

```

    {
        List<string> empleados =
            await this.repo.GetEmpleadosOficioAsync(oficio);
        this.lstEmpleados.Items.Clear();
        foreach (string ape in empleados)
        {
            this.lstEmpleados.Items.Add(ape);
        }
    }

    private async void btnIncrementarSalarios_Click(object sender, EventArgs e)
    {
        int incremento = int.Parse(this.txtIncremento.Text);
        string oldOficio = this.lstOficios.SelectedItem.ToString();
        string newOficio = this.txtOficio.Text;
        int modificados =
            await this.repo.UpdateSalarioEmpleadosOficio
            (oldOficio, newOficio, incremento);
        MessageBox.Show("Salarios modificados " + modificados);
        await this.LoadOficios();
        DatosEmpleadosOficio datos =
            await this.repo.GetDatosEmpleadosOficiosAsync(newOficio);
        this.lblSumaSalarial.Text = "Suma " + datos.SumaSalarial;
        this.lblMediaSalarial.Text = "Media " + datos.MediaSalarial;
        this.lblMaximoSalario.Text = "Máximo salario: "
            + datos.MaximoSalario;
    }
}

```

VISTAS Y PROCEDIMIENTOS SQL SERVER

Esto **NO** es un **STANDARD**. Si nos acordamos, hace nada vimos Ansi SQL Server (92) donde todas las consultas que hicimos son Compatibles con cualquier base de datos.
Existen multitud de objetos en bases de datos como, por ejemplo, Triggers (disparadores al realizar acciones sobre las tablas), propias Tablas o tenemos consultas propias de base de datos, pero esos Objetos son únicos y propios de cada base de datos.

Las vistas y procedimientos, no importa la base de datos, debemos Aprenderlos porque es algo básico.

En cualquier desarrollo siempre se realizan peticiones sobre los Objetos de la base de datos y esos dos tres:

- TABLES
- VIEWS
- PROCEDURES

VIEWS

Una vista es una consulta SELECT dentro de un objeto de base de datos.

En realidad, en nuestro trabajo diario, jamás veréis una tabla, siempre vistas
Una vista es como una tabla.
Las vistas se crean por dos características

1) **Seguridad:** Una vista puede permitir/denegar acciones sobre ella.

Por ejemplo, podríamos impedir que "alguien" pueda borrar datos.
Imaginemos que estamos trabajando sobre la tabla EMP.
EMP sois vosotros, cada empleado y tenéis que trabajar con esa tabla
Deberíamos impedir visualizar el salario para no arder.
Lo que si puedo realizar es una Vista con una consulta sobre
La tabla EMP que no contenga el SALARIO y la COMISION.
Guardamos esa consulta en una VISTA, posteriormente, os quito
Permisos sobre la tabla EMP y os doy permisos sobre la vista.

2) **Simplificar consultas para filtros y mejorar rendimiento.**

Una vista contiene consultas SELECT. Dichas consultas se llaman Consultas Ad Hoc.
Una consulta Ad Hoc es una consulta que ya se ha ejecutado en un Servidor de base de datos, automáticamente, el servidor almacena La consulta y, la próxima vez que se ejecute, la devolverá más rápido.
Cuando tenemos una consulta enorme, ya sea con joins, group by y Demás elementos, si queremos ejecutarla, nos toca escribir un montón.
Si necesitamos crearla varias veces, con filtros y demás elementos,
Pues ya nos va costando más.

Algunas consideraciones:

- No deberíamos poner **where** a una vista. Pero si eliminamos datos De la tabla que afectan a dicho **where**, estamos creando algo inutil.
- Una vista no puede llevar **order by** en su instrucción

Para trabajar con objetos tenemos tres instrucciones:

- **CREATE:** Creación de un objeto
- **ALTER:** Modificar un objeto
- **DROP:** Eliminar un objeto

Sintaxis:

```

create view nombrevista
as
    select ... from TABLA
go

--EJEMPLO DE UNA VISTA DE EMPLEADOS SIN SALARIO Y COMISION
create view V_EMPLEADOS_SIMPLE
as
    select EMP_NO, APELLIDO, OFICIO
    , DIR, FECHA_ALT, DEPT_NO
    from EMP
go
--UNA VISTA ES UNA TABLA
--LA VISTA NUNCA TIENE DATOS, LOS DATOS ESTARAN
--SIEMPRE EN LA TABLA
select * from V_EMPLEADOS_SIMPLE order by EMP_NO
select * from EMP order by EMP_NO
--INSERTAMOS EN LA VISTA
insert into V_EMPLEADOS_SIMPLE values
(1112, 'VISTA', 'WORKER', 7839, GETDATE(), 10)
--MODIFICAMOS ALGO
update V_EMPLEADOS_SIMPLE set OFICIO='VIERNES'
where EMP_NO=1112
--DELETE
delete from V_EMPLEADOS_SIMPLE where EMP_NO=1112
--VAMOS A CREAR UNA VISTA PARA SIMPLIFICAR CONSULTAS
--TENDREMOS LOS DATOS DE EMPLEADOS Y LOS DATOS DE DEPARTAMENTOS
create view V_EMPLEADOS_DEPARTAMENTOS
as
    select EMP.EMP_NO, EMP.APELLIDO, EMP.OFICIO
    , EMP.SALARIO, DEPT.DNOMBRE as DEPARTAMENTO
    , DEPT.LOC as LOCALIDAD
    , DEPT.DEPT_NO
    from EMP
    inner join DEPT
    on EMP.DEPT_NO = DEPT.DEPT_NO
go
select * from V_EMPLEADOS_DEPARTAMENTOS
where LOCALIDAD='ELCHE'
--SUBIR EL SALARIO DE LOS EMPLEADOS DE ELCHE EN 1
update V_EMPLEADOS_DEPARTAMENTOS set SALARIO=SALARIO+1
where LOCALIDAD='ELCHE'
--MODIFICAR EL NOMBRE DE DEPARTAMENTO DE VENTAS POR I+D
update V_EMPLEADOS_DEPARTAMENTOS set DEPARTAMENTO='I+D'
where DEPARTAMENTO='VENTAS'
--VOLVEMOS A DEJAR EL DEPARTAMENTO DE I+D COMO VENTAS Y
--SUBIMOS EL SALARIO A SUS EMPLEADOS COMO REDENCION
--SI LA CONSULTAS AFECTA A VARIAS TABLAS, NO PUEDE REALIZAR
--LA MODIFICACION
update V_EMPLEADOS_DEPARTAMENTOS set DEPARTAMENTO='VENTAS'
, SALARIO=SALARIO+1
where DEPARTAMENTO='I+D'
--ELIMINAMOS LOS EMPLEADOS DE MADRID
--DE QUE TABLA VAMOS A ELIMINAR?
delete from V_EMPLEADOS_DEPARTAMENTOS where LOCALIDAD='MADRID'
delete from V_EMPLEADOS_DEPARTAMENTOS where EMP_NO=1111

```

PROCEDIMIENTOS ALMACENADOS

Un procedimiento es un código que se almacena dentro de la base de datos y
 Puede ser ejecutado cuando lo necesitemos.
 Podemos incluir cualquier tipo de consulta: INSERT, UPDATE, DELETE, SELECT y
 También puede incluir código TRANSACT SQL, que es el código lógico de SQL Server

Dicho código se ejecuta en el servidor de bases de datos y ofrece multitud de ventajas
 Un procedimiento puede recibir parámetros.
 Por ejemplo, dependiendo de los parámetros recibidos (IF) podríamos hacer una
 Acción u otra distinta dentro de nuestro código.

Imaginemos que tenemos que INSERTAR o MODIFICAR un dato.
 Si el dato existe, hacemos un **update** si el dato no existe, hacemos un **insert**
 Esto lo hacemos con Transact SQL.

Los parámetros de los procedimientos se nombran con @
 No existe case sensitive en el nombre de los parámetros: **@nombparametro**
 Los parámetros pueden ser de entrada o salida.

Sintaxis:

```

create procedure nombreProcedimiento
(@param1 tipo, @param2 tipo)
as
    --CODIGO LOGICO
go

```

```

1 --PROCEDIMIENTO PARA BUSCAR EMPLEADOS
2 --POR SU SALARIO
3 create procedure SP_BUSCADOR_EMP_SALARIOS
4 (@salario int)
5 as
6 select * from EMP
7 where SALARIO > @salario
8 go

```

Para ejecutar un procedimiento se utiliza la palabra EXEC o EXECUTE

```
exec SP_BUSCADOR_EMP_SALARIOS 250000
```

```

--PROCEDIMIENTO PARA INSERTAR UN DEPARTAMENTO
--EN LOS TIPOS DE DATO TEXTO (nvarchar)
--TENEMOS QUE INCLUIR SU LONGITUD
alter procedure SP_INSERTAR_DEPARTAMENTO
(@num int, @nombre nvarchar(30), @localidad nvarchar(30))
as
    insert into DEPT values (@num, @nombre, @localidad)
go
execute SP_INSERTAR_DEPARTAMENTO 72, 'STORED', 'PROCEDURE'
select * from DEPT

```

	DEPT_NO	DNOMBRE	LOC
5	71	S	P
6	72	STORED	PROCEDURE

LENGUAJE TRANSACT SQL

Este lenguaje es lógico y se utiliza en Functions, Stored Procedures o Triggers

En algunas bases de datos, utilizar un lenguaje lógico es opcional, pero en otras bases de datos es obligatorio.

Este tipo de lenguaje se suele utilizar únicamente para IF y comprobaciones. Por supuesto, tiene bucles y otras opciones del lenguaje, como funciones. Para lo que no se utiliza es para bucles lógicos, por ejemplo, la tabla de multiplicar

Los bucles que utiliza cualquier base de datos son bucles de CURSOR, como todos sabemos, Un CURSOR es una consulta SQL.

Las variables se declaran como un parámetro y con la palabra declare

```
declare @variable type
```

Tenemos dos formas de inicializar las variables

- 1) **SET:** Nos permite almacenar valores estáticos o que vengan de funciones
- 2) **SELECT:** Nos permite almacenar valores que vengan de consultas

```

--TRANSACT SQL
--ESTE CODIGO SE EJECUTA EN BLOQUE
declare @numero int
declare @fecha datetime
declare @mensaje nvarchar(50)
--ASIGNAMOS VALORES ESTATICOS
set @numero=15
set @mensaje='hoy es viernes'
--VALORES MEDIANTE FUNCIONES
set @fecha = getdate()
--REPRESENTAR LOS VALORES POR PANTALLA
--1) SELECT: UN SELECT REPRESENTA LAS VARIABLES
--Y PODRIAMOS RECUPERARLO EN CUALQUIER APP. (ADO NET)

```

```

--select @numero as NUMERO, @fecha as FECHA, @mensaje as MENSAJE
--2) PRINT: SE UTILIZA PARA MOSTRAR MENSAJES EN EL SERVIDOR DE BASE
--DE DATOS. SOLAMENTE SE PUEDEN RECUPERAR EN BASES DE DATOS. (mentira)
--EN CUALQUIER IF DE TRANSACT, DEBERIAMOS INCLUIRLOS COMO BUENA PRAXIS
--EN ESTE LENGUAJE DEBEMOS UTILIZAR CASTING PARA LAS CONVERSIONES
--cast(@variable as type)
print @mensaje + ', @numero: '
+ cast(@numero as nvarchar)
+ ', @fecha: '
+ cast(@fecha as nvarchar)
--ALMACENAR LOS VALORES DE LAS VARIABLES A PARTIR
--DE UN SELECT
--MOSTRAR LOS DATOS DEL EMPLEADO MAS ANTIGUO DE LA EMPRESA
select * from EMP where FECHA_ALT=
(select min(FECHA_ALT) from EMP)
--EN LUGAR DE UTILIZAR UNA SUBCONSULTA, PODEMOS
--UTILIZAR UNA VARIABLE Y ALMACENAR LA FECHA EN DICHA VARIABLE
declare @fechaALT datetime
select @fechaALT=min(FECHA_ALT) from EMP
print @fechaALT
select * from EMP where FECHA_ALT=@fechaALT
--LAS UTILIZACION DE SELECT ES O PARA ALMACENAR O PARA MOSTRAR
--NO SE PUEDEN COMBINAR:
--select @fechaALT=min(FECHA_ALT), APELLIDO from EMP
--CONDICIONALES DENTRO DE TRANSACT SQL
--PARA LAS CONDICIONES SE UTILIZAN LOS MISMOS OPERADORES QUE WHERE
--IN, OR, AND, =, <>
--if (condicion)
--begin
    --CONDICIONES TRUE
--end
--else if (otra condicion)
--begin
    --OTRA CONDICION
--end
--else
--begin
    --ELSE
--end
--VAMOS A COMPROBAR SI UN NUMERO ES POSITIVO/NEGATIVO O CERO
declare @num int
set @num=5
if (@num > 0)
begin
    print 'positivo'
end
else if (@num < 0)
begin
    print 'negativo'
end
else
begin
    print 'cero'
end
--MOSTRAR LOS EMPLEADOS DEL DEPARTAMENTO DE CONTABILIDAD
--SI NO EXISTE EL DEPARTAMENTO, MOSTRAMOS UN MENSAJE
--SI EXISTE, MOSTRAMOS LOS EMPLEADOS
declare @deptno int
select @deptno=DEPT_NO from DEPT
where DNOMBRE='CONTABILIDAD'
if (@deptno is null)
begin
    print 'No existe el departamento'
end
else
begin
    select * from EMP
    where DEPT_NO=@deptno
end
--MODIFICAR EL SALARIO DE UN EMPLEADO POR SU APELLIDO (SANCHAS)
--SI COBRA MENOS DE 150.000, SUBIMOS EL SALARIO EN 2.000
--SI COBRA MAS DE 150.000. BAJAMOS EL SALARIO EN 2.000
declare @sueldo int
select @sueldo=SALARIO from EMP
where APELLIDO='SANCHAS'
if (@sueldo > 150000)
begin
    update EMP set SALARIO=SALARIO-2000
    where APELLIDO='SANCHAS'
    print 'abajamos sueldo'
end
else
begin
    update EMP set SALARIO=SALARIO + 2000
    where APELLIDO='SANCHAS'
    print 'subimos sueldo'
end
--MODIFICAR EL SALARIO DE LOS EMPLEADOS DE LA PLANTILLA DE LA PAZ
--SI LA SUMA SALARIAL SUPERA 1.000.000, BAJAMOS EL SALARIO
--A TODOS LOS DE LA PAZ EN 10.000
--DE LO CONTRARIO, SUBIMOS EL SALARIO EN 10.000
declare @sumasalarial int
declare @hospitalcod int
select @hospitalcod = HOSPITAL_COD from HOSPITAL
where NOMBRE='LA PAZ'
select @sumasalarial=sum(SALARIO) from PLANTILLA
where HOSPITAL_COD=@hospitalcod
if (@sumasalarial >= 1000000)
begin
    update PLANTILLA set SALARIO=SALARIO-10000
    where HOSPITAL_COD=@hospitalcod
    print 'Bajando sueldos: ' + cast(@sumasalarial as nvarchar)
end
else
begin
    update PLANTILLA set SALARIO=SALARIO+10000
    where HOSPITAL_COD=@hospitalcod
    print 'Subiendo sueldos: ' + cast(@sumasalarial as nvarchar)

```

```

end
--SINTAXIS DE BUCLES
--LOS BUCLES NO SE LLEVAN BIEN CON LAS BASES DE DATOS
--NO SUELEN SER UNA HERRAMIENTA FUNCIONAL AUNQUE SE PUEDEN REALIZAR.
--SOLO SE UTILIZAN BUCLES CON CURSORES
declare @contador int
set @contador = 1
while (@contador <= 10)
begin
    print 'Contador ' + cast(@contador as nvarchar)
    set @contador = @contador + 1
end
--LOS BUCLES SE UTILIZAN EN CURSORES
--QUEREMOS ALMACENAR EL APELLIDO DE LOS EMPLEADOS
declare @ape nvarchar(50)
select @ape = APELLIDO from EMP
print @ape --ALMACENA UNO A UNO Y SE QUEDA EL ULTIMO
--SI QUEREMOS RECORRER CADA ELEMENTO DE FORMA INDIVIDUAL
--NECESITAMOS UNA VARIABLE LLAMADA CURSOR EN LA QUE SE ALMACENA
--UNA CONSULTA Y SE PUEDE RECORRER CON while
--VAMOS A RECORRER UNO A UNO CADA APELLIDO Y OFICIO DE LOS EMPLEADOS
declare @apellido nvarchar(50)
declare @oficio nvarchar(50)
--1) DECLARAR EL CURSOR COMO VARIABLE DE UNA CONSULTA
--LAS VARIABLES CURSOR NO LLEVAN @@
declare QUERY cursor for
select APELLIDO, OFICIO from EMP
--2) ABRIR CURSOR
open QUERY
--3) LEER LA PRIMERA FILA Y SE REALIZA CON fetch
fetch next from QUERY into @apellido, @oficio
--UNA VEZ QUE ESTAMOS EN LA PRIMERA FILA, EXISTE UNA
--VARIABLE LLAMADA @@fetch_status QUE INDICA SI HA TERMINADO DE
--LEER EL CURSOR O NO
--SI @@fetch_status = 0 EXISTEN REGISTROS
--4) BUCLE CON @@fetch_status RECORRIENDO EL CURSOR
while (@@FETCH_STATUS = 0)
begin
    print @apellido + ' - ' + @oficio
    --DEBEMOS INDICAR QUE VAYA MOVIENDOSE DE FILA EN FILA
    fetch next from QUERY into @apellido, @oficio
end
--5) CERRAR EL CURSOR
close QUERY
--6) LIBERAR LA MEMORIA DEL CURSOR
deallocate QUERY
--NECESITAMOS INCREMENTAR EL SALARIO DE LOS DOCTORES
--EN UN VALOR ALEATORIO HASTA 1000
--rand() * 1000
select rand() * 1000 as aleatorio
declare @incremento int
set @incremento = rand() * 1000
print @incremento
update DOCTOR set SALARIO=SALARIO+@incremento
--select * from DOCTOR
--NECESITO QUE CADA DOCTOR TENGA UN INCREMENTO INDIVIDUAL

```

Todo esto que hemos realizado se pierde en el tiempo.

Podemos almacenar dentro de un Procedimiento nuestros códigos realizados en Transact
Y ejecutarlos cuando deseemos.

Es más, con los parámetros, podremos hacer dinámico el código.

Por ejemplo, quiero realizar un procedimiento que modifique el Salario de un empleado
Por su apellido.

Si el empleado cobra más de 150.000 bajamos salario en 2.000, al contrario

Subimos en 2.000

```

create procedure SP_UPDATESALARIO_EMP_APELLIDO
(@apellido nvarchar(50)) as
    declare @sueldo int
    select @sueldo=SALARIO from EMP
    where APELLIDO=@apellido
    if (@sueldo > 150000)
        begin
            update EMP set SALARIO=SALARIO-2000
            where APELLIDO=@apellido
            print 'bajamos sueldo ' + cast(@sueldo as nvarchar)
        end
    else
        begin
            update EMP set SALARIO=SALARIO + 2000
            where APELLIDO=@apellido
            print 'subimos sueldo ' + cast(@sueldo as nvarchar)
        end
go

```

Podemos almacenar cualquier tipo de consulta, por ejemplo, lo que hemos realizado

Con los doctores y su incremento Random, podríamos llevarlo a un Procedimiento

Que actualizará respecto al valor que deseemos del random y lo hará sobre los
Doctores de una Especialidad.

```

exec SP_RANDOM_SALARIO_DOCTORES 'Cardiología', 50
create procedure SP_RANDOM_SALARIO_DOCTORES
(@especialidad nvarchar(50), @valorincremento int)
as
    declare @incremento int
    declare @iddoctor int
    declare @apellido nvarchar(50)
    declare @salario int
    declare CONSULTA cursor for
    select DOCTOR_NO, APELLIDO, SALARIO from DOCTOR
    where ESPECIALIDAD=@especialidad
    open CONSULTA
    fetch next from CONSULTA into @iddoctor, @apellido, @salario

```

```

while (@@FETCH_STATUS = 0)
begin
    set @incremento = rand() * @valorincremento
    set @salario = @salario + @incremento
    update DOCTOR set SALARIO=@salario
    where DOCTOR_NO=@iddoctor
    print 'Dr/Dra ' + @apellido + ' tiene un nuevo salario de '
    + cast(@salario as nvarchar) + '. Incremento de '
    + cast(@incremento as nvarchar)
    fetch next from CONSULTA into @iddoctor, @apellido, @salario
end
close CONSULTA
deallocate CONSULTA
go

exec SP_EXISTE_DEPARTAMENTO 'CONTABILIDAD'
create procedure SP_EXISTE_DEPARTAMENTO
(@nombre nvarchar(50)) as
declare @deptno int
select @deptno=DEPT_NO from DEPT
where DNOMBRE=@nombre
if (@deptno is null)
begin
    print 'No existe el departamento'
end
else
begin
    select * from EMP
    where DEPT_NO=@deptno
end
go

```

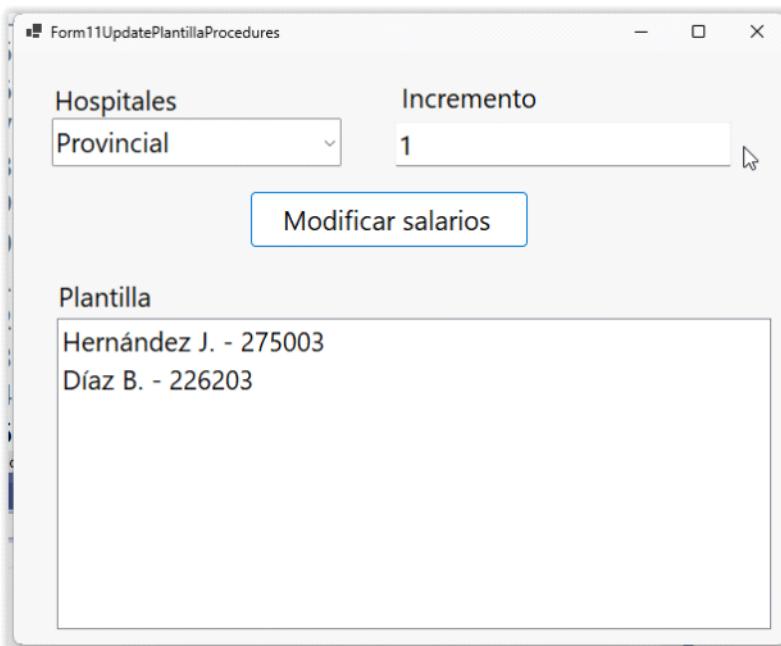
El siguiente paso está en visualizar cómo podemos llamar a procedimientos Almacenados desde Net Core.

Ahora mismo, no voy a utilizar repositorios, todo en el form para no estar saltando Entre pantallas

Vamos a realizar una aplicación que mostrará los datos de los nombres de Hospital

Tendremos un botón para modificar el salario de la plantilla.
Mostraremos también los datos de la plantilla en una lista.

Creamos un nuevo formulario llamado **Form11UpdatePlantillaProcedures**



PROCEDIMIENTOS ALMACENADOS

```

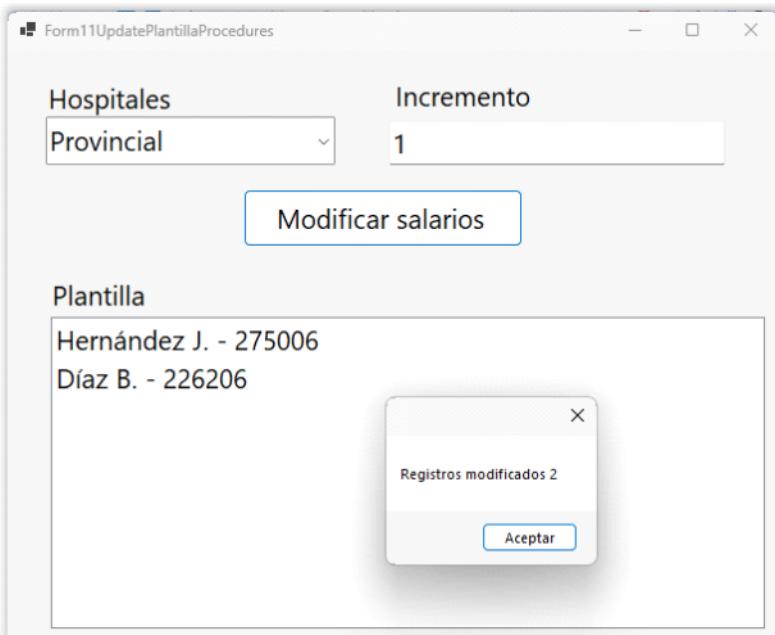
create procedure SP_ALL_HOSPITALES
as
    select * from HOSPITAL
go
create procedure SP_UPDATEPLANTILLA_HOSPITAL
(@nombre nvarchar(50), @incremento int)
as
    declare @hospitalcod int
    select @hospitalcod = HOSPITAL_COD from HOSPITAL
    where NOMBRE=@nombre
    update PLANTILLA set SALARIO=SALARIO + @incremento
    where HOSPITAL_COD=@hospitalcod
    select * from PLANTILLA
    where HOSPITAL_COD=@hospitalcod
go

```

CODIGO FORMULARIO

- Necesito un procedimiento para mostrar los empleados de la plantilla de un Hospital al seleccionar un hospital.
- Cuando modifiquemos los empleados de la plantilla quiero visualizar el número

De registros afectados en un mensaje y también ver los cambios en la lista.



PROCEDIMIENTOS ALMACENADOS

```
create procedure SP_ALL_HOSPITALES
as
    select * from HOSPITAL
go
create procedure SP_GETPLANTILLA_HOSPITAL
(@nombre nvarchar(50))
as
    declare @hospitalcod int
    select @hospitalcod = HOSPITAL_COD from HOSPITAL
    where NOMBRE=@nombre
    select * from PLANTILLA
    where HOSPITAL_COD=@hospitalcod
go
alter procedure SP_UPDATEPLANTILLA_HOSPITAL
(@nombre nvarchar(50), @incremento int)
as
    declare @hospitalcod int
    select @hospitalcod = HOSPITAL_COD from HOSPITAL
    where NOMBRE=@nombre
    update PLANTILLA set SALARIO=SALARIO + @incremento
    where HOSPITAL_COD=@hospitalcod
go
```

CODIGO FORMULARIO

```
public partial class Form11UpdatePlantillaProcedures : Form
{
    SqlConnection cn;
    SqlCommand com;
    SqlDataReader reader;

    public Form11UpdatePlantillaProcedures()
    {
        InitializeComponent();
        string connectionString = @"Data Source=LOCALHOST;Initial Catalog=HOSPITAL;Persist Security Info=True;User ID=SA;Trusted Connection=True";
        this.cn = new SqlConnection(connectionString);
        this.com = new SqlCommand();
        this.com.Connection = this.cn;
        this.LoadHospitales();
    }

    public async void LoadHospitales()
    {
        string sql = "SP_ALL_HOSPITALES";
        this.com.CommandType = CommandType.StoredProcedure;
        this.com.CommandText = sql;
        await this.cn.OpenAsync();
        this.reader = await this.com.ExecuteReaderAsync();
        this.cmbHospitales.Items.Clear();
        while (await this.reader.ReadAsync())
        {
            string nombre = this.reader["NOMBRE"].ToString();
            this.cmbHospitales.Items.Add(nombre);
        }
        await this.reader.CloseAsync();
        await this.cn.CloseAsync();
    }

    private async void btnModificarSalarios_Click(object sender, EventArgs e)
    {
        string nombre = this.cmbHospitales.SelectedItem.ToString();
        int incremento = int.Parse(this.txtIncremento.Text);
        string sql = "SP_UPDATEPLANTILLA_HOSPITAL";
        this.com.Parameters.AddWithValue("@nombre", nombre);
        this.com.Parameters.AddWithValue("@incremento", incremento);
```

```

        this.com.CommandType = CommandType.StoredProcedure;
        this.com.CommandText = sql;
        await this.cn.OpenAsync();
        int afectados = await this.com.ExecuteNonQueryAsync();
        await this.cn.CloseAsync();
        this.com.Parameters.Clear();
        await this.LoadPlantilla(nombre);
        MessageBox.Show("Registros modificados " + afectados);
    }

    public async Task LoadPlantilla(string nombre)
    {
        string sql = "SP_GETPLANTILLA_HOSPITAL";
        this.com.Parameters.AddWithValue("@nombre", nombre);
        this.com.CommandType = CommandType.StoredProcedure;
        this.com.CommandText = sql;
        await this.cn.OpenAsync();
        this.reader = await this.com.ExecuteReaderAsync();
        this.lstPlantilla.Items.Clear();
        while (await this.reader.ReadAsync())
        {
            string apellido = this.reader["APELIDO"].ToString();
            string salario = this.reader["SALARIO"].ToString();
            this.lstPlantilla.Items.Add(apellido + " - " + salario);
        }
        await this.reader.CloseAsync();
        await this.cn.CloseAsync();
        this.com.Parameters.Clear();
    }

    private async void cmbHospitales_SelectedIndexChanged(object sender, EventArgs e)
    {
        if (this.cmbHospitales.SelectedIndex != -1)
        {
            string nombre =
                this.cmbHospitales.SelectedItem.ToString();
            await this.LoadPlantilla(nombre);
        }
    }
}

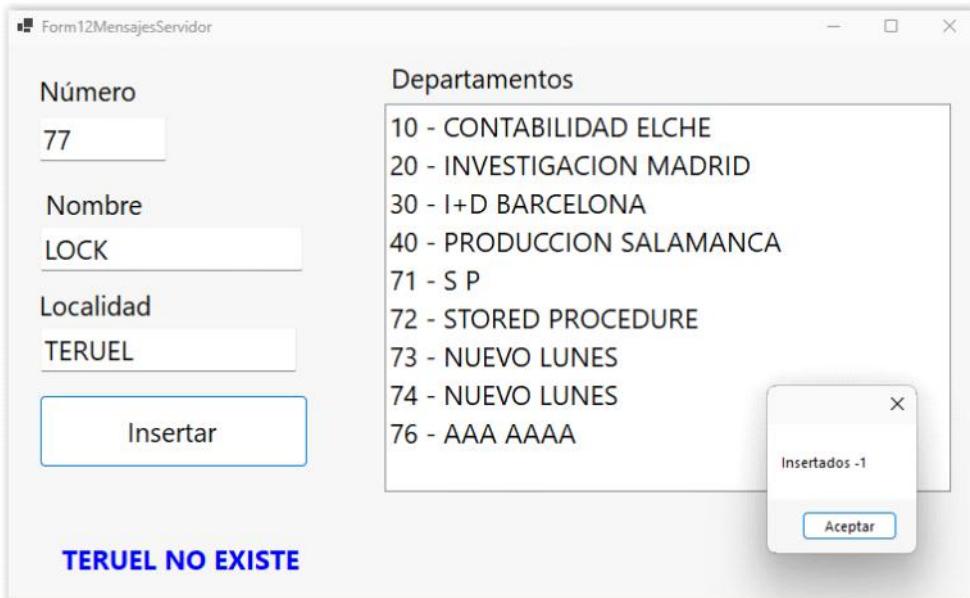
```

Creamos un nuevo formulario llamado **Form12MensajesServidor**

Por ahora, el Label de Mensaje como si no existiera
 Necesitamos una aplicación para insertar departamentos.
 Al iniciar, mostraremos los departamentos.
 Al insertar, mostramos un mensaje con el número de departamentos insertados
 Volvemos a cargar la lista.
 Todo el código en el formulario, nada de Repos.

Necesitamos dos procedimientos:

SP_ALL_DEPARTAMENTOS: Procedimiento para devolver todos los departamentos
SP_INSERT_DEPARTAMENTO(@numero, @nombre, @localidad): Procedimiento
 Para insertar un departamento



En el procedimiento de insertar departamentos vamos a incluir una lógica.
 Por ejemplo, no queremos departamentos en TERUEL, Teruel no existe.

```

alter procedure SP_INSERT_DEPARTAMENTO
(@numero int, @nombre nvarchar(50), @localidad nvarchar(50))
as
    --NO QUEREMOS LA LOCALIDAD DE TERUEL
    if (@localidad = 'TERUEL')
        begin
            print 'TERUEL NO EXISTE'
        end
    else
        begin
            insert into DEPT values (@numero, @nombre, @localidad)
        end
go

```

Todos sabemos que nos está bloqueando Teruel porque lo hemos escrito nosotros
 Pero si no fuera así, la única forma (si se ha seguido la pauta de buena praxis) es
 Recuperar el mensaje del servidor.

No importa la base de datos, podremos recuperar cualquier mensaje del servidor:

Print, dbms_output.

Nota: Esto es para nosotros como desarrolladores.

Para recuperar mensajes del servidor se utiliza un evento de la conexión llamado **InfoMessage**. Dentro de dicho evento se recuperan TODOS los mensajes que Ha lanzado la aplicación, no importa el número.

CODIGO FORMULARIO

```

public partial class Form12MensajesServidor : Form
{
    SqlConnection cn;
    SqlCommand com;
    SqlDataReader reader;

    public Form12MensajesServidor()
    {
        InitializeComponent();
        string connectionString = @"Data Source=LOCALHOST
\DESARROLLO;Initial Catalog=HOSPITAL;Persist Security Info=True;User ID=SA;Trusted Server Certificate=True";
        this.cn = new SqlConnection(connectionString);
        this.com = new SqlCommand();
        this.com.Connection = this.cn;

        //RECUPERAMOS EL EVENTO DE MENSAJES DE LA CONEXION
        this.cn.InfoMessage += Cn_InfoMessage;

        this.LoadDepartamentos();
    }

    private void Cn_InfoMessage(object sender, SqlInfoMessageEventArgs e)
    {
        this.lblMensaje.Text = e.Message;
    }

    public async Task LoadDepartamentos()
    {
        string sql = "SP_ALL_DEPARTAMENTOS";
        this.com.CommandType = CommandType.StoredProcedure;
        this.com.CommandText = sql;
        await this.cn.OpenAsync();
        this.reader = await this.com.ExecuteReaderAsync();
        this.lstDepartamentos.Items.Clear();
        while (await this.reader.ReadAsync())
        {
            string id = this.reader["DEPT_NO"].ToString();
            string nombre = this.reader["DNOMBRE"].ToString();
            string localidad = this.reader["LOC"].ToString();
            this.lstDepartamentos.Items.Add(id + " - "
                + nombre + " " + localidad);
        }
        await this.reader.CloseAsync();
        await this.cn.CloseAsync();
    }

    private void btnInsertar_Click(object sender, EventArgs e)
    {
        this.lblMensaje.Text = "";
        int id = int.Parse(this.txtNumero.Text);
        string nombre = this.txtNombre.Text;
        string localidad = this.txtLocalidad.Text;
        string sql = "SP_INSERT_DEPARTAMENTO";
        this.com.Parameters.AddWithValue("@numero", id);
        this.com.Parameters.AddWithValue("@nombre", nombre);
        this.com.Parameters.AddWithValue("@localidad", localidad);
    }
}

```

```

        this.com.CommandType = CommandType.StoredProcedure;
        this.com.CommandText = sql;
        this.cn.Open();
        int afectados = this.com.ExecuteNonQuery();
        this.cn.Close();
        this.com.Parameters.Clear();
        //await this.LoadDepartamentos();
        MessageBox.Show("Insertados " + afectados);
    }
}

```

PARAMETROS DE SALIDA PROCEDIMIENTOS ALMACENADOS

Dentro de cualquier procedimiento, podemos tener parámetros de entrada
 Para la funcionalidad del procedimiento y también parámetros de salida que pueda
 Recibir la llamada.

Depende de la base de datos, existen parámetros de entrada (defecto), parámetros
 De salida y parámetros de entrada/salida a la vez.

La funcionalidad está cuando necesitamos elementos "extra" de nuestras consultas
 Y los podemos recuperar desde la aplicación que ha realizado la llamada.

En algunas bases de datos, por ejemplo, al realizar consultas de acción sobre un objeto
 Procedimiento, NO devuelve el número de registros afectados. No lo podríamos saber.

También nos podría servir para devolver un dato extra que podría ser necesario y
 Así no necesitamos hacer más consultas.
 Por ejemplo:

Pongamos que, después de insertar el departamento, en el código también insertamos
 Un empleado en dicho departamento.
 ¿Qué tenemos que hacer?

```

create procedure SP_INSERT_DEPARTAMENTO
(@nombre nvarchar(50), @localidad nvarchar(50))
as
    declare @maxid int
    select @maxid = max(DEPT_NO) + 1 from DEPT
    insert into DEPT values (@maxid, @nombre, @localidad)
go

```

Para devolver parámetros de salida se utiliza la palabra OUT.

Sintaxis:

```

create procedure SP_PARAM_SALIDA
(@paramin int, @paramsalida int out)
as
    --código
Go

```

Ejemplo

```

create procedure SP_INSERT_DEPARTAMENTO_OUT
(@nombre nvarchar(50)
, @localidad nvarchar(50)
, @maxid int OUT)
as
    --DENTRO DEL PROCEDIMIENTO, DEBEMOS DAR VALOR
    --AL PARAMETRO DE SALIDA
    select @maxid = max(DEPT_NO) + 1 from DEPT
    insert into DEPT values (@maxid, @nombre, @localidad)
go

```

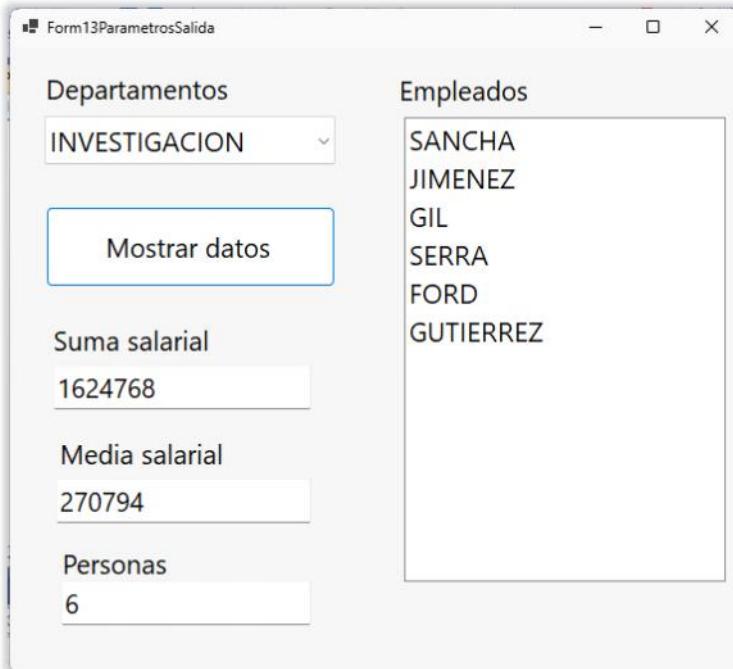
Para llamar al procedimiento con un parámetro de salida necesitamos una variable
 Para almacenar el valor devuelto por el procedimiento
 Se debe incluir la palabra OUTPUT en la llamada.

```

declare @respuesta int
exec SP_INSERT_DEPARTAMENTO_OUT 'LUNES', 'MADRID', @respuesta OUTPUT
--MOSTRARMOS EL VALOR DEVUELTO
print @respuesta

```

Para probarlo vamos a crear un nuevo form llamado **Form13ParametrosSalida**



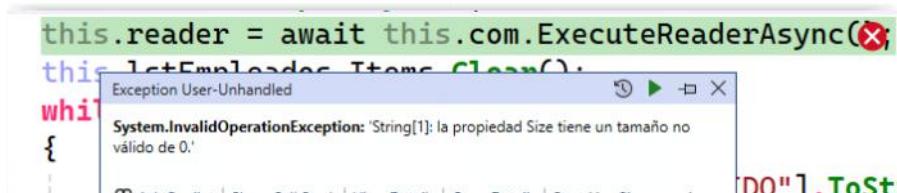
PROCEDIMIENTO ALMACENADO

```

create procedure SP_EMPLEADOS_DEPT_OUT
(@nombre nvarchar(50)
, @suma int OUT
, @media int OUT
, @personas int OUT)
as
    declare @id int
    select @id = DEPT_NO from DEPT
    where DNOMBRE=@nombre
    select * from EMP where DEPT_NO=@id
    select @suma = SUM(SALARIO), @media = AVG(SALARIO)
    , @personas = COUNT(EMP_NO) from EMP
    where DEPT_NO=@id
go

```

Lo primero que veremos es el siguiente error:
A pesar que los parámetros de SALIDA no enviarán ningún valor, es **obligatorio** que tengan
Un Value por defecto al ser llamados.

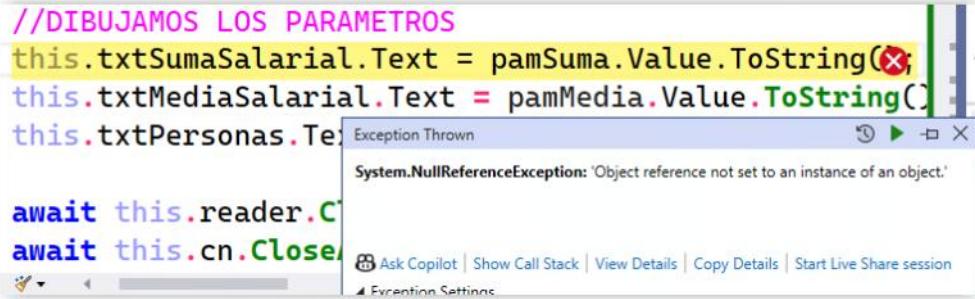


El siguiente error nos indica que los valores son NULL

A pesar de que los parámetros tienen valor, no nos devuelve información.

La información de los parámetros siempre se debe capturar de dos formas posibles.

- 1) Si la consulta del procedimiento es SELECT, se capturan los parámetros después de cerrar el READER.
- 2) Si la consulta es de Acción, se capturan después de cerrar la conexión.



Solución

```
await this.reader.CloseAsync();
//DIBUJAMOS LOS PARAMETROS
this.txtSumaSalarial.Text = pamSuma.Value.ToString();
this.txtMediaSalarial.Text = pamMedia.Value.ToString();
this.txtPersonas.Text = pamPersonas.Value.ToString();

await this.cn.CloseAsync();
this.com.Parameters.Clear();
```

CODIGO FORMULARIO

```
public partial class Form13ParametrosSalida : Form
{
    SqlConnection cn;
    SqlCommand com;
    SqlDataReader reader;

    public Form13ParametrosSalida()
    {
        InitializeComponent();
        string connectionString = @"Data Source=LOCALHOST
\DESARROLLO;Initial Catalog=HOSPITAL;Persist Security Info=True;User ID=SA;Trusted Server Certificate=True";
        this.cn = new SqlConnection(connectionString);
        this.com = new SqlCommand();
        this.com.Connection = this.cn;
        this.LoadDepartamentos();
    }

    private async Task LoadDepartamentos()
    {
        string sql = "SP_ALL_DEPARTAMENTOS";
        this.com.CommandType = CommandType.StoredProcedure;
        this.com.CommandText = sql;
        await this.cn.OpenAsync();
        this.reader = await this.com.ExecuteReaderAsync();
        this.cmbDepartamentos.Items.Clear();
        while (await this.reader.ReadAsync())
        {
            string nombre = this.reader["DNOMBRE"].ToString();
            this.cmbDepartamentos.Items.Add(nombre);
        }
        await this.reader.CloseAsync();
        await this.cn.CloseAsync();
    }

    private async void btnMostrarDatos_Click(object sender, EventArgs e)
    {
        string sql = "SP_EMPLEADOS_DEPT_OUT";
        string nombre = this.cmbDepartamentos.SelectedItem.ToString();
        //PARA LOS PARAMETROS DE ENTRADA PODEMOS UTILIZAR
        //AddWithValue SIN PROBLEMAS
        //PARA LOS PARAMETROS DE ENTRADA ES IMPRESCINDIBLE
        //UTILIZAR OBJETOS Parameter
        this.com.Parameters.AddWithValue("@nombre", nombre);
        SqlParameter pamSuma = new SqlParameter();
        pamSuma.ParameterName = "@suma";
        pamSuma.Value = 0;
        //INDICAMOS LA DIRECCION DEL PARAMETRO
        pamSuma.Direction = ParameterDirection.Output;
        this.com.Parameters.Add(pamSuma);
        SqlParameter pamMedia = new SqlParameter();
        pamMedia.ParameterName = "@media";
        pamMedia.Value = 0;
        pamMedia.Direction = ParameterDirection.Output;
        this.com.Parameters.Add(pamMedia);
        SqlParameter pamPersonas = new SqlParameter();
        pamPersonas.ParameterName = "@personas";
        pamPersonas.Value = 0;
```

```

pamPersonas.Direction = ParameterDirection.Output;
this.com.Parameters.Add(pamPersonas);
this.com.CommandType = CommandType.StoredProcedure;
this.com.CommandText = sql;
await this.cn.OpenAsync();
this.reader = await this.com.ExecuteReaderAsync();
this.lstEmpleados.Items.Clear();
while (await this.reader.ReadAsync())
{
    string apellido = this.reader["APELLIDO"].ToString();
    this.lstEmpleados.Items.Add(apellido);
}
await this.reader.CloseAsync();
//DIBUJAMOS LOS PARAMETROS
this.txtSumaSalarial.Text = pamSuma.Value.ToString();
this.txtMediaSalarial.Text = pamMedia.Value.ToString();
this.txtPersonas.Text = pamPersonas.Value.ToString();

await this.cn.CloseAsync();
this.com.Parameters.Clear();
}
}

```

Quiero llevar esta aplicación que hemos realizado a **Models** y **Repositories**

Creamos un nuevo repositorio llamado **RepositoryParametrosOut**

Sobre **Models**, creamos un nuevo modelo llamado **EmpleadosModelOut**

```

public class EmpleadosModelOut
{
    0 references | 0 changes | 0 authors, 0 changes
    public List<string> Apellidos { get; set; }
    0 references | 0 changes | 0 authors, 0 changes
    public int SumaSalarial { get; set; }
    0 references | 0 changes | 0 authors, 0 changes
    public int MediaSalarial { get; set; }
    0 references | 0 changes | 0 authors, 0 changes
    public int Personas { get; set; }
}

```

Sobre **Repositories** creamos una nueva clase llamada **RepositoryParametrosOut**

REPOSITORYPARAMETROSOUT

```

public class RepositoryParametrosOut
{
    SqlConnection cn;
    SqlCommand com;
    SqlDataReader reader;

    public RepositoryParametrosOut()
    {
        string connectionString = @"Data Source=LOCALHOST
\DESARROLLO;Initial Catalog=HOSPITAL;Persist Security Info=True;User ID=SA;Tru
st Server Certificate=True";
        this.cn = new SqlConnection(connectionString);
        this.com = new SqlCommand();
        this.com.Connection = this.cn;
    }

    public async Task<List<string>> GetDepartamentosAsync()
    {
        string sql = "SP_ALL_DEPARTAMENTOS";
        this.com.CommandType = CommandType.StoredProcedure;
        this.com.CommandText = sql;
        await this.cn.OpenAsync();
        this.reader = await this.com.ExecuteReaderAsync();
        List<string> departamentos = new List<string>();
        while (await this.reader.ReadAsync())
        {
            string nombre = this.reader["DNOMBRE"].ToString();
            departamentos.Add(nombre);
        }
        await this.reader.CloseAsync();
        await this.cn.CloseAsync();
        return departamentos;
    }

    public async Task<EmpleadosModelOut>
        GetEmpleadosModelAsync(string nombre)
    {
        string sql = "SP_EMPLEADOS_DEPT_OUT";
        //PARA LOS PARAMETROS DE ENTRADA PODEMOS UTILIZAR
        //AddWithValue SIN PROBLEMAS
        //PARA LOS PARAMETROS DE ENTRADA ES IMPRESCINDIBLE
        //UTILIZAR OBJETOS Parameter
        this.com.Parameters.AddWithValue("@nombre", nombre);
        SqlParameter pamSuma = new SqlParameter();
        pamSuma.ParameterName = "@suma";
        pamSuma.Value = 0;
        //INDICAMOS LA DIRECCION DEL PARAMETRO
        pamSuma.Direction = ParameterDirection.Output;
        this.com.Parameters.Add(pamSuma);
    }
}

```

```

        SqlParameter pamMedia = new SqlParameter();
        pamMedia.ParameterName = "@media";
        pamMedia.Value = 0;
        pamMedia.Direction = ParameterDirection.Output;
        this.com.Parameters.Add(pamMedia);
        SqlParameter pamPersonas = new SqlParameter();
        pamPersonas.ParameterName = "@personas";
        pamPersonas.Value = 0;
        pamPersonas.Direction = ParameterDirection.Output;
        this.com.Parameters.Add(pamPersonas);
        this.com.CommandType = CommandType.StoredProcedure;
        this.com.CommandText = sql;
        await this.cn.OpenAsync();
        this.reader = await this.com.ExecuteReaderAsync();
        EmpleadosModelOut model = new EmpleadosModelOut();
        List<string> apellidos = new List<string>();
        while (await this.reader.ReadAsync())
        {
            string apellido = this.reader["APELIDO"].ToString();
            apellidos.Add(apellido);
        }
        await this.reader.CloseAsync();
        //GUARDAMOS LOS DATOS EN NUESTRO MODEL
        model.Apellidos = apellidos;
        model.SumaSalarial = int.Parse(pamSuma.Value.ToString());
        model.MediaSalarial = int.Parse(pamMedia.Value.ToString());
        model.Personas = int.Parse(pamPersonas.Value.ToString());

        await this.cn.CloseAsync();
        this.com.Parameters.Clear();
        return model;
    }
}

```

CODIGO FORMULARIO

```

public partial class Form13ParametrosSalida : Form
{
    RepositoryParametrosOut repo;

    public Form13ParametrosSalida()
    {
        InitializeComponent();
        this.repo = new RepositoryParametrosOut();
        this.LoadDepartamentos();
    }

    private async Task LoadDepartamentos()
    {
        List<string> departamentos =
            await this.repo.GetDepartamentosAsync();
        this.cmbDepartamentos.Items.Clear();
        foreach (string name in departamentos)
        {
            this.cmbDepartamentos.Items.Add(name);
        }
    }

    private async void btnMostrarDatos_Click(object sender, EventArgs e)
    {
        string nombre = this.cmbDepartamentos.SelectedItem.ToString();
        EmpleadosModelOut model =
            await this.repo.GetEmpleadosModelAsync(nombre);
        this.lstEmpleados.Items.Clear();
        foreach (string ape in model.Apellidos)
        {
            this.lstEmpleados.Items.Add(ape);
        }
        this.txtSumaSalarial.Text = model.SumaSalarial.ToString();
        this.txtMediaSalarial.Text = model.MediaSalarial.ToString();
        this.txtPersonas.Text = model.Personas.ToString();
    }
}

```

Pongamos que os comento que deseo modificar la aplicación de Ado Net Core
Para que se conecte con vuestra base de datos de Azure.

Pasos a realizar:

- 1) Recuperar la cadena de conexión de Azure
- 2) Dentro del Repository cambiar el String
- 3) Modificar cada cadena de conexión en cada Form/Repository

```

    preferences for changes for authors, or changes
public RepositoryParametrosOut()
{
    string connectionString = @"Data Source=AZURE;Initial Catalog
    this.cn = new SqlConnection(connectionString);
    ...
}

```

- 1) Crear los procedimientos en Azure

FICHEROS DE CONFIGURACION

Los ficheros de configuración nos permiten centralizar características de nuestra aplicación.
Dichas características pueden ser simples Key:Value que representen un Color o una imagen
O pueden ser partes básicas como URL a Updates/Api o Cadenas de conexión o Logger

Este concepto es algo imprescindible dentro del Back.
Dependiendo del entorno dónde estemos trabajando, ya tendremos ficheros de Configuración creados o no.
Por ejemplo, en Apps de Consola o Forms no tenemos nativo, debemos crearlo manualmente

El fichero de settings se puede llamar como queramos y tendrá un formato JSON

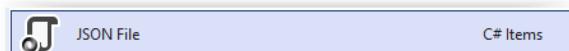
Dentro de dicho formato, tendremos partes reconocibles directamente o partes Que nos podemos inventar.

```
{  
    "ConnectionStrings":  
    {  
        "SqlLocal": "Data Source=Aula",  
        "SqlAzure": "Data Source=Azure"  
    },  
    "Imagenes": {  
        "logo1": "https://imagenes/1.png",  
        "fondo": "https://imagenesback/fondo.png"  
    }  
}
```

Elementos necesarios para implementar el ficheros de Settings.

- 1) Fichero JSON en nuestro proyecto
- 2) Debemos indicar que dicho fichero será copiado en el Output de la aplicación
- 3) Necesitamos los siguientes Nuget:
 - a. Extensions Configuration
 - b. Extensions.Configuration.Json

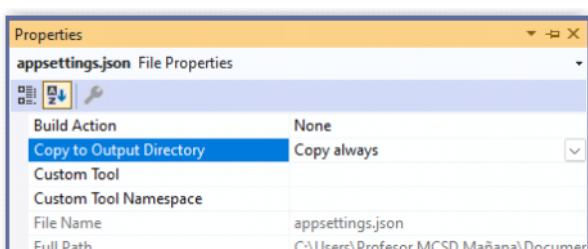
Sobre el proyecto, agregamos un nuevo fichero JSON llamado **appsettings.json**



APPSETTINGS.JSON

```
{  
    "ConnectionStrings": {  
        "SqlTajamar": "Data Source=LOCALHOST\\DESARROLLO;Initial Catalog=Tajamar;Integrated Security=True"  
    },  
    "Imagenes": {  
        "imagen1": "https://platform.vox.com/wp-content/uploads/sites/10/2018/08/10-best-free-photoshop-alternatives-for-windows-and-mac-1024x576.jpg",  
        "imagen2": "https://displaygeek.com/cdn/shop/files/gh0569_wol_1024x1024.jpg"  
    },  
    "Colores": {  
        "letra": "fuchsia",  
        "fondo": "lightgreen"  
    }  
}
```

Sobre nuestro fichero de configuración JSON, Properties y seleccionamos
Copiar al directorio de salida: **ALWAYS**



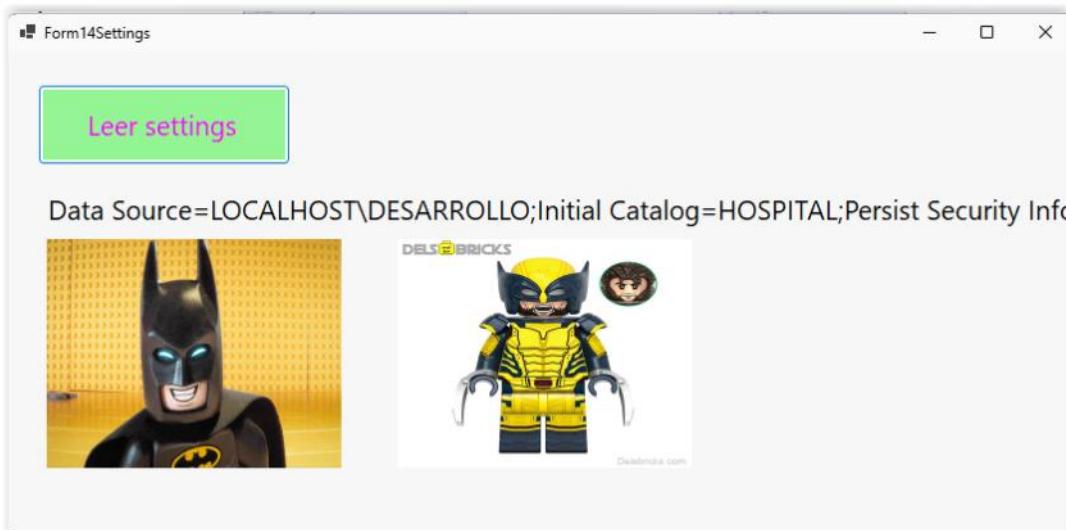
Agregamos los siguientes Nuget

.NET Microsoft.Extensions.Configuration by aspnet, dotnetframework, Microsoft, 3,95B do 9.0.1
Implementation of key-value pair based configuration for Microsoft.Extensions.Configuration.
Includes the memory configuration provider.

.NET Microsoft.Extensions.Configuration.Json by aspnet, dotnetframework, Microsoft, 1,6 9.0.1
JSON configuration provider implementation for Microsoft.Extensions.Configuration. This package
enables you to read your application's settings from a JSON file. You can use JsonConfigurationExt...

Vamos a crear un formulario que NO tiene nada que ver con datos, nos va a servir para
Comprobar cómo podemos crear el fichero de Settings y recuperarlo.

Creamos un nuevo formulario llamado **Form14Settings**



CODIGO FORMULARIO

```
public partial class Form14Settings : Form
{
    public Form14Settings()
    {
        InitializeComponent();
    }

    private void btnLeerSettings_Click(object sender, EventArgs e)
    {
        //NECESITAMOS UN CONSTRUCTOR DE CONFIGURACIONES
        ConfigurationBuilder builder = new ConfigurationBuilder();
        //EN ESTE ENTORNO NO ES NATIVO, POR LO QUE DEBEMOS
        //INDICAR DE FORMA EXPLICITA EL NOMBRE DEL FICHERO Y
        //SU UBICACION
        builder.SetBasePath(Directory.GetCurrentDirectory())
            .AddJsonFile("appsettings.json", false, true);
        //EL OBJETO PARA RECUPERAR LAS KEYS
        IConfigurationRoot configuration = builder.Build();
        //EXISTEN CLAVES QUE YA VIENEN POR DEFECTO: ConnectionStrings
        string connectionString =
            configuration.GetConnectionString("SqlTajamar");
        this.lblCadenaConexion.Text = connectionString;
        //SI NO ES UNA ZONA CONOCIDA COMO Imagenes/Colores
        //LOS DATOS SE RECUPERAN EN CASCADA CON CADA key/subkey
        string imagen1 =
            configuration.GetSection("Imagenes:imagen1").Value;
        string imagen2 =
            configuration.GetSection("Imagenes:imagen2").Value;
        string colorFondo =
            configuration.GetSection("Colores:fondo").Value;
        string colorLetra =
            configuration.GetSection("Colores:letra").Value;
        this.pictureBox1.Load(imagen1);
        this.pictureBox2.Load(imagen2);
        this.btnLeerSettings.BackColor =
            Color.FromName(colorFondo);
        this.btnLeerSettings.ForeColor =
            Color.FromName(colorLetra);
    }
}
```

El siguiente paso es utilizar este código en cualquier Form/Repository por si cambiamos
Nuestra cadena de conexión, que sea modificada para todos los elementos que accedan
A datos.

Necesitamos una clase que nos proporcione la cadena de conexión.

Tenemos dos opciones:

- 1) Que dicha clase nos ofrezca directamente el objeto Configuration

```
//EL OBJETO PARA RECUPERAR LAS KEYS  
IConfigurationRoot configuration = builder.Build();
```

- 1) Podríamos tener un método que nos devuelva la propia ConnectionString

```
string connectionString =  
    configuration.GetConnectionString("SqlTajamar");
```

Para recuperar la cadena de conexión, vamos a realizar una clase con un método
static

Sobre nuestro proyecto, creamos una nueva carpeta llamada **Helpers** y una
Clase llamada **HelperConfiguration**

HELPERCONFIGURATION

```
public class HelperConfiguration  
{  
    0 references | 0 changes | 0 authors, 0 changes  
    public static string GetConnectionString()  
    {  
        ConfigurationBuilder builder = new ConfigurationBuilder();  
        //EN ESTE ENTORNO NO ES NATIVO, POR LO QUE DEBEMOS  
        //INDICAR DE FORMA EXPLICITA EL NOMBRE DEL FICHERO Y  
        //SU UBICACION  
        builder.SetBasePath(Directory.GetCurrentDirectory())  
            .AddJsonFile("appsettings.json", false, true);  
        //EL OBJETO PARA RECUPERAR LAS KEYS  
        IConfigurationRoot configuration = builder.Build();  
        //EXISTEN CLAVES QUE YA VIENEN POR DEFECTO: ConnectionStrings  
        string connectionString =  
            configuration.GetConnectionString("SqlTajamar");  
        return connectionString;  
    }  
}
```

Ya podremos realizar las llamadas desde cualquier clase:

REPOSITORYPARAMETERSOUT

```
public RepositoryParametrosOut()  
{  
    string connectionString =  
        HelperConfiguration.GetConnectionString();  
    this.cn = new SqlConnection(connectionString);  
    this.com = new SqlCommand();  
    this.com.Connection = this.cn;  
}
```

PRACTICA

En un proyecto **nuevo** llamado **AdoNetCorePractica**
creamos un formulario para realizar las siguientes acciones:

- Al iniciar la aplicación, mostraremos los Hospitales (nombre)
- Al seleccionar un nombre, visualizaremos todos los empleados de ese hospital, es decir, Plantilla y Doctores.
- Mostraremos un resumen con el número de personas, la media salarial y la suma Salarial del conjunto.
- Debemos realizar la aplicación con los siguientes requisitos:
 - Procedimientos almacenados
 - Repositories, Models y Configuration

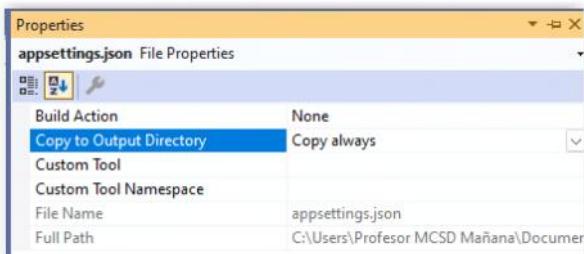
Sobre el proyecto, agregamos los siguientes Nuget

- Microsoft.Data.SqlClient** by Microsoft, nugetsqltools, 752M downloads. Version 5.2.2. Description: The current data provider for SQL Server and Azure SQL databases. This has replaced System.Data.SqlClient. These classes provide access to SQL and encapsulate database-sp...
- Microsoft.Extensions.Configuration** by aspnet, dotnetframework, Microsoft, 9.0.1. Description: Implementation of key-value pair based configuration for Microsoft.Extensions.Configuration. Includes the memory configuration provider.
- Microsoft.Extensions.Configuration.Json** by aspnet, dotnetframework, Microsoft, 9.0.1. Description: JSON configuration provider implementation for Microsoft.Extensions.Configuration. This package enables you to read your application's settings from a JSON file. You can use Jso...

Sobre el proyecto, agregamos un nuevo archivo llamado **appsettings.json**

```
store.org/appsettings.json
{
  "ConnectionStrings": {
    "SqlTajamar": "Data Source=LOCALHOST\\DESARROLLO;Initial
  }
}
```

Ponemos el fichero en sus propiedades como **Copy Always** en **Directory Output**



Para poder recuperar la cadena de conexión vamos a utilizar un Helper
Sobre el proyecto, creamos una nueva carpeta llamada **Helpers** y una clase
Llamada **HelperConfiguration**

HELPERCONFIGURATION

```
public class HelperConfiguration
{
  public static string GetConnectionString()
  {
    ConfigurationBuilder builder =
      new ConfigurationBuilder();
    builder.SetBasePath(Directory.GetCurrentDirectory())
      .AddJsonFile("appsettings.json", false, true);
    IConfigurationRoot configuration = builder.Build();
    string connectionString =
      configuration["ConnectionStrings:SqlTajamar"];
  }
}
```

```

        configuration.GetConnectionString("SqlTajamar");
        return connectionString;
    }
}

```

Creamos un procedimiento para mostrar los datos del nombre de cada Hospital

```

create procedure SP_ALL_HOSPITALES
as
    select * from HOSPITAL
go
exec SP_ALL_HOSPITALES

```

```

create view V_EMPLEADOS_HOSPITAL
as
select DOCTOR.APELLIDO, DOCTOR.ESPECIALIDAD, DOCTOR.SALARIO
, HOSPITAL.HOSPITAL_COD, HOSPITAL.NOMBRE
from DOCTOR
INNER JOIN HOSPITAL
on DOCTOR.HOSPITAL_COD = HOSPITAL.HOSPITAL_COD
UNION
select PLANTILLA.APELLIDO, PLANTILLA.FUNCION, PLANTILLA.SALARIO
, HOSPITAL.HOSPITAL_COD, HOSPITAL.NOMBRE
from PLANTILLA
INNER JOIN HOSPITAL
on PLANTILLA.HOSPITAL_COD = HOSPITAL.HOSPITAL_COD
go
create procedure SP_EMPLEADOS_HOSPITAL
(@nombre nvarchar(50), @suma int out, @media int out, @personas int out)
as
    select * from V_EMPLEADOS_HOSPITAL
    where NOMBRE=@nombre
    select @suma = sum(salario), @media = avg(salario),
    @personas = count(apellido) from V_EMPLEADOS_HOSPITAL
    where NOMBRE=@nombre
go

```

Sobre el proyecto, creamos una nueva carpeta llamada **Models** y una clase llamada **Empleado**

EMPLEADO

```

public class Empleado
{
    public string Apellido { get; set; }
    public string Funcion { get; set; }
    public int Salario { get; set; }
}

```

El procedimiento nos devolverá una colección de Empleados y también tres datos Numéricos.

Sobre **Models** creamos una nueva clase llamada **DatosEmpleados**

DATOSEMPLEADOS

```

public class DatosEmpleados
{
    public List<Empleado> Empleados { get; set; }
    public int SumaSalarial { get; set; }
    public int MediaSalarial { get; set; }
    public int Personas { get; set; }
}

```

Sobre el proyecto, creamos una nueva carpeta llamada **Repositories** y una clase llamada **RepositoryHospital**

REPOSITORYHOSPITAL

```

public class RepositoryHospital
{
    SqlConnection cn;
    SqlCommand com;
    SqlDataReader reader;

    public RepositoryHospital()
    {
        string connectionString =
            HelperConfiguration.GetConnectionString();
        this.cn = new SqlConnection(connectionString);
        this.com = new SqlCommand();
        this.com.Connection = this.cn;
    }

    public async Task<List<string>> GetHospitalesAsync()
    {
        string sql = "SP_ALL_HOSPITALES";
        this.com.CommandType = CommandType.Text;
        this.com.CommandText = sql;
        await this.cn.OpenAsync();
        this.reader = await this.com.ExecuteReaderAsync();
        List<string> hospitales = new List<string>();
        while (await this.reader.ReadAsync())

```

```

    {
        string nombre = this.reader["NOMBRE"].ToString();
        hospitales.Add(nombre);
    }
    await this.reader.CloseAsync();
    await this.cn.CloseAsync();
    return hospitales;
}

public async Task<DatosEmpleados>
GetDatosEmpleadosAsync(string nombre)
{
    string sql = "SP_EMPLEADOS_HOSPITAL";
    this.com.Parameters.AddWithValue("@nombre", nombre);
    SqlParameter pamSuma = new SqlParameter("@suma", 0);
    pamSuma.Direction = ParameterDirection.Output;
    this.com.Parameters.Add(pamSuma);
    SqlParameter pamMedia = new SqlParameter("@media", 0);
    pamMedia.Direction = ParameterDirection.Output;
    this.com.Parameters.Add(pamMedia);
    SqlParameter pamPersonas = new SqlParameter("@personas", 0);
    pamPersonas.Direction = ParameterDirection.Output;
    this.com.Parameters.Add(pamPersonas);
    this.com.CommandType = CommandType.StoredProcedure;
    this.com.CommandText = sql;
    await this.cn.OpenAsync();
    this.reader = await this.com.ExecuteReaderAsync();
    DatosEmpleados data = new DatosEmpleados();
    List<Empleado> empleados = new List<Empleado>();
    while (await this.reader.ReadAsync())
    {
        string apellido = this.reader["APELLODO"].ToString();
        string funcion = this.reader["ESPECIALIDAD"].ToString();
        int salario = int.Parse(this.reader["SALARIO"].ToString());
        Empleado empleado = new Empleado();
        empleado.Apellido = apellido;
        empleado.Funcion = funcion;
        empleado.Salario = salario;
        empleados.Add(empleado);
    }
    await this.reader.CloseAsync();
    int suma = int.Parse(pamSuma.Value.ToString());
    int media = int.Parse(pamMedia.Value.ToString());
    int personas = int.Parse(pamPersonas.Value.ToString());
    await this.cn.CloseAsync();
    this.com.Parameters.Clear();
    data.SumoSalarial = suma;
    data.MediaSalarial = media;
    data.Personas = personas;
    return data;
}
}

```

Por último, escribimos los dibujos en el formulario.

CODIGO FORMULARIO

```

public partial class Form1 : Form
{
    RepositoryHospital repo;

    public Form1()
    {
        InitializeComponent();
        this.repo = new RepositoryHospital();
        this.LoadHospitalesAsync();
    }

    private async Task LoadHospitalesAsync()
    {
        List<string> nombres =
            await this.repo.GetHospitalesAsync();
        this.cmbHospitales.Items.Clear();
        foreach (string nombre in nombres)
        {
            this.cmbHospitales.Items.Add(nombre);
        }
    }

    private async void cmbHospitales_SelectedIndexChanged(object sender, EventArgs e)
    {
        if (this.cmbHospitales.SelectedIndex != -1)
        {
            string nombre =
                this.cmbHospitales.SelectedItem.ToString();
            DatosEmpleados data =
                await this.repo.GetDatosEmpleadosAsync(nombre);
            this.lstEmpleadosHospital.Items.Clear();
            foreach (Empleado emp in data.Empleados)
            {
                this.lstEmpleadosHospital.Items.Add
                    (emp.Apellido + " - " + emp.Funcion + " - "
                     + emp.Salario);
            }
            this.txtSumoSalarial.Text = data.SumoSalarial.ToString();
            this.txtMediaSalarial.Text = data.MediaSalarial.ToString();
            this.txtPersonas.Text = data.Personas.ToString();
        }
    }
}

```

PRACTICA FINAL

Mínimo dos procedimientos, lo mejor es combinar consultas con procedimientos para

La práctica.

Al iniciar la aplicación, cargaremos los departamentos

Al seleccionar un departamento, visualizaremos sus datos en las cajas y también

Veremos los empleados de dicho departamento

Podremos insertar nuevos departamentos

Al seleccionar un Empleado, veremos sus datos en las cajas

Tendremos la posibilidad de modificar los datos del empleado seleccionado

Departamentos	Empleados	
<input type="text"/>	IstEmpleados	<input type="text"/> Apellido
Id <input type="text"/>		<input type="text"/> Oficio
Nombre <input type="text"/>		<input type="text"/> Salario
Localidad <input type="text"/>		<input type="button" value="Update"/>
<input type="button" value="Insertar Departamento"/>		

MVC NET CORE

Lunes, 27 de enero de 2025 9:07

PRIMER PROYECTO INDIVIDUAL

Ya podemos empezar a pensar en una idea para nuestro proyecto personal.
El proyecto tendrá una serie de "fases" para su desarrollo:

- **Versión 1:** Un proyecto utilizando Mvc Net Core
- **Versión 2:** El mismo proyecto pero desplegando todos servicios en Azure.
- **Versión 3:** Mismo proyecto, pero migrando los servicios de Azure a AWS.

La idea del proyecto es abierta completamente, vuestra decisión de lo que
Deseais hacer.

La idea, antes de poneros, debe ser expuesta a **PACO**

Requisitos:

- Base de datos en SQL Server
- En los campos de imagen, los haremos de formato texto (nvarchar) y solamente
Pondremos el nombre de la imagen, nada de rutas.

Producto: Nike Air: [nikeair.png](#)

Necesitamos las imágenes en Local.

Todos los recursos en Local (los que se puedan...)

MVC NET CORE es un patrón de diseño de aplicaciones.

Dicho patrón utiliza Model-View-Controller utilizado por múltiples Frameworks de
Múltiples lenguajes.

Cada MVC es distinto en su funcionalidad, pero no en su forma.

Lo importante de este patrón es la **organización**.

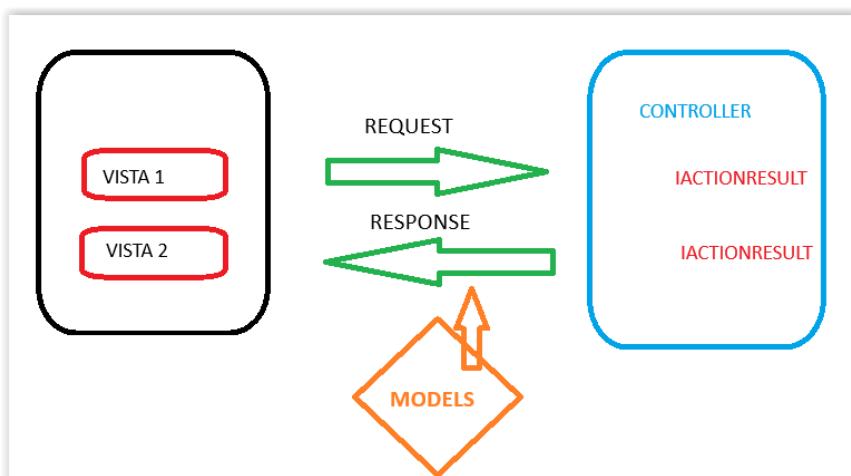
Al principio puede ser algo lioso, pero a la larga nos permite tener un
Absoluto control sobre cada uno de los elementos que conforman la aplicación

Con este patrón, lo que realmente es importante es poder acoplar/desacoplar
Módulos o arquitecturas de forma muy sencilla en nuestra App.

Estamos hablando de aplicaciones BACK, pero son una mezcla de FRONT también.

Mvc Net Core nos permite también poder utilizar nuestras aplicaciones en cualquier
Fabricante y poder hacer despliegues de forma muy sencilla: Linux, Docker, Kubernetes,
Windows...

Todas las capas están separadas.



Nota: Los nombres no son negociables ni en los métodos ni en los objetos.

Hasta ahora, os he dejado un poco "jugar".

```
async Task getCosas()
```

```
async Task GetCosas()
```

Características del patrón MVC Net Core:

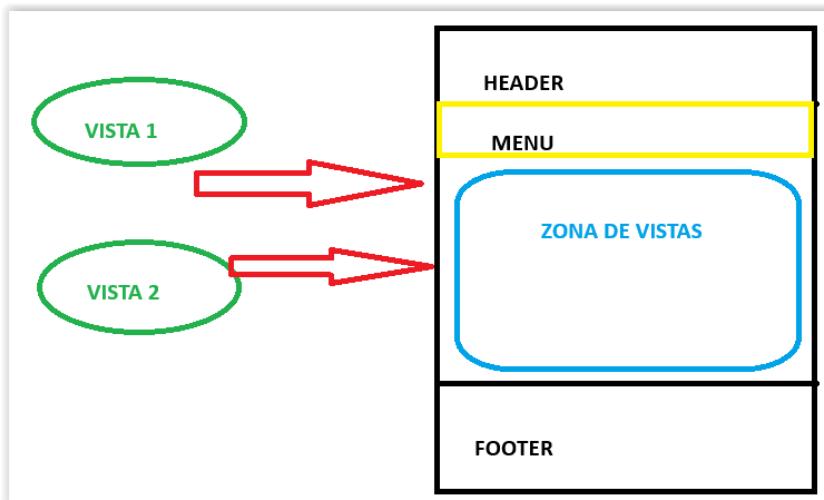
- **VIEWS:** Son la parte visual de nuestra App. Se comunican siempre con un Controller.
No podemos tener vistas sin controlador.
Dentro de su controlador asociado, debemos tener un método **IActionResult** que es
El encargado de administrar la petición de la vista y mostrarla en nuestro proyecto
Utilizan múltiples lenguajes:
 - HTML, CSS y JS, Jquery, Bootstrap...
 - Razor: Lenguaje Back en las vistas, es decir, lenguaje C# incluido en el FrontTodas las vistas estarán incluidas dentro de una carpeta llamada **Views**
Dentro de la carpeta **Views**, tendremos otra carpeta con el nombre de nuestro
Controller asociado.
La extensión para las vistas en Mvc es **cshtml**
- **CONTROLLER:** Utilizan el lenguaje C#. Heredan de la clase **Controller**.
El nombre del controlador debe terminar con la palabra **Controller** y debe
Estar dentro de una carpeta llamada **Controllers**

- **MODELS:** Los modelos son las clases que estarán en las peticiones entre vistas Y controladores, es la información.

No existe una página de inicio como tal. Las páginas son devueltas mediante **MAPPING o ROUTING**

Las vistas no tienen código HTML completo, forman parte de una plantilla principal. Podemos tener múltiples plantillas principales y cada vista irá asociada a una plantilla o varias.

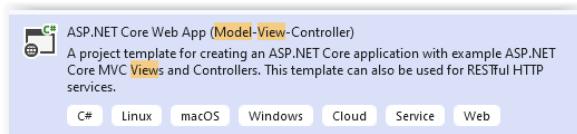
Las vistas están integradas dentro de la plantilla.



La plantilla principal se llama **_Layout** y se encuentra dentro de la carpeta **Views/Shared**

El tipo de proyecto que vamos a utilizar es **ASP Net Core (Model View Controller)**

Comenzamos creando un primer proyecto llamado **PrimerMvcNetCore**



Additional information

ASP.NET Core Web App (Model-View-Controller) C# Linux macOS Windows Cloud

Framework ⓘ .NET 9.0 (Standard Term Support)

Authentication type ⓘ None

Configure for HTTPS ⓘ

Enable container support ⓘ

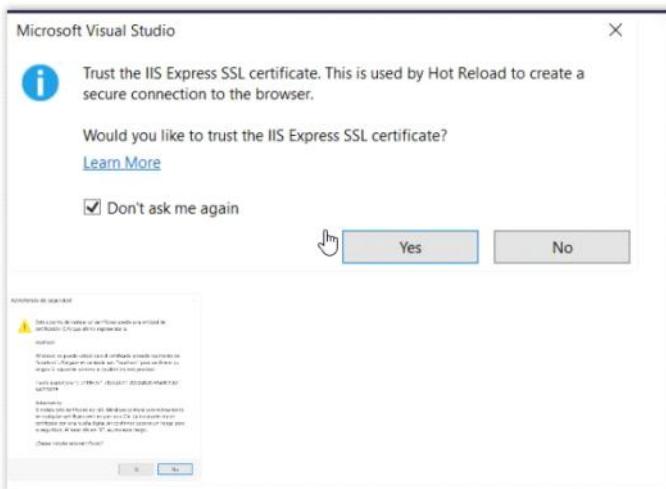
Container OS ⓘ Linux

Container build type ⓘ Dockerfile

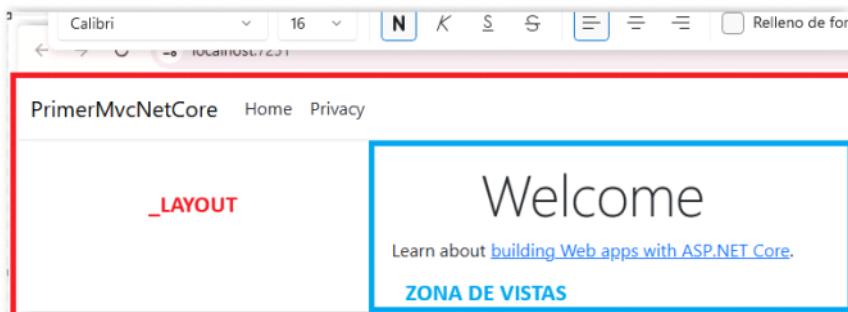
Do not use top-level statements ⓘ

Enlist in .NET Aspire orchestration ⓘ

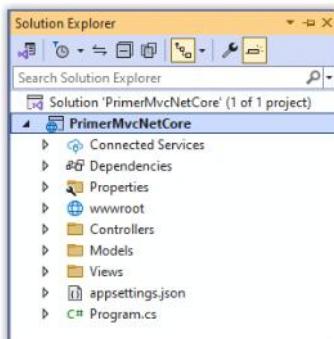
Al ejecutar, nos preguntará si confiamos en nuestro certificado HTTPS, indicamos que SI a todo



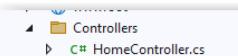
Una vez ejecutado el proyecto, veremos esta información:



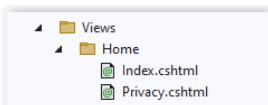
Estructura del proyecto



- **wwwroot:** Esta carpeta es dónde estarán los recursos Front de nuestra aplicación, es decir, dónde tendremos las imágenes o los CSS/JS.
- **Controllers:** En esta carpeta tendremos las clases Controllers de C#. Deben estar aquí de forma obligatoria
- **Models:** Es dónde debemos incluir los modelos de nuestra aplicación. No es obligatorio tener los modelos en dicha ubicación, pero sí recomendable.
- **Views:** Es la zona de las vistas y el diseño HTML. Tendremos una carpeta por cada Controller.



Dentro de **Views** veremos una carpeta con el nombre del **Controller** sin la palabra **Controller**



Estamos viendo que tenemos dos Vistas.
En el controlador **HomeController** tendremos dos métodos **IActionResult** que se llamarán como Las vistas.

```

public IActionResult Index()
{
    return View();
}

public IActionResult Privacy()
{
    return View();
}

```

- **appsettings.json:** Es el fichero de configuración de nuestra aplicación, es dónde tendremos las cadenas de conexión o elementos como la seguridad, por ejemplo.
 - **Program.cs:** Es la clase más importante de todo el proyecto. Es la jefa. Todo lo que necesitemos realizar, debemos pedir permiso en esta clase, es decir, Declararlo.
- En esta clase es dónde se declaran todas las características del Middleware:

- Cookies, Seguridad, Ficheros en el servidor, Dependencias, Session, Inicio de App...

Vamos a visualizar nuestro primer controller.

Sobre la carpeta **Controllers** creamos un nuevo controlador de tipo **MVC Controller Empty** llamado **PruebaController**



Creamos un nuevo método llamado **PrimeraPagina**

Actualmente, tenemos dos métodos, lo que quiere decir que tendremos dos Vistas
En este Controller: Index, PrimeraPagina

```

public class PruebaController : Controller
{
    0 references
    public IActionResult PrimeraPagina()
    {
        return View();
    }

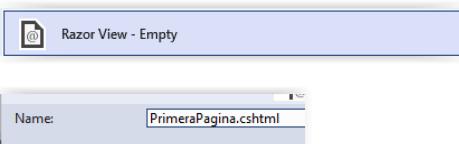
    0 references
    public IActionResult Index()
    {
        return View();
    }
}

```

El controlador administra las peticiones de las Vistas.

A continuación, debemos incluir Vistas para ser dibujadas.
Lo primero será crear una "Zona" de Controller para nuestras vistas.
Sobre **Views** creamos una nueva carpeta con el nombre del **Controller: Prueba**

Dentro de la carpeta **Prueba**, creamos una nueva página llamada **PrimeraPagina.cshtml**



Incluimos cualquier código HTML. Solamente debemos incluir el dibujo de nuestra Vista,
Nada más.

PRIMERAPAGINA.CSHTML

```

<h1 style="color:blue">
    Mi primera página MVC de Lunes!!!!
</h1>

```

El siguiente paso es tener un Link para poder navegar a dicha Vista.

El enlace lo vamos a incluir dentro de nuestra página `_Layout.cshtml`.

Los Links a vistas son enlaces que contienen código Razor.
Son etiquetas HTML vitaminadas. Se llaman HTML RAZOR TAGS

Para llamar a cualquier Link en HTML, simplemente incluimos `<a>`

```
<a href="https://www.google.com">Ir a Google</a>
```

Para llamar a una vista en un controlador debemos utilizar la siguiente sintaxis:

```
<a asp-controller="Controller" asp-action="IActionResult">  
    Texto del Link  
</a>
```

Abrimos `_Layout.cshtml` dentro de `Views/Shared`

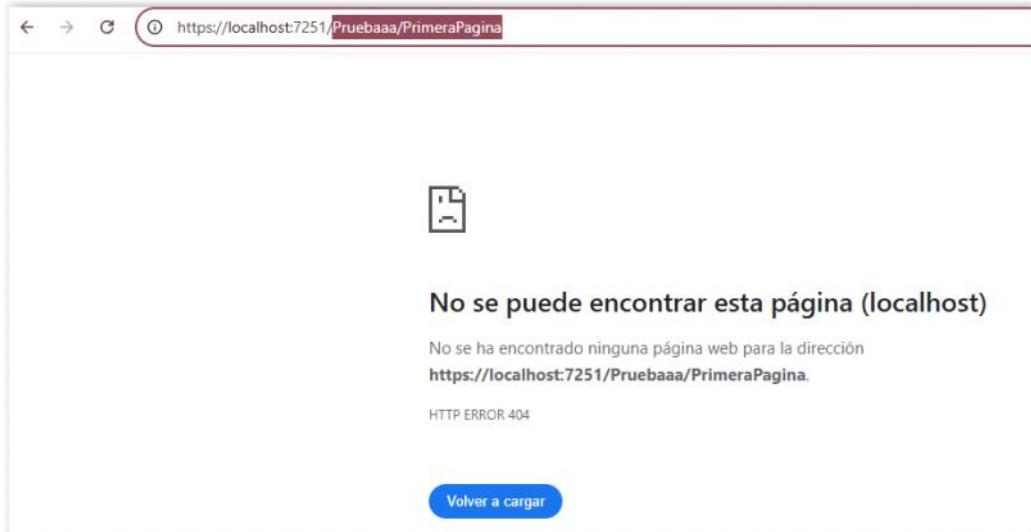
```
<a asp-controller="Prueba" asp-action="PrimeraPagina">  
    Mi primera página  
</a>
```

POSIBLES ERRORES

Vista sin Controller o Controller sin Vista.

- 1) Es posible que escribamos mal el nombre del Controlador o el nombre del Action En nuestro Link.

```
<a asp-controller="Pruebaaaa" asp-action="PrimeraPagina">  
    Mi primera página  
</a>
```



- 2) El `IActionResult` del controlador no existe, pero si existe la página dentro de `Views`
Debemos mirar el nombre del `IActionResult` que hemos puesto y que coincida con
El nombre de la vista.

```
<a asp-controller="Prueba" asp-action="Index">  
    Mi primera página  
</a>
```

No encuentra la página, pero SI tenemos un método `IActionResult` en `PruebaController`

```
public class PruebaController : Controller  
{  
    0 references  
    public IActionResult PrimeraPagina()  
    {  
        return View();  
    }  
  
    0 references  
    public IActionResult Index()  
    {  
        return View();  
    }  
}
```

An unhandled exception occurred while processing the request.

InvalidOperationException: The view 'Index' was not found. The following locations were searched:
/Views/Prueba/Index.cshtml
/Views/Shared/Index.cshtml

Microsoft.AspNetCore.Mvc.ViewEngines.ViewEngineResult.EnsureSuccessful(IEnumerable<string> originalLocations)

Stack Query Cookies Headers Routing

InvalidOperationException: The view 'Index' was not found. The following locations were searched: /Views/Prueba/Index.cshtml
/Views/Shared/Index.cshtml

Microsoft.AspNetCore.Mvc.ViewEngines.ViewEngineResult.EnsureSuccessful(IEnumerable<string> originalLocations)

LENGUAJE RAZOR

Es un lenguaje que se utiliza en las vistas y sirve para complementar los dibujos del HTML
Mediante código Back (C#)

Nos permite realizar peticiones no solo a los controllers, sino también a otras clases o
Elementos que tengamos en nuestra App.

Lo bueno que tiene el lenguaje es que es inteligente. Sabe perfectamente cuando
Tenemos código C# y cuando tenemos código HTML y cambia solo.

Para escribir código Razor se utiliza el símbolo @

Creamos la página `Index.cshtml` dentro de la carpeta `Views/Prueba`

```
@*
    COMENTARIOS EN RAZOR
*@
```

DECLARACION DE VARIABLES A NIVEL DE PAGINA

```
@{
    //ESTO ES CODIGO C#
    int valor = 14;
    string texto = "Hoy es lunes";
    valor += 1;
    texto += " y mañana martes";
    texto = texto.ToUpper();
}
```

DIBUJAR VARIABLES

```
<h2 style="color:red">
    Texto: @texto, Número: @valor
</h2>
```

CODIGO LOGICO CON HTML

```
@if (valor > 0){
    //CODIGO C#
    texto = "POSITIVO";
    <h3 style="color:green">@texto</h3>
} else{
    //CODIGO C#
    texto = "NEGATIVO";
    <h3 style="color:red">@texto</h3>
}
```

Y podremos visualizar los resultados:

[Mi primera página | Código Razor](#)

DECLARACION DE VARIABLES A NIVEL DE PAGINA DIBUJAR VARIABLES

Texto: HOY ES LUNES Y MAÑANA MARTES, Número: 15

CODIGO LOGICO CON HTML

POSITIVO

En todo Framework/Arquitectura inicial lo más importante es manejar la información.

La información se utiliza de dos formas:

- 1) Enviar la información del Controller a la Vista.
- 2) Enviar información de la Vista al Controller

Dentro de MVC existen varias sintaxis para realizar lo mismo, todo depende lo que Vayamos a realizar. Por ejemplo, los Links.

Podemos crear un Link mediante **Tag Helper**

```
<a asp-controller=...>
```

Podemos crear un Link mediante código **Razor**

```
@Html.Link("Texto", "Controller", "Action")
```

ENVIAR INFORMACION CONTROLLER --> VIEW

Para enviar información tenemos dos opciones:

- 1) **ViewBag/ViewData**: Son exactamente lo mismo las dos opciones, apuntan al Mismo espacio de memoria, es decir, son la misma variable. La diferencia está en Su sintaxis.
Nos permiten enviar información simple a nuestras páginas (Mentira)
No existe control en las vistas sobre su declaración, es decir, si vienen vacías o Me he equivocado en el nombre, no pasa nada de nada.
No diferencia mayúsculas de minúsculas en el nombre de las variables.
El nombre de las variables es dinámico, es decir, nos lo inventamos nosotros.

```
ViewBag.NombrePropiedad = 18;  
ViewData["NOMBREPROPIEDAD"] = 21;
```

- 2) **Models**: Un Model es un objeto enviado a la vista.
Dicho Model representa lo que necesitamos dibujar en la vista.
El compilador detecta si el Model está vacío en su lectura.
Solamente podemos enviar un Model a cada página

Comenzamos creando un nuevo **Controller** de tipo **Mvc Empty** llamado **InformacionController**



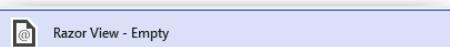
```
public IActionResult ControladorVista()  
{  
    //VAMOS A ENVIAR INFORMACION SIMPLE A NUESTRA VISTA  
    ViewData["NOMBRE"] = "Alumno";  
    ViewData["EDAD"] = 24;  
    ViewBag.DiaSemana = "Lunes";  
    return View();  
}
```

Sobre Views creamos una nueva carpeta llamada **Informacion** y una vista llamada **Index.cshtml**

INDEX.CSHTML

```
<h1>Menú información</h1>  
<ul>  
    <li>  
        <a asp-controller="Informacion"  
            asp-action="ControladorVista">  
            Controlador --> Vista  
        </a>  
    </li>  
</ul>
```

Creamos una nueva vista llamada **ControladorVista.cshtml** en **Views/Informacion**



```
<h1 style="color:blue">
    Información Controller --> View
</h1>
<h2 style="color:fuchsia">
    Nombre @ViewData["NOMBRE"]
</h2>
<h2 style="color:forestgreen">
    Edad: @ViewData["EDAD"]
</h2>
<h2 style="color:blueviolet">
    Día semana: @ViewBag.DiaSemana
</h2>
```

Por último, incluimos un Link dentro de _Layout.cshtml para mostrar el menú de Index.cshtml de Informacion

_LAYOUT.CSHTML

```
<li class="nav-item">
    <a class="nav-link"
        asp-controller="Informacion"
        asp-action="Index">
        Información
    </a>
</li>
```

Y podremos visualizar los valores de la información

Mi primera página | Código Razor

Información Controller --> View

Nombre Alumno

Edad: 24

Día semana: Lunes

Si deseamos enviar información algo más compleja (Model) debemos aplicar esa Lógica.
Podemos combinar Model y ViewData, no importa.

- Un Model es un objeto con sus propiedades
- Solamente un Model por Vista
- Para enviar la información a la vista se utiliza la siguiente Sintaxis:

```
return View(MODEL);
```

- Los modelos pueden estar en cualquier carpeta, pero los pondremos en la carpeta De **Models**

Para recibir nuestro Modelo en la vista tenemos que seguir dos normas:

- 1) **@model**: Declara el tipo de model a recibir en la vista
- 2) **@Model**: Es el propio objeto Model recibido en la vista.

Sobre la carpeta **Models** creamos una nueva clase llamada **Persona**

```
public class Persona
{
    0 references
    public string Nombre { get; set; }
    0 references
    public string Email { get; set; }
    0 references
    public int Edad { get; set; }
}
```

A continuación, sobre **InformacionController** creamos un nuevo método **IActionResult**

Llamado ControladorViewModel

INFORMACIONCONTROLLER

```
public IActionResult ControladorViewModel()
{
    Persona persona = new Persona();
    persona.Nombre = "Alumno";
    persona.Email = "alumno@email.com";
    persona.Edad = 27;
    return View( persona );
}
```

Sobre Views/Informacion creamos una nueva vista llamada
ControladorViewModel.cshtml

```
@model Persona

<h1>
    Información Controller -> View Model
</h1>

<h2 style="color:blue">
    @Model.Nombre, @Model.Email, @Model.Edad
</h2>
```

Y ya debería ser funcional.

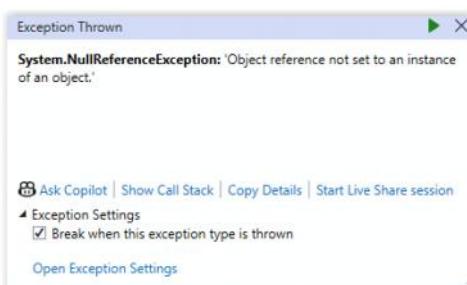
[Mi primera página](#) | Código Razor

Información Controller -> View Model

Alumno, alumno@email.com, 27

El model es obligatorio enviarlo, si se nos olvida enviarlo, nos dará una excepción

```
persona.Email = "alumno@email.com";
persona.Edad = 27;
return View( );
```



Si el Model es opcional, deberíamos preguntar por su contenido

```
@if (Model != null){
    <h2 style="color:blue">
        @Model.Nombre, @Model.Email, @Model.Edad
    </h2>
} else{
    <h1 style="color:red">No tenemos Model</h1>
}
```

ENVIAR INFORMACION VISTA AL CONTROLADOR

Dentro de MVC tenemos dos métodos diferentes para las peticiones a un controller:

- **GET/QUERYSTRING:** La información es enviada mediante la URL, es decir, los datos Son visibles en la cabecera del explorador Web.

Siempre debemos tener una petición GET para las vistas.
Los métodos **IActionResult** son GET por defecto, pero podríamos indicar, de forma explícita que un método será GET mediante la decoración [**HttpGet**]

```
[HttpGet]  
0 references  
public IActionResult Index()  
{  
    return View();  
}
```

- **POST:** Esta petición es opcional. Si incluimos un POST debemos tener un GET también. Se utiliza para capturar la información de los **Forms**. La información no es visible en el envío de los datos.

Para poder utilizar un método POST necesitaremos realizar dos acciones:

- a. HTML: <form method="post">
- b. CONTROLLER: Utilizar la decoración [**HttpPost**]

Siempre un método GET:

```
[HttpGet]  
0 references  
public IActionResult Index()  
{  
    return View();  
}  
  
[HttpPost]  
0 references  
public IActionResult Index()  
{  
    return View();  
}
```

Comenzamos viendo cómo podemos enviar información con GET.

Para enviar información vamos a utilizar código **RAZOR** para generar los Links

Este código genera un Link <a>

```
@Html.ActionLink("Texto Link", "Action", "Controller")
```

Para enviar la información desde el Link de Razor se utilizan objetos anónimos.

```
new {  
    Propiedad1 = valor1, Propiedad2 = valor2  
}
```

La sintaxis es la siguiente:

```
@Html.ActionLink("Texto Link", "Action", "Controller"  
, new { variable1 = "valor1", variable2 = "valor2"})
```

El siguiente paso es averiguar cómo podemos recuperar la información de los Links dentro de Nuestro método **IActionResult** del Controller.

Nota: La información se recupera IGUAL en los métodos GET/POST

La información se captura en los métodos mediante variables.

Nota: Los nombres de variables son únicos y muy importantes. La información se realiza Mediante **BINDING**, es decir, el nombre de la variable se enlaza con el otro nombre de variable. El nombre de la variable enviada en el @Html.ActionLink debe ser igual que el nombre De la variable recibida en el método **IActionResult**

```
[HttpGet]  
0 references  
public IActionResult Index  
(string variable1, string variable2)  
{  
    return View();  
}
```

Sobre **InformacionController** agregamos un nuevo método **IActionResult** llamado **VistaControllerGet**

```
//VAMOS A RECIBIR DOS PARAMETROS
0 references
public IActionResult VistaControllerGet
    (string saludo, int year)
{
    //PARA COMPROBAR QUE LOS HEMOS RECIBIDO,
    //GUARDAMOS LOS PARAMETROS EN ViewData
    //PARA VER LA INFORMACION
    ViewData["DATA"] = "Hola " + saludo
        + " en el año " + year;
    return View();
}
```

Sobre Views/Informacion creamos una nueva vista llamada `VistaControllerGet.cshtml`

```
<h1>
    Información Vista --> Controller GET
</h1>

<h2 style="color:red">
    @ViewData["DATA"]
</h2>
```

Para enviar la información lo haremos con un Link dentro de la vista `Index.cshtml`

`INDEX.CSHTML`

```
@Html.ActionLink("Vista Controller Get", "VistaControllerGet"
, "Informacion"
, new { saludo = "Hola caracola", year = 2025})
```

Ejecutamos nuestra App y deberíamos visualizar la información

[Mi primera página | Código Razor](#)

Información Vista --> Controller GET

Hola Hola caracola en el año 2025

Los parámetros son opcionales?

¿Qué sucede si no recibimos parámetros?

Creamos un nuevo Link dentro de `Index.cshtml` sin enviar los parámetros

```
@Html.ActionLink("Vista Controller Get VACIO"
, "VistaControllerGet"
, "Informacion")
```

Los parámetros al enviarlos son opcionales.

¿Qué sucede si enviamos parámetros y nos los recibimos?

```
public IActionResult VistaControllerGet()
    //((string saludo, int year)
{
    //PARA COMPROBAR QUE LOS HEMOS RECIBIDO,
    //GUARDAMOS LOS PARAMETROS EN ViewData
    //PARA VER LA INFORMACION
    //ViewData["DATA"] = "Hola " + saludo
    //    + " en el año " + year;
    return View();
}
```

Todos los parámetros son opcionales, tanto al enviarlos como al recibirlas.

En algunas ocasiones, pudiera ser que recibamos parámetrosopcionales.

Si recibimos el parámetro, hacemos una acción y si no lo recibimos haremos otra.

Vamos a comprobar que sucede con el parámetro **saludo**

Modificamos el método **IActionResult** del controller

```
//VAMOS A RECIBIR DOS PARAMETROS
0 references | serragutti, 10 minutes ago | 1 author, 1 change
public IActionResult VistaControllerGet
    (string saludo, int year)
{
    if (saludo != null)
    {
        ViewData["DATA"] = "Hola " + saludo
            + " en el año " + year;
    }
    else
    {
        ViewData["DATA"] = "Aquí nadie saluda ya...";
    }
    return View();
}
```

A continuación, sobre **Index.cshtml** incluimos dos Links, uno con saludo y otro sin saludo.

INDEX.CSHTML

```
<li>
    @Html.ActionLink("Vista Controller Get", "VistaControllerGet"
        , "Informacion"
        , new { saludo = "Hola caracola", year = 2025})
</li>
<li>
    @Html.ActionLink("Vista Controller Get VACIO"
        , "VistaControllerGet"
        , "Informacion", new { year = 2025 })
</li>
```

Hemos visto que con **saludo** podemos hacer un IF con sus parámetros opcionales.

Vamos a realizar lo mismo, pero en lugar de utilizar **saludo**, utilizamos **year**

INDEX.CSHTML

```
<li>
    @Html.ActionLink("Vista Controller Get", "VistaControllerGet"
        , "Informacion"
        , new { saludo = "Hola caracola", year = 2025})
</li>
<li>
    @Html.ActionLink("Vista Controller Get VACIO"
        , "VistaControllerGet"
        , "Informacion", new { saludo = "Estoy saludando!!!!" })
</li>
```

Y modificamos el código del método **IActionResult**

Podemos comprobar que nos indica que el IF siempre será **true**

```

public IActionResult VistaControllerGet
    (string saludo, int year)
{
    //PREGUNTAMOS SI AÑO HA VENIDO EN LA INFORMACION
    if (year != null)
    {
        ViewData["DATA"] = "Aqui no tenemos tiempo";
    }
}

```

En los tipos de datos primitivos, excepto `string`, no podemos admitir NULOS

```

int num = null;
string dato = null;

```

Por este hecho, no podemos preguntar si un dato primitivo es null o no (opcional)

Existe un nuevo tipo de dato primitivo que admite nulos para estas situaciones

`System.Nullable<PRIMITIVO>`

```

System.Nullable<int> num = null;
string dato = null;

```

Este tipo de dato se utiliza cuando el tipo de dato recibido en el método es opcional y También es primitivo, para poder preguntar su valor null o no.

```

//VAMOS A RECIBIR DOS PARAMETROS
public IActionResult VistaControllerGet
    (string saludo, System.Nullable<int> year)
{
    //PREGUNTAMOS SI AÑO HA VENIDO EN LA INFORMACION
    if (year != null)
    {
        ViewData["DATA"] = "Hola " + saludo
            + " en el año " + year;
    }
    else
    {
        ViewData["DATA"] = "Aqui no tenemos tiempo";
    }
    return View();
}

```

Existe otra sintaxis para indicar que un tipo primitivo admitirá nulos.

```

public IActionResult VistaControllerGet
    (string saludo, int? year)
{
}

```

ENVIAR INFORMACION DESDE LA VISTA CON POST

La información mediante POST se realiza con Formularios.

Las etiquetas `<form>` deben incluir `method="post"`

Si no tienen `method="post"` todo funciona pero leerán el método GET.

La captura de los parámetros en el método `IActionResult` no cambia, se realiza Mediante **binding** con el **name** de los objetos HTML de formulario.

Debemos tener un método GET además de un método POST

El método POST debe tener la decoración `[HttpPost]`

Para comprobar el funcionamiento vamos a crear una vista en `Views/Informacion`
Llamada `VistaControllerPost.cshtml`

`VISTACONTROLLERPOST.CSHTML`

```

<h1 style="color:blue">
    Vista Controller POST
</h1>

<form method="post">
    <label>Nombre</label>
    <input type="text" name="cajanombre"
        class="form-control"/><br/>
    <label>Email</label>
    <input type="text" name="cajaemail"
        class="form-control"/><br/>
    <label>Edad</label>
    <input type="text" name="cajaedad"
        class="form-control"/>
    <button class="btn btn-info">
        Enviar información
    </button>
</form>
<h1 style="color:green">@ViewData["DATA"]</h1>

```

Sobre **InformacionController** agregamos un nuevo método para capturar la información del Formulario.

INFORMACIONCONTROLLER

```

0 references | 0 changes | 0 authors, 0 changes
public IActionResult VistaControllerPost()
{
    return View();
}

[HttpPost]
0 references | 0 changes | 0 authors, 0 changes
public IActionResult VistaControllerPost(
    string cajanombre, string cajaemail, int cajaedad)
{
    ViewData["DATA"] = "Nombre: " + cajanombre
        + ", Email: " + cajaemail
        + ", Edad: " + cajaedad;
    return View();
}

```

Por último, incluimos un Link en **Index.cshtml** y comprobamos nuestra App

```

<li>
    <a asp-controller="Informacion"
        asp-action="VistaControllerPost">
        Vista Controller POST
    </a>
</li>

```

Y podremos comprobar la funcionalidad

Vista Controller POST

Nombre
Alumno

Email
alumno@email.com

Edad
26

Enviar información

Nombre: Alumno, Email: alumno@email.com, Edad: 26

Como podemos visualizar, los valores de las cajas vienen directamente, es decir, no Viene la propia caja.

[HttpPost]

0 references | 0 changes | 0 authors, 0 changes

public IActionResult VistaControllerPost

(string cajanombre, string cajaemail, int cajaedad)

Cuando trabajamos en BACK al recuperar los valores, en el **name** no se pone el nombre
De la caja, sino que se pone el nombre del valor de la variable.

Por norma, el **Id** del FRONT si se llama como el objeto **input** (**cajanombre**)
Por norma, el **name** del BACK se llama como el valor del **input** (**nombre**)

Modificamos el código HTML y el código del Post para tener los nombres de forma correcta.

INFORMACIONCONTROLLER

[HttpPost]

0 references | 0 changes | 0 authors, 0 changes

public IActionResult VistaControllerPost

(string nombre, string email, int edad)

{

 ViewData["DATA"] = "Nombre: " + nombre
 + ", Email: " + email
 + ", Edad: " + edad;
 return View();

}

VISTACONTROLLERPOST.CSHTML

```
<form method="post">  
    <label>Nombre</label>  
    <input type="text" name="nombre"  
        class="form-control"/><br/>  
    <label>Email</label>  
    <input type="text" name="email"  
        class="form-control"/><br/>  
    <label>Edad</label>  
    <input type="text" name="edad"  
        class="form-control"/>  
    <button class="btn btn-info">  
        Enviar información  
    </button>  
</form>
```

Por último, existe un concepto llamado **Model Binding**

Este concepto nos permite **recibir** un objeto en lugar de variables.

Necesitamos una serie de condiciones para poder hacer un **Model Binding**

- 1) El nombre de los objetos HTML deben llamarse como las propiedades del Model
- 2) No diferencia mayúsculas de minúsculas
- 3) No es necesario enviar todas las propiedades, solamente enlazará las propiedades que reciba el objeto.
- 4) En lugar de recibir N parámetros, se recibe una clase **Model**

Nota: Debemos tener cuidado con el nombre de la variable del **binding**.
Sí supera 15 caracteres no funciona.

Vamos a enlazar el Model de **Persona**

```
public class Persona  
{  
    2 references | serraguti, 1 hour ago | 1 author, 1 change  
    public string Nombre { get; set; }  
    2 references | serraguti, 1 hour ago | 1 author, 1 change  
    public string Email { get; set; }  
    2 references | serraguti, 1 hour ago | 1 author, 1 change  
    public int Edad { get; set; }  
}
```

Regla número 1: Los nombres de HTML deben coincidir con los nombres de las propiedades

```

<form method="post">
    <label>Nombre</label>
    <input type="text" name="nombre"
           class="form-control"/><br/>
    <label>Email</label>
    <input type="text" name="email"
           class="form-control"/><br/>
    <label>Edad</label>
    <input type="text" name="edad"
           class="form-control"/>
    <button class="btn btn-info">
        Enviar información
    </button>
</form>

```

En el **IActionResult** debemos recibir un objeto en lugar de variables sueltas.

INFORMACIONCONTROLLER

```

[HttpPost]
0 references | 0 changes | 0 authors, 0 changes
public IActionResult VistaControllerPost
    (Persona persona)
{
    ViewData["DATA"] = "Nombre: " + persona.Nombre
        + ", Email: " + persona.Email
        + ", Edad: " + persona.Edad;
    return View();
}

```

Si recibimos **menos** parámetros para el objeto **Persona**, simplemente no falla sino que
No hace el binding en esa propiedad

```

<form method="post">
    <label>Nombre</label>
    <input type="text" name="nombre"
           class="form-control"/><br/>
    @*     <label>Email</label>
    <input type="text" name="email"
           class="form-control"/><br/> *@
    <label>Edad</label>

```

Podríamos tener la posibilidad de recibir más parámetros que no tengan que ver con
El **Model**.
Dichos parámetros deben ser incluidos **DESPUES** de la declaración del Model en el método.

```

        class="form-control"/>
    <label>Aficiones</label>
    <input type="text" name="aficiones"
           class="form-control"/><br/>
    <button class="btn btn-info">
        Enviar información
    </button>

```

En el método **IActionResult** del **POST** incluimos la nueva variable **aficiones** siempre
Después del **Model Binding**

INFORMACIONCONTROLLER

```

[HttpPost]
0 references | 0 changes | 0 authors, 0 changes
public IActionResult VistaControllerPost
    (Persona persona, string aficiones)
{
    ViewData["DATA"] = "Nombre: " + persona.Nombre
        + ", Email: " + persona.Email
        + ", Edad: " + persona.Edad
        + ", Aficiones: " + aficiones;
    return View();
}

```

PRACTICA SUMAR DOS NUMEROS

Debemos crear un nuevo controlador llamado **Matematicas**

Dicho controlador tendrá un menú en una vista **Index.cshtml** para ir al resto de páginas.

Tendremos dos vistas:

- 1) **SumarNumerosGet**: Recibirá dos números mediante GET y dibujará la suma de los Dos números en la página.

La suma de $4 + 6 = 10$

- 2) **SumarNumerosPost**: Tendremos un formulario y recibiremos dos números en dicho Formulario.
Debemos mostrar la suma de los dos números después de pulsar el botón.

La suma de $4 + 6 = 10$

Sobre la carpeta **Controllers** creamos un nuevo controlador llamado **MatematicasController**

MATEMATICASCONTROLLER

```

public class MatematicasController : Controller
{
    public IActionResult Index()
    {
        return View();
    }

    public IActionResult SumarNumerosGet
        (int numero1, int numero2)
    {
        int suma = numero1 + numero2;
        ViewData["MENSAJE"] = "La suma de "
            + numero1 + " + " + numero2
            + " = " + suma;
        return View();
    }
}

```

Sobre Views creamos una nueva carpeta llamada **Matematicas** y una vista llamada **Index.cshtml**

INDEX.CSHTML

```

<h1>Menú matemáticas Controller</h1>
<ul>
    <li>
        @Html.ActionLink("Sumar 5 + 6", "SumarNumerosGet",
            "Matematicas",
            new { numero1 = 5, numero2 = 6})
    </li>
    <li></li>
</ul>

```

Sobre Views/Matematicas creamos una vista llamada **SumarNumerosGet.cshtml**

SUMARNUMEROSET.CSHTML

```

<h1>Sumar números Get</h1>
<h2 style="color:blue">
    @ViewData["MENSAJE"]
</h2>

```

Para acceder al menú de Index lo que haremos será incluir un nuevo Tag Helper Link dentro De **_Layout.cshtml** en **Views/Shared**

```

<li class="nav-item">
    <a class="nav-link"
        asp-controller="Matematicas"
        asp-action="Index">
        Matemáticas
    </a>
</li>

```

Y comprobamos el resultado

Sumar números Get

La suma de $5 + 6 = 11$

A continuación, el siguiente paso es realizar el método `SumarNumerosPost`

Sobre `Views/Matematicas` agregamos una nueva vista llamada `SumarNumerosPost.cshtml`

`SUMARNUMEROSPOST.CSHTML`

```
<h1 style="color:fuchsia">
    Sumar números POST
</h1>

<form method="post">
    <label>Número 1</label>
    <input type="text" name="numero1"
        class="form-control"/><br/>
    <label>Número 2</label>
    <input type="text" name="numero2"
        class="form-control"/><br/>
    <button class="btn btn-success">
        Sumar números
    </button>
</form>
<h3 style="color:red">
    @ViewData["MENSAJE"]
</h3>
```

Sobre `MatematicasController` creamos un nuevo método para GET y otro para POST que Recibirá las variables del Form.

`MATEMATICASCONTROLLER`

```
public IActionResult SumarNumerosPost()
{
    return View();
}

[HttpPost]
public IActionResult
    SumarNumerosPost(int numero1, int numero2)
{
    int suma = numero1 + numero2;
    ViewData["MENSAJE"] = "La suma de "
        + numero1 + " + " + numero2
        + " = " + suma;
    return View();
}
```

Por último, incluimos un Link dentro de `Index.cshtml`

`INDEX.CSHTML`

```
<li>
    <a asp-controller="Matematicas"
        asp-action="SumarNumerosPost">
        Sumar números POST
    </a>
</li>
```

Y podremos comprobar el resultado correcto

Sumar números POST

Número 1

7

Número 2

7

Sumar números

La suma de $7 + 7 = 14$

A continuación, un clásico: **Collatz**

Todo número positivo siempre llegará a ser 1:

- Si el número es Par, lo dividimos entre 2
- Si el número es Impar, lo multiplicamos por 3 y sumamos 1

Sobre **MatematicasController** creamos un nuevo **IActionResult** llamado **ConjeturaCollatz**

MATEMATICASCONTROLLER

```
public IActionResult ConjeturaCollatz()
{
    return View();
}

[HttpPost]
public IActionResult ConjeturaCollatz(int numero)
{
    //DEBEMOS DEVOLVER UN OBJETO COMPLEJO CON
    //UNA LISTA DE NUMEROS
    List<int> numeros = new List<int>();
    while (numero != 1)
    {
        if (numero % 2 == 0)
        {
            numero = numero / 2;
        }
        else
        {
            numero = numero * 3 + 1;
        }
        numeros.Add(numero);
    }
    //DEVOLVEMOS EL MODEL A LA VISTA
    return View(numeros);
}
```

Sobre **Views/Matematicas** creamos una nueva vista llamada **ConjeturaCollatz.cshtml**

CONJETURACOLLATZ.CSHTML

```
@model List<int>
<h2 style="color:blue">
    Conjetura Collatz
</h2>

<form method="post">
    <label>Número Collatz</label>
    <input type="text" name="numero"
        class="form-control"/>
    <button class="btn btn-warning">
        Mostrar conjectura
    </button>
</form>

@if (Model != null){
    <ul>
        @foreach (int num in Model){
            <li>@num</li>
        }
    </ul>
}
```

En **Index.cshtml** incluimos un Link para mostrar los datos de la Vista

```
<li>
    <a asp-controller="Matematicas"
        asp-action="ConjeturaCollatz">
        Conjetura Collatz
    </a>
</li>
```

Y podremos visualizar el resultado

Número Collatz

6

Mostrar conjectura

- 3
- 10
- 5
- 16
- 8
- 4
- 2
- 1

Necesito una vista para mostrar la tabla de multiplicar.
Lo haremos igual que Collatz, devolviendo números con el resultado simplemente.

Tabla multiplicar simple

Número

7

Mostrar tabla

- 7
- 14
- 21
- 28
- 35
- 42
- 49
- 56
- 63
- 70

Creamos un método llamado **TablaMultiplicarSimple** sobre **Matematicas**

MATEMATICASCONTROLLER

```
public IActionResult TablaMultiplicarSimple()
{
    return View();
}

[HttpPost]
public IActionResult TablaMultiplicarSimple(int numero)
{
    List<int> resultados = new List<int>();
    for (int i = 1; i <= 10; i++)
    {
        int operacion = numero * i;
        resultados.Add(operacion);
    }
    return View(resultados);
}
```

Sobre Views/Matemáticas creamos una nueva vista llamada **TablaMultiplicarSimple.cshtml**

TABLAMULTIPLICAR SIMPLE.CSHTML

```
@model List<int>
<h1>Tabla multiplicar simple</h1>
<form method="post">
    <label>Número</label>
    <input type="text" name="numero"
        class="form-control" />
    <button class="btn btn-danger">
        Mostrar tabla
    </button>
</form>
@if (Model != null){
    <ul>
        @foreach (int num in Model){
            <li>@num</li>
        }
    </ul>
}
```

A continuación, vamos a realizar la misma práctica, pero mostrando una tabla, con la Operación y el resultado de la operación

Tabla multiplicar Model

Número

7

Mostrar tabla



Operación	Resultado
7 * 1	7
7 * 2	14
7 * 3	21
7 * 4	28

Vamos a crear un Model para devolver una colección de dicho Model.
Sobre **Models** creamos una nueva clase llamada **FilaTablaMultiplicar**

FILATABLAMULTIPLICAR

```

public class FilaTablaMultiplicar
{
    0 references | 0 changes | 0 authors, 0 changes
    public string Operacion { get; set; }
    0 references | 0 changes | 0 authors, 0 changes
    public int Resultado { get; set; }
}

```

Sobre MatematicasController creamos un nuevo método llamado TablaMultiplicarModel

MATEMATICASCONTROLLER

```

public IActionResult TablaMultiplicarModel()
{
    return View();
}

[HttpPost]
public IActionResult TablaMultiplicarModel(int numero)
{
    List<FilaTablaMultiplicar> filas =
        new List<FilaTablaMultiplicar>();
    for (int i = 1; i <= 10; i++)
    {
        FilaTablaMultiplicar fila =
            new FilaTablaMultiplicar();
        fila.Operacion = numero + " * " + i;
        fila.Resultado = numero * i;
        filas.Add(fila);
    }
    return View(filas);
}

```

Creamos una nueva vista sobre Views/Matematicas llamada TablaMultiplicarModel.cshtml

TABLAMULTPLICARMODEL.CSHTML

```

@model List<FilaTablaMultiplicar>
<h1 style="color:blue">Tabla multiplicar Model</h1>
<form method="post">
    <label>Número</label>
    <input type="text" name="numero"
           class="form-control" />
    <button class="btn btn-outline-warning">
        Mostrar tabla
    </button>
</form>
@if (Model != null){
    <table class="table table-bordered">
        <thead>
            <tr>
                <th>Operación</th>
                <th>Resultado</th>
            </tr>
        </thead>
        <tbody>
            @foreach (FilaTablaMultiplicar fila in Model){
                <tr>
                    <td>@fila.Operacion</td>
                    <td>@fila.Resultado</td>
                </tr>
            }
        </tbody>
    </table>
}

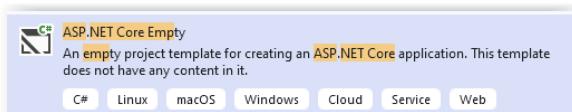
```

FUNCIONAMIENTO DE MVC NET CORE

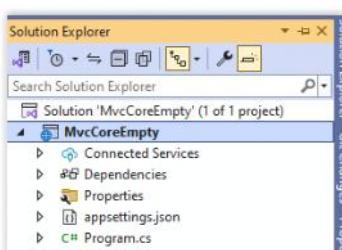
Cuando creamos un proyecto, ya viene todo montado y definido estupendamente, pero la realidad es que debemos saber cómo funcionan los proyectos Mvc Net Core para poder montar todas las piezas.

Vamos a crear un nuevo proyecto, desde cero y lo vamos a ir montando pieza a pieza.

Creamos un nuevo proyecto de tipo **ASP Mvc Net Core Empty** llamado **MvcCoreEmpty**



Como podemos comprobar, está algo vacío



Sobre el proyecto, creamos las carpetas **Models**, **Controllers** y **Views**

Sobre Views creamos una nueva carpeta llamada Shared

Sobre Shared creamos una nueva vista llamada _Layout.cshtml

```
<!doctype html>
<html>
  <head>
    <title>Plantilla principal</title>
  </head>
  <body>
    <h1>Plantilla principal</h1>
    <hr/>
    <div style="background-color: lightgreen">
      <p>ZONA DE VISTAS</p>
    </div>
  </body>
</html>
```

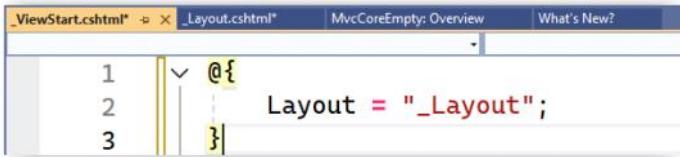
Para que las vistas puedan utilizar esta plantilla tenemos dos formas:

- 1) Indicar en cada vista que plantilla va a utilizar

```
@{
  Layout = "_Layout.cshtml";
}
```

- 2) Indicar, mediante un fichero Global que plantilla por defecto utilizarán todas las vistas.

El nombre de dicho fichero debe llamarse _ViewStart.cshtml y debe estar situado dentro de Views
Creamos dicha vista



```
_ViewStart.cshtml* - P X _Layout.cshtml | MvcCoreEmpty: Overview | What's New? |
1 1  v @{
2 2  |   Layout = "_Layout";
3 3  | }
```

Por supuesto, podemos combinar las dos opciones, es decir, una plantilla general para Todas las vistas y otras plantillas opcionales para algunas vistas

El siguiente paso es crear unas vistas para indicar que utilizarán el Layout

Sobre Views creamos una carpeta llamada Home y una vista llamada Index.cshtml

```
<h1 style="color: fuchsia">
  Estas en casa...
</h1>
```

Sobre la carpeta Controllers agregamos un nuevo controlador llamado HomeController

HOMECONTROLLER

```
public class HomeController : Controller
{
  0 references
  public IActionResult Index()
  {
    return View();
  }
}
```

Con esto ya tenemos todo montado para funcionar.

Lo siguiente es configurar el tipo de aplicación que vamos a utilizar.

Para ello, es necesario configurar una serie de parámetros dentro del Jefe de la App: Program

Es el objeto builder el encargado de incluir los diferentes elementos que conforman una App.

```

var builder = WebApplication.CreateBuilder(args);
//TODO LO QUE INDIQUEMOS SERAN SERVICIOS
//AÑADIMOS LOS SERVICIOS DE VISTAS Y CONTROLLERS
builder.Services.AddControllersWithViews();

var app = builder.Build();

```

El objeto **app** es el encargado de comunicar el servidor con los elementos que utilizaremos en él.

Dicho objeto también **monta** las **Routes** que utilizará nuestra App.

Debemos indicar, mediante un Mapping dónde están nuestros Controller y Vistas
Para ello se utiliza una sintaxis llamada **Pattern**

```
{ controller = CONTROLLER } / { action = VISTA }
```

<https://servidor/Home/Index>

A lo mejor queremos poner otra ruta...

```
{ controller = CONTROLLER } /MisVistas / { action = VISTA }
```

<https://servidor/Home/MisVistas/Index>

También podemos incluir parámetros dentro de las rutas

```

var app = builder.Build();
//DEBEMOS INDICAR QUE UTILIZAREMOS RUTAS PARA
//CONTROLLERS Y VIEWS

//app.MapGet("/", () => "Hello World!");
app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}"
);

app.Run();

```

Por último, al iniciar nuestra App ya podremos comprobar que funciona correctamente.

An unhandled exception occurred while processing the request.

InvalidOperationException: RenderBody has not been called for the page at '/Views/Shared/_Layout.cshtml'. To ignore call IgnoreBody().

Debemos indicar, dentro del **LAYOUT** dónde se dibujarán nuestras Vistas.

Se utiliza una instrucción llamada **RenderBody**

Solamente podemos tener un **RenderBody** en cada Layout

_LAYOUT.CSHTML

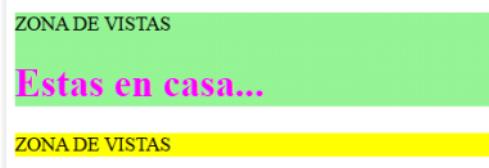
```

<!doctype html>
<html>
    <head>
        <title>Plantilla principal</title>
    </head>
    <body>
        <h1>Plantilla principal</h1>
        <hr/>
        <div style="background-color: lightgreen">
            <p>ZONA DE VISTAS</p>
            @RenderBody()
        </div>
        <div style="background-color: yellow">
            <p>ZONA DE VISTAS</p>
        </div>
    </body>
</html>

```

Como podemos comprobar, ya es funcional

Plantilla principal



El siguiente paso son los fuegos artificiales, vamos a incluir **Bootstrap** en local. Dentro de nuestro proyecto.

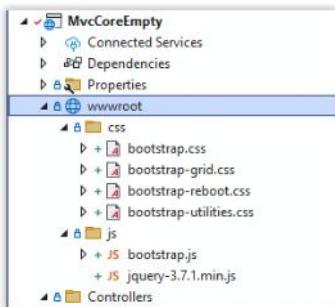
Descargamos Bootstrap y Jquery.

Nombre	Fecha de modificación
css	09/10/2024 9:09
js	09/10/2024 9:09
jquery-3.7.1.min.js	30/01/2025 10:30

Los ficheros locales, JS, CSS o imágenes deben ir dentro de la carpeta **wwwroot** de nuestro Proyecto.

Sobre el proyecto, agregamos una nueva carpeta llamada **wwwroot**

Dentro de dicha carpeta, copiamos las carpetas de css y js.



Buscamos una plantilla SIMPLE en los ejemplos de Bootstrap.

Esta plantilla mismo

<https://getbootstrap.com/docs/5.3/examples/sticky-footer-navbar/>

Copiamos la plantilla y la pegamos dentro de nuestra plantilla **_Layout.cshtml**

Posteriormente, sobre **wwwroot/css** creamos un nuevo css llamado **sticky.css**

De la plantilla **_Layout.cshtml** copiamos el código de **<style>** y lo pegamos en nuestro CSS.

El siguiente paso es poner en LOCAL todos los recursos que utiliza la plantilla

Para indicar en las rutas que estamos accediendo a **wwwroot** es necesario utilizar el símbolo **~ (ALT GR + 4)**

```
<link href="~/css/bootstrap.min.css" rel="stylesheet" />
<link href="~/css/sticky.css" rel="stylesheet" />
```

```
    </div>
</footer>
<script src="~/js/jquery-3.7.1.min.js"></script>
<script src="~/js/bootstrap.bundle.min.js"></script>
```

El siguiente paso es incluir **RenderBody** en cualquier zona de nuestra página

```
<!-- Begin page content -->
<main class="flex-shrink-0">
  <div class="container">
    <h1 class="mt-5">Sticky footer with fixed navbar</h1>
    @RenderBody()
  </div>
</main>
```

Al ejecutar, podremos comprobar que no encuentra ningún recurso local.

```
✖ Failed to load resource: the server responded    bootstrap.min.css:1 ⓘ
with a status of 404 ()
✖ Failed to load resource: the server responded with a color-modes.js:1 ⓘ
status of 404 ()
✖ Failed to load resource: the server responded    jquery-3.7.1.min.js:1 ⓘ
with a status of 404 ()
✖ Failed to load resource: the server      bootstrap.bundle.min.js:1 ⓘ
responded with a status of 404 ()
✖ Failed to load resource: the server responded    bootstrap.min.css:1 ⓘ
with a status of 404 ()
✖ Failed to load resource: the server      sticky-footer-navbar.css:1 ⓘ
responded with a status of 404 ()
✖ Failed to load resource: the server responded with a    sticky.css:1 ⓘ
status of 404 ()
```

Dentro de nuestro servidor debemos indicar TODO lo que vayamos a utilizar.
A esto se le llama Middleware, que son los recursos que debemos establecer en el Servidor donde se ejecute nuestra App.

Tiene un orden y debemos utilizar dicho orden

1. Control de errores y excepciones
2. Protocolo de seguridad de transporte estricta de HTTP
3. Redireccionamiento de HTTPS
4. Servidor de archivos estáticos
5. Cumplimiento de directivas de cookies
6. Autenticación
7. Sesión
8. MVC

Dentro de **Program** debemos indicar que utilizaremos **Static Files**

```
app.UseStaticFiles();

app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}"
);
```

Vamos a intentar utilizar **JQUERY** dentro del **_Layout**

Incluimos un button, le ponemos un ID y hacemos lo que sea desde Jquery

_LAYOUT.CSHTML

```
<div class="container">
    <h1 class="mt-5" id="mensaje">Zona de Vistas</h1>
    <button class="btn btn-danger" id="botonlayout">
        Botón Layout
    </button>
    @RenderBody()
</div>
```

```
<script src="~/js/jquery-3.7.1.min.js"></script>
<script src="~/js/bootstrap.bundle.min.js"></script>
<script>
    $(document).ready(function() {
        $("#botonlayout").click(function() {
            $("#mensaje").text("Botón pulsado en LAYOUT!!!!");
        })
    })
</script>
```

Ejecutamos y debería funcionar

Botón pulsado en LAYOUT!!!

Boton Layout

Estas en casa...

Vamos a realizar lo mismo, pero desde una VISTA (Index).

Necesito un `<h1>` para dibujar el mensaje y un `button` para llamar a Jquery

Al pulsar el botón, cambiamos el mensaje.

Todo en INDEX

INDEX.CSHTML

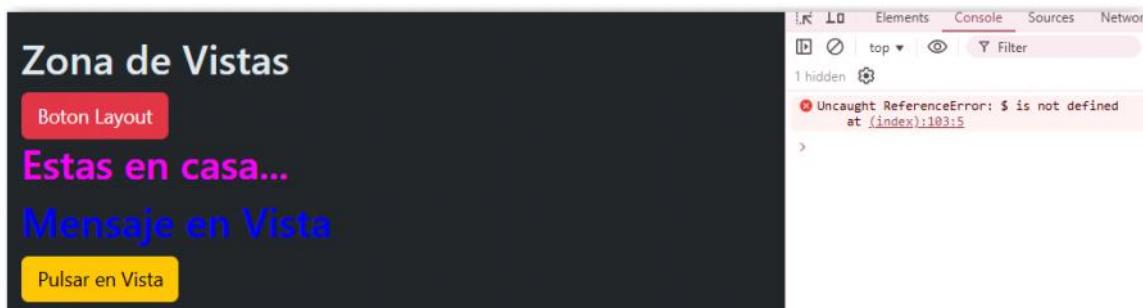
```
<h1 style="color:fuchsia">
    Estas en casa...
</h1>

<h1 style="color:blue" id="mensajejvista">
    Mensaje en Vista
</h1>

<button class="btn btn-warning" id="botonvista">
    Pulsar en Vista
</button>

<script>
    $(document).ready(function() {
        $("#botonvista").click(function() {
            $("#mensajejvista").text("Botón de pulsado en Juernes!!!");
        })
    })
</script>
```

Ejecutamos y comprobamos que NO funciona



Para saber qué sucede, debemos visualizar el código fuente:

```
<script>
    $(document).ready(function() {
        $("#botonvista").click(function() {
            $("#mensajejvista").text("Botón de pulsado en Juernes!!!");
        })
    })
</script>
</div>
</main>

<footer class="footer mt-auto py-3 bg-body-tertiary">
    <div class="container">
        <span class="text-body-secondary">Place sticky footer content here.</span>
    </div>
</footer>
<script src="/js/jquery-3.7.1.min.js"></script>
<script src="/js/bootstrap.bundle.min.js"></script>
```

La solución es poder dibujar el script dentro del Layout y después de la declaración del JS

Tenemos una funcionalidad que nos permite dibujar código de las vistas dentro del Layout en la zona que deseemos, es decir, cualquier código de la vista podemos integrarlo En alguna "zona" del Layout.

Por ejemplo, una estilos propio y único de esa vista.

Esta funcionalidad nos permite crear **Sections** dentro de las plantillas **Layout** para que las Vistas utilicen dichas secciones si lo desean.

Al crear un Section, es obligatorio que las vistas lo utilicen.

La sintaxis: `@RenderSection("NAME")`

_LAYOUT.CSHTML

```

<link href="~/css/sticky-footer-navbar.css" rel="stylesheet">
<*/ CREAMOS UNA SECCION PARA LAS VISTAS PUEDAN INCLUIR SUS
<style> SI LO NECESITAN
*@
@RenderSection("styles")
<meta name="theme-color" content="#712cf9">
</head>
<body class="d-flex flex-column h-100">

```

Al ejecutar, veremos que nos está dando un error.

An unhandled exception occurred while processing the request.

InvalidOperationException: The layout page '/Views/Shared/_Layout.cshtml' cannot find the section 'styles' in the content page '/Views/Home/Index.cshtml'.

Tenemos dos formas de declarar Sections:

1) Obligatoria: Todas las vistas deben utilizar dicho Section

```
@RenderSection("styles")
```

2) Opcional: Solamente algunas vistas utilizarán dicho Section

```
@RenderSection("styles", false)
```

Para que una Vista pueda utilizar un Section se debe escribir la siguiente sintaxis dentro de la Vista:

```
@section NAMESECTION {
    Nuestro código
}
```

Primero vamos a utilizar el Section **styles** y lo probamos

INDEX.CSHTML

```

@section styles {
    <style>
        h2 {
            background-color: yellow
        }
    </style>
}

```

Y ahora lo probamos con una sección que vamos a crear dentro de **_Layout.cshtml**

_LAYOUT.CSHTML

```

<script src="~/js/jquery-3.7.1.min.js"></script>
<script src="~/js/bootstrap.bundle.min.js"></script>
@RenderSection("scripts", false)
<script>

```

Utilizamos dicha ZONA Section dentro de **Index.cshtml**

INDEX.CSHTML

```

@section scripts {
    <script>
        $(document).ready(function() {
            $("#botonvista").click(function() {
                $("#mensajevista").text("Botón de pulsado en Juernes!!!");
            })
        })
    </script>
}

```

A continuación vamos a probar algunas características extra que tienen los Section.

Sobre **wwwroot** creamos una carpeta llamada **images** y pegamos algunas imágenes.



Tendremos una imagen estática para el **Layout**.

Vamos a crear un **RenderSection** llamado **images** para que las Vistas puedan utilizarlo.

_LAYOUT.CSHTML

```
<main class="flex-shrink-0">
    <div class="container">
        <h1 class="mt-5" id="mensaje">Zona de Vistas</h1>
        @RenderSection("images", false)
        
        <button class="btn btn-danger" id="botonlayout">
```

El siguiente paso es crear una nueva vista.

Sobre **Views/Home** creamos una nueva vista llamada **EjemploSection.cshtml**

EJEMPLOSECTION.CSHTML



Sobre **HomeController** creamos un **IActionResult** llamado **EjemploSection**

HOMECONTROLLER

```
public IActionResult EjemploSection ()
{
    return View();
}
```

En **_Layout** incluimos unos Links para mostrar las vistas que tenemos.

_LAYOUT.CSHTML

```
<li class="nav-item">
    @Html.ActionLink("Home", "Index", "Home")
</li>
<li class="nav-item">
    @Html.ActionLink("Sections", "EjemploSection", "Home")
</li>
```

En la Vista **EjemploSection** vamos a añadir un **Section** para la zona **images** del **Layout**

EJEMPLOSECTION.CSHTML

```
<h1 style="color: lightcoral">
    Ejemplo Section
</h1>

@section images {
    
}
```



¿Qué sucede si, en lugar de añadir, necesitamos **SUSTITUIR**?

Quitar un elemento de la página principal y poner otro elemento en su lugar.
Esto se hace con **Section** también, pero tenemos que hacer algo más.

Imaginemos que tenemos un menú para hacer **Login**

Una vez que se ha realizado el Login, queremos quitar ese menú y poner otro.

Para sustituir debemos crear un **Section** condicional dentro del **Layout**

```
@if (IsSectionDefined("NAME SECTION")) {  
    //ALGUNA VISTA ESTA UTILIZANDO EL SECTION  
    //CREAMOS EL SECTION  
    @RenderSection("NAME SECTION")  
}  
else {  
    //HACEMOS EL CODIGO SI NO ESTUVIERA EL SECTION  
}
```

Lo que vamos a realizar es **sustituir** la imagen de Home.
Si alguna página quiere, que cambie la imagen de Home, en el resto de páginas se verá
La imagen Home

_LAYOUT.CSHTML

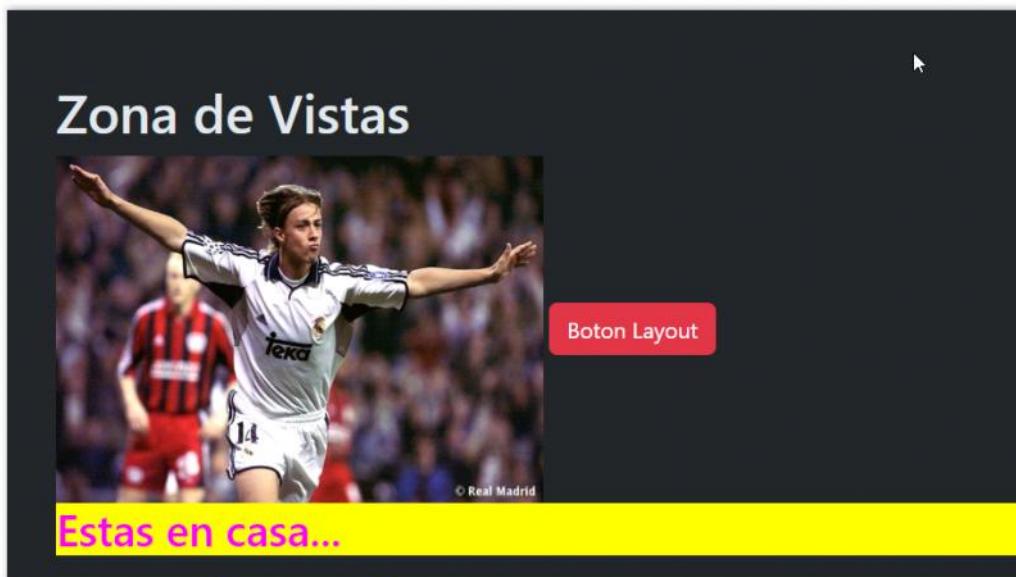
```
@if (IsSectionDefined("Imagenhome")){  
    //ALGUNA VISTA QUIERE SUSTITUIR LA IMAGEN DE HOME  
    @RenderSection("Imagenhome", false)  
}  
else{  
    //CODIGO ESTATICO PARA EL RESTO DE VISTAS  
    //QUE NO QUIEREN SUSTITUIR  
      
}
```

Lo que vamos a realizar es **SUSTITUIR** la imagen de Home desde **Index**

INDEX.CSHTML

```
@section Imagenhome {  
      
}
```

Y comprobaremos que todo es funcional



A continuación, vamos a jugar con **Models**

Sobre la carpeta **Models** creamos una nueva clase llamada **Persona**

PERSONA

```
public class Persona
{
    0 references | 0 changes | 0 authors, 0 changes
    public string Nombre { get; set; }
    0 references | 0 changes | 0 authors, 0 changes
    public string Email { get; set; }
    0 references | 0 changes | 0 authors, 0 changes
    public int Edad { get; set; }
}
```

Sobre **HomeController** creamos un nuevo método llamado **VistaPersona** y que Devolverá nuestro Model para dibujarlo en dicha vista.

HOMECONTROLLER

```
public IActionResult VistaPersona()
{
    Persona persona = new Persona();
    persona.Nombre = "Alumno Core";
    persona.Email = "alumnocore@gmail.com";
    persona.Edad = 25;
    return View(persona);
}
```

Sobre **Views/Home** creamos una nueva vista llamada **VistaPersona.cshtml**

VISTAPERSONA.CSHTML

```
@model Persona

<h1>Vista Persona</h1>

<h2 style="color: red">
    @Model.Nombre, @Model.Email, Edad: @Model.Edad
</h2>
```

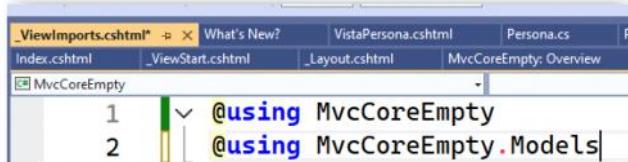
Podremos comprobar que no reconoce Persona

Podemos solucionar esto de dos formas distintas:

- 1) **@using**: Utilizar **@using** para acceder a nuestros Models o lo que sea

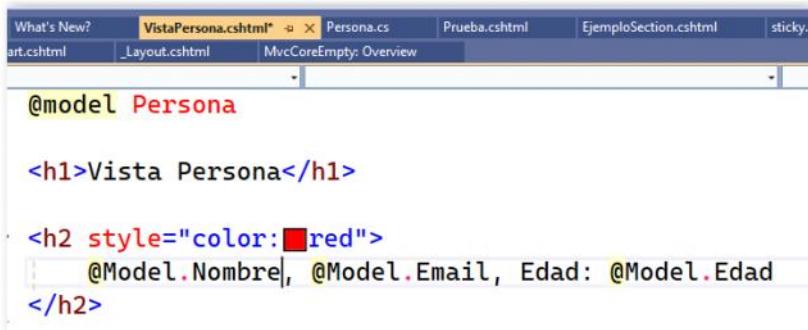
```
@using MvcCoreEmpty.Models  
@model Persona
```

- 2) Utilizar @using GLOBAL para todas las vistas.
Esta opción nos permite incluir varios using y funcionalidades "extra" para
Todas las vistas de nuestra App.
Para ello, debemos crear una vista llamada _ViewImports.cshtml dentro de Views



Ahora ya podemos utilizar el Model Persona directamente desde VistaPersona.cshtml

VISTAPERSONA.CSHTML



El último concepto es poder utilizar Tag Helpers

```
<li class="nav-item">  
  <a class="nav-link active"  
     asp-controller="Home"  
     asp-action="Index">Casita</a>  
</li>
```

Un Tag Helper es una etiqueta **Razor** con código amigable HTML.

Para poder utilizar Tag Helpers en nuestras vistas, debemos habilitar un **Namespace** dentro _ViewImports.cshtml

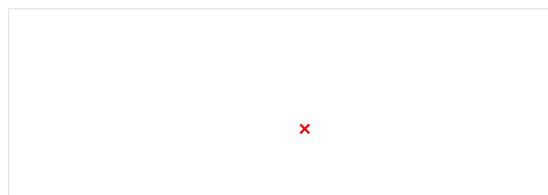
La instrucción es la siguiente:

```
@addTagHelper *, NAMESPACE
```

El namespace es **Microsoft.AspNetCore.Mvc.TagHelpers**

_VIEWIMPORTS.CSHTML

Y ya tendremos habilitado el uso de Tag Helpers en cualquier vista



Por último, el último concepto es cambiar de Template si lo deseamos hacer en
Alguna vista.

Utilizar otro Layout en lugar del definido dentro de _ViewStarts.cshtml

Simplemente, debemos utilizar esta sintaxis en las vistas que deseemos cambiar el
Layout predefinido:

```
@{  
  Layout = "_MiPlantilla";  
}
```

Sobre Views/Shared creamos una nueva vista llamada _MiPlantilla.cshtml

```
<!doctype html>
<html>
  <head>
    <title>Nueva plantilla</title>
  </head>
  <body>
    <h1>Plantilla nueva</h1>
    <div style="background-color: #aqua">
      <h1>Zona de vistas</h1>
      @RenderBody()
    </div>
  </body>
</html>
```

Incluimos, en cualquier página, que sustituya el Layout principal por esta nueva plantilla

```
@{{ Layout = "_MiPlantilla"; }}
@model Persona
<h1>Vista Persona</h1>
```

PRACTICA PROYECTO DESDE CERO

Debemos crear un nuevo proyecto llamado MvcCoreAdoNet

Configuramos una plantilla con Bootstrap y todo lo que vimos el Jueves para que sea funcional

Configuramos nuestro proyecto para SQL Server

Agregamos el siguiente Nuget

Microsoft.Data.SqlClient by Microsoft, nugetsqltools, 762M download 6.0.1
The current data provider for SQL Server and Azure SQL databases. This has replaced System.Data.SqlClient. These classes provide access to SQL and enca...

Vamos a comenzar creando una vista de Hospitales

Hospitales SQL Server

Nombre	Dirección	Teléfono
Provincial	O' Donell 50	964-4256
General	Atocha s/n	595-3111
La Paz	Castellana 1000	923-5411
San Carlos	Ciudad Universitaria	597-1500
Ruber	Juan Bravo, 49	91-4027100

Sobre Models, creamos una nueva clase llamada Hospital

X

Sobre el proyecto, creamos una nueva carpeta llamada **Repositories** y una clase llamada **RepositoryHospital**

REPOSITORYHOSPITAL

```
public class RepositoryHospital
{
    SqlConnection cn;
    SqlCommand com;
    SqlDataReader reader;

    public RepositoryHospital()
    {
        string connectionString = @"Data Source=LOCALHOST
\DESARROLLO;Initial Catalog=HOSPITAL;Persist Security Info=True;User ID=SA;Encrypt=True;Trust Server Certificate=True";
        this.cn = new SqlConnection(connectionString);
        this.com = new SqlCommand();
        this.com.Connection = this.cn;
    }

    public List<Hospital> GetHospitales()
    {
        string sql = "select * from HOSPITAL";
        this.com.CommandType = System.Data.CommandType.Text;
        this.com.CommandText = sql;
        this.cn.Open();
        this.reader = this.com.ExecuteReader();
        List<Hospital> hospitales = new List<Hospital>();
        while (this.reader.Read())
        {
            Hospital hospital = new Hospital();
            hospital.IdHospital =
                int.Parse(this.reader["HOSPITAL_COD"].ToString());
            hospital.Nombre =
                this.reader["NOMBRE"].ToString();
            hospital.Direccion =
                this.reader["DIRECCION"].ToString();
            hospital.Telefono =
                this.reader["TELEFONO"].ToString();
            hospital.Camas =
                int.Parse(this.reader["NUM_CAMA"].ToString());
            hospitales.Add(hospital);
        }
        this.reader.Close();
        this.cn.Close();
        return hospitales;
    }
}
```

Sobre Controllers creamos un nuevo controlador Empty llamado **HospitalesController**

HOSPITALESCONTROLLER

```
public class HospitalesController : Controller
{
    private RepositoryHospital repo;

    public HospitalesController()
    {
        this.repo = new RepositoryHospital();
    }

    public IActionResult Index()
    {
        List<Hospital> hospitales = this.repo.GetHospitales();
        return View(hospitales);
    }
}
```

X

Sobre Views creamos una nueva carpeta llamada **Hospitales** y una vista llamada **Index.cshtml**

INDEX.CSHTML

```
@model List<Hospital>
<h1>Hospitales SQL Server</h1>
<table class="table table-bordered">
<thead>
<tr>
<th>Nombre</th>
<th>Dirección</th>
```

```

        <th>Teléfono</th>
    </tr>
</thead>
<tbody>
    @foreach (Hospital hospital in Model){
        <tr>
            <td>@hospital.Nombre</td>
            <td>@hospital.Direccion</td>
            <td>@hospital.Telefono</td>
        </tr>
    }
</tbody>
</table>

```

Sobre _Layout.cshtml incluimos un link para mostrar los hospitales

_LAYOUT.CSHTML



El siguiente paso es realizar un Detalles para el Hospital que seleccionemos.

Sobre el Repo, agregamos un nuevo método

REPOSITORYHOSPITAL

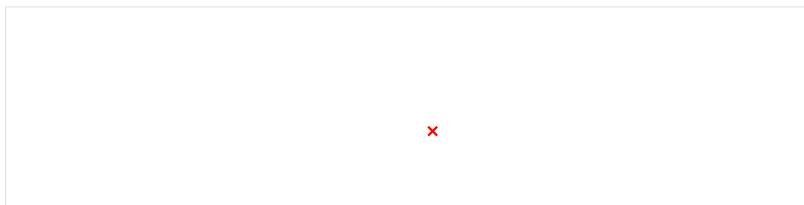
```

public Hospital FindHospital(int idhospital)
{
    string sql =
        "select * from HOSPITAL where HOSPITAL_COD=@idhospital";
    this.com.Parameters.AddWithValue("@idhospital", idhospital);
    this.com.CommandType = System.Data.CommandType.Text;
    this.com.CommandText = sql;
    this.cn.Open();
    this.reader = this.com.ExecuteReader();
    Hospital hospital = new Hospital();
    this.reader.Read();
    hospital.IdHospital =
        int.Parse(this.reader["HOSPITAL_COD"].ToString());
    hospital.Nombre = this.reader["NOMBRE"].ToString();
    hospital.Direccion = this.reader["DIRECCION"].ToString();
    hospital.Telefono = this.reader["TELEFONO"].ToString();
    hospital.Camas =
        int.Parse(this.reader["NUM_CAMA"].ToString());
    this.reader.Close();
    this.cn.Close();
    this.com.Parameters.Clear();
    return hospital;
}

```

Agregamos un nuevo método IActionResult llamado Details sobre HospitalesController

HOSPITALESCONTROLLER



El siguiente paso es crear una vista llamada Details.cshtml sobre Views/Hospitales

DETAILS.CSHTML

```

@model Hospital


# Details


    Back to List


- Id: @Model.IdHospital
- Nombre: @Model.Nombre
- Dirección: @Model.Direccion
- Teléfono: @Model.Telefono
- Camas: @Model.Camas

```

Dentro de Index.cshtml de Views/Hospitales agregamos un Link para cada uno de los Hospitales y mostrar sus detalles

INDEX.CSHTML



Podemos visualizar que nos está ofreciendo la siguiente Ruta:



Dicha ruta, si lo deseamos, podemos indicar que sea distinta, todo funciona igual, pero Las rutas dentro de MVC tienen la siguiente estructura:

<https://localhost/Controller/Action/PARAM>

Debemos indicar, dentro de **Program** que podemos recibir parámetros en las Rutas:

PROGRAM



El siguiente paso es realizar un INSERT de Hospital



Creamos un nuevo método dentro de **RepositoryHospital**

REPOSITORYHOSPITAL

```
public void CreateHospital
    (int idHospital, string nombre, string direccion
     , string telefono, int camas)
{
    string sql = "insert into HOSPITAL values (@idhospital, @nombre "
        + ", @direccion, @telefono, @camas)";
    this.com.Parameters.AddWithValue("@idhospital", idHospital);
    this.com.Parameters.AddWithValue("@nombre", nombre);
    this.com.Parameters.AddWithValue("@direccion", direccion);
    this.com.Parameters.AddWithValue("@telefono", telefono);
    this.com.Parameters.AddWithValue("@camas", camas);
    this.com.CommandType = System.Data.CommandType.Text;
    this.com.CommandText = sql;
    this.com.Open();
    this.com.ExecuteNonQuery();
    this.com.Close();
    this.com.Parameters.Clear();
}
```

El siguiente paso es crear dos métodos dentro de **HospitalesController**

HOSPITALESCONTROLLER

```

public IActionResult Create()
{
    return View();
}

[HttpPost]
public IActionResult Create(Hospital hospital)
{
    this.repo.CreateHospital(hospital.IdHospital, hospital.Nombre,
        hospital.Direccion, hospital.Telefono, hospital.Camas);
    ViewData["MENSAJE"] = "Hospital insertado";
    return View();
}

```

Sobre Views/Hospitales creamos una nueva vista llamada **Create.cshtml**

CREATE.CSHTML

```

<h1 style="color:blue">Create</h1>

<form method="post">
    <label>Id Hospital</label>
    <input type="text" name="idhospital"
        class="form-control"/>
    <label>Nombre: </label>
    <input type="text" name="nombre"
        class="form-control"/>
    <label>Dirección </label>
    <input type="text" name="direccion"
        class="form-control"/>
    <label>Teléfono </label>
    <input type="text" name="telefono" class="form-control"/>
    <label>Camas </label>
    <input type="text" name="camas" class="form-control"/>
    <button class="btn btn-info">Create</button>
</form>
<h2 style="color:red">@ViewData["MENSAJE"]</h2>

```

El siguiente paso será crear un método para Modificar un Hospital.

Comenzamos creando el método dentro de **RepositoryHospital**

REPOSITORYHOSPITAL

```

public void UpdateHospital(int idHospital,
    string nombre, string direccion, string telefono, int camas)
{
    string sql = "update HOSPITAL set NOMBRE=@nombre "
        + ", DIRECCION=@direccion, TELEFONO=@telefono "
        + ", NUM_CAMA=@camas "
        + " where HOSPITAL_COD=@idhospital";
    this.com.Parameters.AddWithValue("@nombre", nombre);
    this.com.Parameters.AddWithValue("@direccion", direccion);
    this.com.Parameters.AddWithValue("@telefono", telefono);
    this.com.Parameters.AddWithValue("@camas", camas);
    this.com.Parameters.AddWithValue("@idhospital", idHospital);
    this.com.CommandType = System.Data.CommandType.Text;
    this.com.CommandText = sql;
    this.cn.Open();
    this.com.ExecuteNonQuery();
    this.cn.Close();
    this.com.Parameters.Clear();
}

```

Sobre **HospitalesController** creamos dos nuevos métodos

HOSPITALESCONTROLLER

```

public IActionResult Update(int id)
{
    Hospital hospital = this.repo.FindHospital(id);
    return View(hospital);
}

[HttpPost]
public IActionResult Update(Hospital hospital)
{
    this.repo.UpdateHospital(hospital.IdHospital
        , hospital.Nombre, hospital.Direccion
        , hospital.Telefono, hospital.Camas);
    ViewData["MENSAJE"] = "Hospital modificado";
    return View(hospital);
}

```

Sobre Views/Hospitales creamos una nueva vista llamada **Update.cshtml**

UPDATE.CSHTML

```

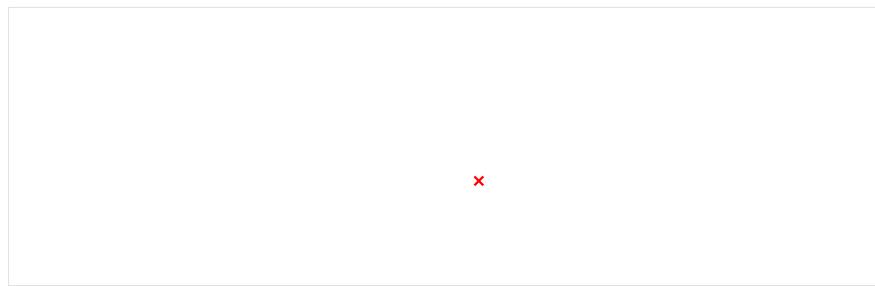
@model Hospital
<h1 style="color:red">
    Update
</h1>

<form method="post">
    <input type="hidden" name="idhospital" value="@Model.IdHospital"/>
    <label>Nombre </label>
    <input type="text" name="nombre" class="form-control"
        value="@Model.Nombre"/>
    <label>Dirección</label>
    <input type="text" name="direccion" class="form-control"
        value="@Model.Direccion"/>
    <label>Teléfono</label>
    <input type="text" name="telefono" class="form-control"
        value="@Model.Telefono"/>
    <label>Camas</label>
    <input type="text" name="camas" class="form-control"
        value="@Model.Camas"/>
    <button class="btn btn-warning">Update</button>
</form>
<h2 style="color:blue">@ViewData["MENSAJE"]</h2>

```

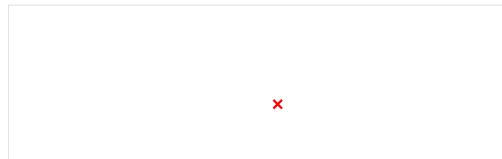
Sobre la vista **Index.cshtml** dentro de Views/Hospitales incluimos un Link enviando el ID del Hospital

INDEX.CSHTML

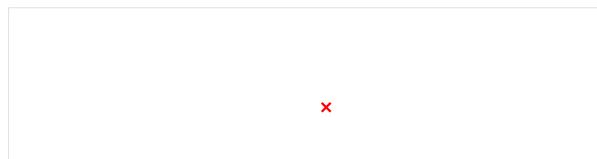


Tenemos tres formas posibles de trabajar con la devolución de las vistas desde los Controllers

- 1) Navegar desde un controller a la propia vista



- 2) Navegar a otra vista desde el Controller: Esta funcionalidad **solo**mente nos lleva a una Vista, sin pasar por su controller.
Se puede utilizar para Diseño, llevar a una vista de error genérico.



- 3) Navegar desde un IActionResult a otro IActionResult. Envía desde un IActionResult a otro código IActionResult dentro del Controller.



Por último, vamos a realizar **Eliminar**

Lo que haremos será un método **Delete** que recibirá un ID.

Dicho método **Delete** no tendrá View. Solamente creamos un View para Delete si Necesito algún tipo de confirmación.

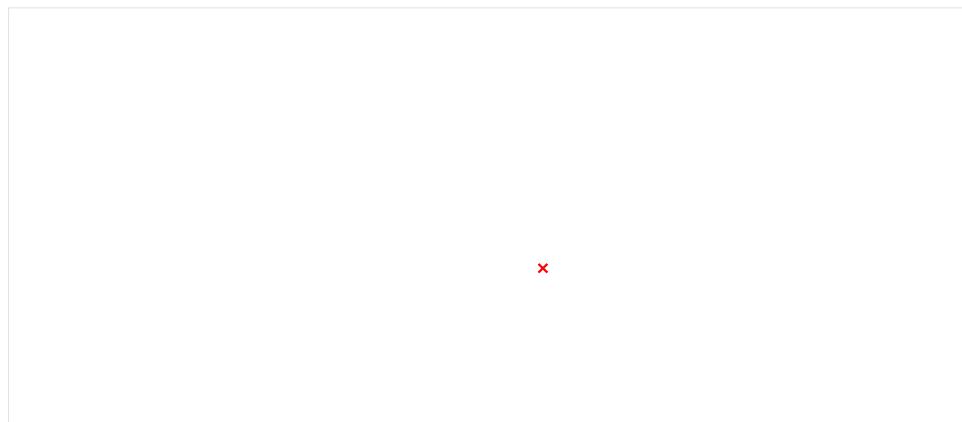
Comenzamos creando un nuevo método dentro de **RepositoryHospital**

REPOSITORYHOSPITAL

```
public void DeleteHospital(int idHospital)
{
    string sql = "delete from HOSPITAL where HOSPITAL_COD=@idhospital";
    this.com.Parameters.AddWithValue("@idhospital", idHospital);
    this.com.CommandType = System.Data.CommandType.Text;
    this.com.CommandText = sql;
    this.cn.Open();
    this.com.ExecuteNonQuery();
    this.cn.Close();
    this.com.Parameters.Clear();
}
```

Sobre **HospitalesController** creamos un nuevo método **Delete**

HOSPITALESCONTROLLER



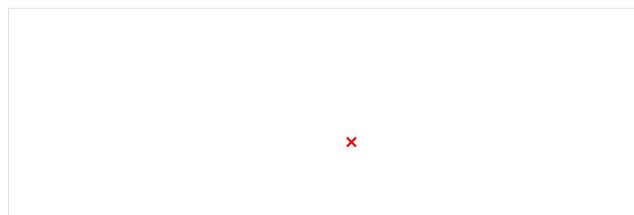
Por último, en la vista **Index.cshtml** incluimos un Link enviando el Id del hospital a Eliminar

INDEX.CSHTML

Vamos a visualizar cómo podemos enviar información mediante Tag Helpers.
Como hemos visto, tenemos la posibilidad de enviar información con
@Html.ActionLink y ahora lo haremos con Tag Helper

Simplemente utilizamos **asp-route-PARAM** para poder enviar la información
Con un GET

Modificamos los Links de **INDEX.CSHTML**



Vamos a realizar una práctica en la que buscaremos los Doctores
Por Especialidad.
Al iniciar la página, mostraremos todos los doctores.
Tendremos la posibilidad de buscar doctores mediante un input.

Doctores por Especialidad

Introduzca especialidad:

Buscar doctores

Id doctor	Apellido	Especialidad	Salario	Hospital
435	López A.	Cardiología	322643	19
982	Cajal R.	Cardiología	152863	18

Sobre Models creamos una nueva clase llamada Doctor

DOCTOR

```
public class Doctor
{
    public int IdDoctor { get; set; }
    public string Apellido { get; set; }
    public string Especialidad { get; set; }
    public int Salario { get; set; }
    public int IdHospital { get; set; }
}
```

Sobre Repositories creamos una nueva clase llamada RepositoryDoctor

REPOSITORYDOCTOR

```
public class RepositoryDoctor
{
    SqlConnection cn;
    SqlCommand com;
    SqlDataReader reader;

    public RepositoryDoctor()
    {
        string connectionString = @"Data Source=LOCALHOST
\DESARROLLO;Initial Catalog=HOSPITAL;Persist Security Info=True;User ID=SA;Encrypt=True;Trust Server Certificate=True";
        this.cn = new SqlConnection(connectionString);
        this.com = new SqlCommand();
        this.com.Connection = this.cn;
    }

    public async Task<List<Doctor>> GetDoctoresAsync()
    {
        string sql = "select * from DOCTOR";
        this.com.CommandType = System.Data.CommandType.Text;
        this.com.CommandText = sql;
        await this.cn.OpenAsync();
        this.reader = await this.com.ExecuteReaderAsync();
        List<Doctor> doctores = new List<Doctor>();
        while (await this.reader.ReadAsync()) {
            Doctor doctor = new Doctor();
            doctor.IdDoctor = (int) this.reader["IdDoctor"];
            doctor.Apellido = this.reader["Apellido"].ToString();
            doctor.Especialidad = this.reader["Especialidad"].ToString();
            doctor.Salario = (int) this.reader["Salario"];
            doctor.IdHospital = (int) this.reader["IdHospital"];
            doctores.Add(doctor);
        }
        return doctores;
    }
}
```

```

        Doctor doc = new Doctor
    {
        IdDoctor = int.Parse(this.reader["DOCTOR_NO"].ToString()),
        Apellido = this.reader["APELIDO"].ToString(),
        Especialidad = this.reader["ESPECIALIDAD"].ToString(),
        Salario = int.Parse(this.reader["SALARIO"].ToString()),
        IdHospital =
            int.Parse(this.reader["HOSPITAL_COD"].ToString())
    };
    doctores.Add(doc);
}
await this.reader.CloseAsync();
await this.cn.CloseAsync();
return doctores;
}

public async Task<List<Doctor>>
GetDoctoresEspecialidadAsync(string especialidad)
{
    string sql =
        "select * from DOCTOR where ESPECIALIDAD=@especialidad";
    this.com.Parameters.AddWithValue("@especialidad", especialidad);
    this.com.CommandType = System.Data.CommandType.Text;
    this.com.CommandText = sql;
    await this.cn.OpenAsync();
    this.reader = await this.com.ExecuteReaderAsync();
    List<Doctor> doctores = new List<Doctor>();
    while (await this.reader.ReadAsync())
    {
        Doctor doc = new Doctor
        {
            IdDoctor = int.Parse(this.reader["DOCTOR_NO"].ToString()),
            Apellido = this.reader["APELIDO"].ToString(),
            Especialidad = this.reader["ESPECIALIDAD"].ToString(),
            Salario = int.Parse(this.reader["SALARIO"].ToString()),
            IdHospital = int.Parse(this.reader["HOSPITAL_COD"].ToString())
        };
        doctores.Add(doc);
    }
    await this.reader.CloseAsync();
    await this.cn.CloseAsync();
    this.com.Parameters.Clear();
    return doctores;
}
}

```

Sobre Controllers creamos un nuevo controlador llamado DoctoresController

DOCTORESCONTROLLER

```

public class DoctoresController : Controller
{
    RepositoryDoctor repo;

    public DoctoresController()
    {
        this.repo = new RepositoryDoctor();
    }

    public async Task<IActionResult> DoctoresEspecialidad()
    {
        List<Doctor> doctores = await this.repo.GetDoctoresAsync();
        return View(doctores);
    }

    [HttpPost]
    public async Task<IActionResult>
    DoctoresEspecialidad(string especialidad)
    {
        List<Doctor> doctores =
            await this.repo.GetDoctoresEspecialidadAsync(especialidad);
        return View(doctores);
    }
}

```

Sobre Views creamos una carpeta llamada **Doctores** y una vista llamada **DoctoresEspecialidad.cshtml**

DOCTORESESPACIALIDAD.CSHTML

```

@model List<Doctor>

<h1>Doctores por Especialidad</h1>

<form method="post">
    <label>Introduzca especialidad: </label>
    <input type="text" class="form-control"
           name="especialidad"/>
    <button class="btn btn-light">
        Buscar doctores
    </button>
</form>

<table class="table table-active">
    <thead>
        <tr>
            <th>Id doctor</th>
            <th>Apellido</th>
            <th>Especialidad</th>
            <th>Salario</th>
            <th>Hospital</th>
        </tr>
    </thead>
    <tbody>
        @foreach (Doctor doctor in Model){
            <tr>
                <td>@doctor.IdDoctor</td>
                <td>@doctor.Apellido</td>
                <td>@doctor.Especialidad</td>
                <td>@doctor.Salario</td>
                <td>@doctor.IdHospital</td>
            </tr>
        }
    </tbody>
</table>

```

ENVIAR MULTIPLES OBJETOS A LAS VISTAS

Como todos sabemos, solamente podemos enviar un Model a cada vista.
Si necesitásemos enviar más objetos complejos, no podemos mediante Model.

Por ejemplo, en la práctica anterior, hemos escrito el nombre de la especialidad, pero
También podríamos mejorar la experiencia del usuario, mostrando una opción para
Seleccionar la Especialidad.

En este planteamiento, necesitaríamos dos elementos en la vista:

- 1) List<string> que corresponda a la Especialidad
- 2) List<Doctor> que son los propios doctores

Podríamos crear una clase para enviar estos dos datos.

Nosotros vamos a utilizar **ViewData/ViewBag**.
Podemos perfectamente utilizar estos objetos para enviar datos complejos pero,
Al no ser nativo, necesitamos realizar algo más en las vistas.

Doctores por Especialidad

Seleccione especialidad:

Count	Name	Especialidad	Count	Count
435	López A.	Cardiología	322643	19
453	Galo D.	Pediatría	147060	22
522	Adams C.	Neurología	521119	45

Comenzamos agregando un método para devolver las Especialidades dentro de **RepositoryDoctor**

REPOSITORYDOCTOR

```
public async Task<List<string>> GetEspecialidadesAsync()
{
    string sql = "select distinct ESPECIALIDAD from DOCTOR";
    this.com.CommandType = System.Data.CommandType.Text;
    this.com.CommandText = sql;
    await this.cn.OpenAsync();
    this.reader = await this.com.ExecuteReaderAsync();
    List<string> especialidades = new List<string>();
    while (await this.reader.ReadAsync())
    {
        especialidades.Add(this.reader["ESPECIALIDAD"].ToString());
    }
    await this.reader.CloseAsync();
    await this.cn.CloseAsync();
    return especialidades;
}
```

Dentro del GET del método **IActionResult** del **Controller** debemos enviar dos datos:

- 1) Especialidades: **ViewData**
- 2) Doctores: **Model**

Por norma, el Model suele ser lo más importante de la página, es decir, en lo que se basa la Propia Vista.

DOCTORESCONTROLLER

```
public async Task<IActionResult> DoctoresEspecialidad()
{
    List<Doctor> doctores = await this.repo.GetDoctoresAsync();
    List<string> especialidades =
        await this.repo.GetEspecialidadesAsync();
    ViewData["ESPECIALIDADES"] = especialidades;
    return View(doctores);
}
```

A continuación, dibujamos sobre la vista **DoctoresEspecialidad.cshtml** un form recibiendo Los datos mediante **ViewData**

DOCTORESESPACIALIDAD.CSHTML

Lo primero que podemos comprobar es que NO reconoce lo que tenemos almacenado dentro de **ViewData**

X

Debemos indicar el tipo de objeto que existe dentro de ViewData mediante código **Razor**

```
@{  
    int numero = (int) ViewData["NUMERO"];  
    Persona persona = (Persona) ViewData["PERSONA"];  
}
```

X

Y recorremos el objeto declarado en Razor

X

Al seleccionar una especialidad y buscar los doctores, estamos teniendo una excepción

X

Debemos dibujar las especialidades tanto en el GET como en el POST, una vez que hemos Filtrado los doctores

X

Vamos a realizar un nuevo proyecto llamado **MvcCoreCrudDepartamentos** y Aplicaremos todo lo que hemos aprendido hasta ahora.

x

Sobre el proyecto, agregamos el siguiente **Nuget**

x

Sobre **Models** creamos una nueva clase llamada **Departamento**

DEPARTAMENTO

x

Para insertar, vamos a crear un nuevo procedimiento para insertar un departamento.

Dentro del Procedure, recuperamos el Max Id departamento.

x

Creamos una nueva carpeta llamada **Repositories** y una clase llamada **RepositoryDepartamentos**

REPOSITORYDEPARTAMENTOS

```
public class RepositoryDepartamentos
{
    SqlCommand com;
    SqlConnection cn;
    SqlDataReader reader;

    public RepositoryDepartamentos()
    {
        string connectionString = @"Data Source=LOCALHOST
\DESARROLLO;Initial Catalog=HOSPITAL;Persist Security Info=True;User ID=SA;Encrypt=True;Trust Server Certificate=True";
        this.cn = new SqlConnection(connectionString);
        this.com = new SqlCommand();
        this.com.Connection = this.cn;
    }

    public async Task<List<Departamento>> GetDepartamentosAsync()
    {
        string sql = "select * from DEPT";
        this.com.CommandType = CommandType.Text;
        this.com.CommandText = sql;
        await this.cn.OpenAsync();
        ...
    }
}
```

```

        this.reader = await this.com.ExecuteReaderAsync();
        List<Departamento> departamentos = new List<Departamento>();
        while (await this.reader.ReadAsync())
        {
            Departamento departamento = new Departamento();
            departamento.IdDepartamento =
                int.Parse(this.reader["DEPT_NO"].ToString());
            departamento.Nombre =
                this.reader["DNOMBRE"].ToString();
            departamento.Localidad =
                this.reader["LOC"].ToString();
            departamentos.Add(departamento);
        }
        await this.reader.CloseAsync();
        await this.cn.CloseAsync();
        return departamentos;
    }
}

```

Sobre Controllers creamos un nuevo controlador llamado **DepartamentosController**

DEPARTAMENTOSCONTROLLER

```

public class DepartamentosController : Controller
{
    RepositoryDepartamentos repo;
    public DepartamentosController()
    {
        this.repo = new RepositoryDepartamentos();
    }
    public async Task<IActionResult> Index()
    {
        List<Departamento> departamentos = await
            this.repo.GetDepartamentosAsync();
        return View(departamentos);
    }
}

```

Dentro de Views creamos una carpeta llamada **Departamentos** y una vista llamada **Index.cshtml**

INDEX.CSHTML

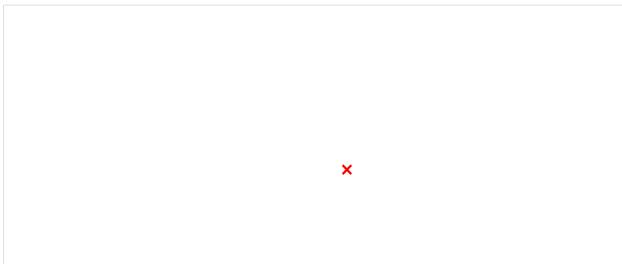
```

@model List<Departamento>
<h1>Departamentos SQL</h1>
<table class="table table-warning">
    <thead>
        <tr>
            <th>Id</th>
            <th>Nombre</th>
            <th>Localidad</th>
        </tr>
    </thead>
    <tbody>
        @foreach (Departamento dept in Model){
            <tr>
                <td>@dept.IdDepartamento</td>
                <td>@dept.Nombre</td>
                <td>@dept.Localidad</td>
            </tr>
        }
    </tbody>
</table>

```

Incluimos un Link dentro de **_Layout.cshtml**

_LAYOUT.CSHTML



El siguiente paso que vamos a realizar es crear departamentos.

Utilizaremos el procedimiento almacenado.

Creamos un nuevo método dentro de **RepositoryDepartamentos**

REPOSITORYDEPARTAMENTOS

```

public async Task InsertDepartmentAsync
    (string nombre, string localidad)
{
    string sql = "SP_INSERT_DEPARTAMENT";
    this.com.Parameters.AddWithValue("@nombre", nombre);
    this.com.Parameters.AddWithValue("@localidad", localidad);
    this.com.CommandType = CommandType.StoredProcedure;
    this.com.CommandText = sql;
    await this.cn.OpenAsync();
    await this.com.ExecuteNonQueryAsync();
    await this.cn.CloseAsync();
    this.com.Parameters.Clear();
}

```

Agregamos un nuevo método dentro de **DepartamentosController**

DEPARTAMENTOSCONTROLLER

```

public IActionResult Create()
{
}

```

```

        return View();
    }

    [HttpPost]
    public async Task<IActionResult>
        Create(string nombre, string localidad)
    {
        await this.repo.InsertDepartmentAsync(nombre, localidad);
        return RedirectToAction("Index");
    }

```

Creamos una nueva vista dentro de Views/Departamentos llamada Create.cshtml

CREATE.CSHTML

```

<h1>Create</h1>
<p>
    <a asp-controller="Departamentos"
        asp-action="Index">
        Back to list
    </a>
</p>
<form method="post">
    <label>Departamento: </label>
    <input type="text" name="nombre" class="form-control"/>
    <label>Localidad: </label>
    <input type="text" name="localidad" class="form-control"/>
    <button class="btn btn-info">
        Insertar
    </button>
</form>

```

El siguiente paso es realizar un Update del departamento.
Para modificar necesitamos poder Buscar el Departamento a modificar.

Creamos dos nuevos métodos dentro del **Repository**

REPOSITORYDEPARTAMENTOS

```

public async Task<Departamento>
    FindDepartamentoAsync(int idDepartamento)
{
    string sql = "select * from DEPT where DEPT_NO=@iddepartamento";
    this.com.Parameters.AddWithValue("@iddepartamento", idDepartamento);
    this.com.CommandType = CommandType.Text;
    this.com.CommandText = sql;
    await this.cn.OpenAsync();
    this.reader = await this.com.ExecuteReaderAsync();
    Departamento departamento = new Departamento();
    await this.reader.ReadAsync();
    departamento.IdDepartamento =
        int.Parse(this.reader["DEPT_NO"].ToString());
    departamento.Nombre = this.reader["DNOMBRE"].ToString();
    departamento.Localidad = this.reader["LOC"].ToString();
    await this.reader.CloseAsync();
    await this.cn.CloseAsync();
    this.com.Parameters.Clear();
    return departamento;
}

public async Task UpdateDepartamentoAsync
    (int idDepartamento, string nombre, string localidad)
{
    string sql = "update DEPT set DNOMBRE=@nombre "
        + ", LOC=@localidad "
        + " where DEPT_NO=@iddepartamento";
    this.com.Parameters.AddWithValue("@iddepartamento", idDepartamento);
    this.com.Parameters.AddWithValue("@nombre", nombre);
    this.com.Parameters.AddWithValue("@localidad", localidad);
    this.com.CommandType = CommandType.Text;
    this.com.CommandText = sql;
    await this.cn.OpenAsync();
    await this.com.ExecuteNonQueryAsync();
    await this.cn.CloseAsync();
    this.com.Parameters.Clear();
}

```

Agregamos dos nuevos métodos dentro de **DepartamentosController**

DEPARTAMENTOSCONTROLLER

```

public async Task<IActionResult> Edit(int id)
{
    Departamento departamento =
        await this.repo.FindDepartamentoAsync(id);
    return View(departamento);
}

[HttpPost]
public async Task<IActionResult> Edit(Departamento departamento)
{
    await this.repo.UpdateDepartamentoAsync
        (departamento.IdDepartamento, departamento.Nombre
        , departamento.Localidad);
    return RedirectToAction("Index");
}

```

Creamos una vista llamada **Edit.cshtml** dentro de Views/Departamentos

EDIT.CSHTML

```

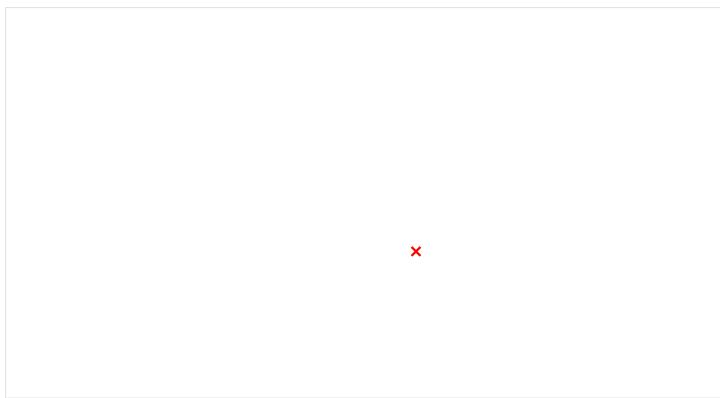
@model Departamento
<h1>Edit</h1>
<p>
    <a asp-controller="Departamentos"
        asp-action="Index">Back to list</a>
</p>
<form method="post">
    <input type="hidden" name="iddepartamento"
        value="@Model.IdDepartamento" />
    <label>Departamento: </label>
    <input type="text" name="nombre" value="@Model.Nombre"
        class="form-control"/>
    <input type="text" name="localidad" value="@Model.Localidad"
        class="form-control"/>
    <button class="btn btn-warning">

```

```
Update  
</button>  
</form>
```

Incluimos la acción dentro de **Index.cshtml** de **Views/Departamentos** para cada departamento

INDEX.CSHTML



El siguiente paso es realizar **Eliminar**.

Para eliminar no quiero confirmación, simplemente que lo elimine y listo.

Vamos a crear un método para eliminar un Departamento por su ID

REPOSITORYDEPARTAMENTOS

```
public async Task DeleteDepartamentoAsync(int idDepartamento)  
{  
    string sql = "delete from DEPT where DEPT_NO=@iddepartamento";  
    this.com.Parameters.AddWithValue("@iddepartamento", idDepartamento);  
    this.com.CommandType = CommandType.Text;  
    this.com.CommandText = sql;  
    await this.cn.OpenAsync();  
    await this.com.ExecuteNonQueryAsync();  
    await this.cn.CloseAsync();  
    this.com.Parameters.Clear();  
}
```

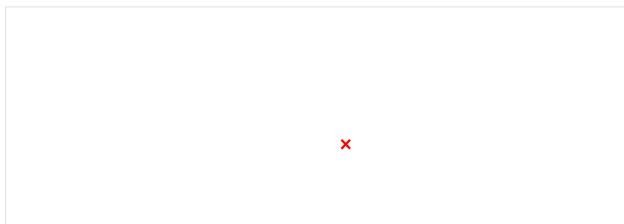
Creamos un nuevo método dentro de **DepartamentosController**

DEPARTAMENTOSCONTROLLER

```
public async Task<IActionResult> Delete(int id)  
{  
    await this.repo.DeleteDepartamentoAsync(id);  
    return RedirectToAction("Index");  
}
```

Incluimos un nuevo Link para realizar la acción de Eliminar dentro de **Index.cshtml**

INDEX.CSHTML



CONSULTAS DINAMICAS LINQ

Linq es un lenguaje propio dentro de C#.

Es un lenguaje de acceso a objetos/colecciones.

Nos permite, mediante una sintaxis única, poder acceder a diferentes conjuntos independientemente del tipo de conjunto.

Podemos acceder a un SQL, XML o JSON de la misma forma.

Se utilizan expresiones **LAMBDA** para poder ejecutar determinadas consultas.

Este lenguaje sustituirá a las consultas tradicionales de SQL de **selección**.

Para las consultas se utilizan genéricos que son una maravilla.

Un genérico es un tipo de dato que el compilador es capaz de reconocer y se representa mediante la letra **<T>**.

En realidad es un object, pero un object reconocible y mutable.

Para utilizar genéricos se utiliza la palabra **var**

```
Object objeto = "Hola mundo";  
objeto.ToUpper() // ESTO NOS GENERA UN ERROR, NO RECONOCE LO QUE TIENE ALMACENADO
```

Ejemplo utilizando var:

```
var objeto = "Hola mundo";  
objeto.ToUpper() //LO RECONOCE
```

```
var objeto = List<int>;  
objeto.Add(88) //LO RECONOCE
```

Vamos a aprender a utilizar consultas Linq.

Las consultas se escriben al revés.

Utilizan un ALIAS para poder acceder al conjunto de objetos y sus propiedades.

Linq es un lenguaje C#, por lo que utiliza los operadores de C# y también es case sensitive

```
from ALIAS in CONJUNTO  
select ALIAS
```

Podemos realizar filtros sobre el conjunto

```
from ALIAS in CONJUNTO  
where ALIAS.Propiedad == valor  
select ALIAS
```

La última instrucción **select** es lo que nos devolverá el conjunto.

Por ejemplo, pongamos que estamos trabajando con la tabla EMP.

Si queremos todos los datos de EMP: select * from EMP

```
from datos in EMP  
select datos
```

Estamos devolviendo todos los datos. También podemos recuperar el Apellido
select APELLIDO from EMP

```
from datos in EMP  
select datos.APELLIDO
```

Una consulta Linq debe devolver tipos reconocibles:

Por ejemplo, pongamos que deseamos el APELLIDO y el OFICIO

```
select APELLIDO, OFICIO from EMP
```

```
from datos in EMP  
select datos.APELLIDO, datos.OFICIO
```

Cuando devolvemos datos NO reconocibles (string,string) debemos utilizar **Objetos anónimos**

Un objeto anónimo es un objeto que creamos dinámicamente dentro del código C#

```
@Html.ActionLink("Link", "Action", "Controller"  
, new { id = 1, dato = "Pulsado" })
```

Traducido a Linq

```
from datos in EMP  
select new { datos.APELLIDO, datos.OFICIO }
```

```
from datos in EMP  
select new { apellido = datos.APELLIDO, funcion = datos.OFICIO }
```

Linq siempre devolverá un conjunto.

Lo que tenemos dentro del conjunto es:

```
Consulta[0].apellido  
Consulta[0].funcion
```

Vamos a comenzar utilizando **Linq To SQL**

Consultas de selección: **LINQ**

Consultas de acción: **SQL**

Vamos a trabajar con dos nuevos objetos de acceso a datos:

- **SqlDataAdapter**: Es el encargado de recuperar los datos y es un conjunto de Command, Connection de forma transparente.
- **DataTable**: Es un objeto C# que nos permite almacenar Tablas dentro de nuestro código.

Ejemplo de nuestro nuevo código, accediendo a todos los departamentos.

```
string sql = "select * from DEPT";  
SqlDataAdapter adDept = new SqlDataAdapter(connectionString, sql);  
DataTable tablaDept = new DataTable();  
adDept.Fill(tablaDept);  
//A PARTIR DE AQUÍ, LINQ CON FILTROS O LO QUE DESEEMOS
```

Creamos un nuevo proyecto llamado **MvcCoreLinqToSQL**

x

Agregamos el Nuget de SqlClient

x

Sobre **Models** creamos una nueva clase llamada **Empleado**

x

Creamos una carpeta llamada **Repositories** y una clase llamada **RepositoryEmpleados**

REPOSITORYEMPLEADOS

```
public class RepositoryEmpleados
{
    private DataTable tablaEmpleados;

    public RepositoryEmpleados()
    {
        string connectionString = @"Data Source=LOCALHOST
\DESARROLLO;Initial Catalog=HOSPITAL;Persist Security Info=True;User ID=SA;Enc
rypt=True;Trust Server Certificate=True";
        string sql = "select * from EMP";
        SqlDataAdapter adEmp = new SqlDataAdapter(sql, connectionString);
        this.tablaEmpleados = new DataTable();
        //RECUPERAMOS LOS DATOS
        adEmp.Fill(this.tablaEmpleados);
    }

    //METODO PARA RECUPERAR TODOS LOS EMPLEADOS
    public List<Empleado> GetEmpleados()
    {
        //LAS CONSULTAS LINQ SE ALMACENAN EN GENERICOS (var)
        var consulta = from datos in this.tablaEmpleados.AsEnumerable()
                      select datos;
        //AHORA MISMO TENEMOS DENTRO DE CONSULTA LA INFORMACION DE
        //LA TABLA EMPLEADOS
        //EN ESTE EJEMPLO TENEMOS OBJETO DataRow QUE SON FILAS
        //DENTRO DE LA TABLA
        //DEBEMOS RECORRER DICHAS FILAS Y EXTRAER LA INFORMACION
        //EN OBJETOS DE TIPO Empleado
        List<Empleado> empleados = new List<Empleado>();
        //RECORREMOS CADA FILA DE LA CONSULTA
        foreach (var row in consulta)
        {
            Empleado emp = new Empleado();
            //PARA EXTRAER DATOS DE UN DataRow
            //DataRow.Field<tipo>("COLUMNA")
            emp.IdEmpleado = row.Field<int>("EMP_NO");
            emp.Apellido = row.Field<string>("APELLIDO");
            emp.Oficio = row.Field<string>("OFICIO");
        }
    }
}
```

```

        emp.Salario = row.Field<int>("SALARIO");
        emp.IdDepartamento = row.Field<int>("DEPT_NO");
        empleados.Add(emp);
    }
    return empleados;
}

//METODO PARA BUSCAR EMPLEADOS POR SU ID
public Empleado FindEmpleado(int idEmpleado)
{
    var consulta = from datos in this.tablaEmpleados.AsEnumerable()
                  where datos.Field<int>("EMP_NO") == idEmpleado
                  select datos;
    //NOSOTROS SABEMOS QUE ESTA CONSULTA DEVUELVE UNA FILA
    //LINQ SIEMPRE DEVUELVE UNA COLECCION
    //DENTRO DEL CONJUNTO TENEMOS METODOS LAMBDA QUE NOS
    //PERMITEN REALIZAR COSITAS
    //TENEMOS UN METODO QUE NOS DEVUELVE EL PRIMER VALOR DEL CONJUNTO
    //First();
    var row = consulta.First();
    Empleado emp = new Empleado();
    emp.IdEmpleado = row.Field<int>("EMP_NO");
    emp.Apellido = row.Field<string>("APELLIDO");
    emp.Oficio = row.Field<string>("OFICIO");
    emp.Salario = row.Field<int>("SALARIO");
    emp.IdDepartamento = row.Field<int>("DEPT_NO");
    return emp;
}
}

```

Sobre Controllers creamos un nuevo controlador llamado **EmpleadosController**

EMPLEADOSCONTROLLER

```

public class EmpleadosController : Controller
{
    RepositoryEmpleados repo;
    public EmpleadosController()
    {
        this.repo = new RepositoryEmpleados();
    }
    public IActionResult Index()
    {
        List<Empleado> empleados = this.repo.GetEmpleados();
        return View(empleados);
    }
    public IActionResult Details(int idempleado)
    {
        Empleado empleado = this.repo.FindEmpleado(idempleado);
        return View(empleado);
    }
}

```

Creamos una carpeta llamada dentro de **Views** llamada **Empleados** y una vista llamada **Index.cshtml**

INDEX.CSHTML

```

@model List<Empleado>
<h1>Linq Empleados</h1>
<table class="table table-bordered">
    <thead>
        <tr>
            <th>Apellido</th>
            <th>Oficio</th>
            <th>Salario</th>
            <th></th>
        </tr>
    </thead>
    <tbody>
        @foreach (Empleado empleado in Model){
            <tr>
                <td>@empleado.Apellido</td>
                <td>@empleado.Oficio</td>
                <td>@empleado.Salario</td>
                <td>
                    <a asp-controller="Empleados"
                        asp-action="Details"
                        asp-route-idempleado="@empleado.IdEmpleado"
                        class="btn btn-info">
                        Details
                    </a>
                </td>
            </tr>
        }
    </tbody>
</table>

```

Sobre Views/Empleados creamos una nueva vista llamada **Details.cshtml**

DETAILS.CSHTML

```

@model Empleado
<h1>Details</h1>
<p>
    <a asp-controller="Empleados"
        asp-action="Index">
        Back to List
    </a>
</p>
<ul class="list-group">
    <li class="list-group-item">
        @Model.Apellido
    </li>
    <li class="list-group-item">
        @Model.Oficio
    </li>
    <li class="list-group-item">
        @Model.Salario
    </li>
    <li class="list-group-item">
        @Model.IdDepartamento
    </li>
</ul>

```

```
</li>
</ul>
```

A continuación, vamos a realizar una nueva vista en la que filtraremos por Oficio y Salario.

Buscador Linq To SQL

Oficio:

ANALISTA

Salario:

200000

Buscar empleados

Apellido	Oficio	Salario
GIL	ANALISTA	390001
GUTIERREZ	ANALISTA	219001

REPOSITORYEMPLEADOS

```
public List<Empleado> GetEmpleadosOficioSalario
    (String oficio, int salario)
{
    var consulta = from datos in this.tablaEmpleados.AsEnumerable()
        where datos.Field<string>("OFICIO") == oficio
        && datos.Field<int>("SALARIO") >= salario
        select datos;
    List<Empleado> empleados = new List<Empleado>();
    foreach (var row in consulta)
    {
        Empleado empleado = new Empleado
        {
            IdEmpleado = row.Field<int>("EMP_NO"),
            Apellido = row.Field<string>("APELLIDO"),
            Oficio = row.Field<string>("OFICIO"),
            Salario = row.Field<int>("SALARIO"),
            IdDepartamento = row.Field<int>("DEPT_NO")
        };
        empleados.Add(empleado);
    }
    return empleados;
}
```

Agregamos dos nuevos métodos dentro de **EmpleadosController**

EMPLEADOSCONTROLLER

```
public IActionResult BuscadorEmpleados()
{
    return View();
}

[HttpPost]
public IActionResult
    BuscadorEmpleados(string oficio, int salario)
{
    List<Empleado> empleados = this.repo
        .GetEmpleadosOficioSalario(oficio, salario);
    return View(empleados);
}
```

Sobre Views/Empleados creamos una nueva vista llamada **BuscadorEmpleados.cshtml**

BUSCADOREMPLEADOS.CSHMTL

```
@model List<Empleado>


# Buscador Linq To SQL


<form method="post">
    <label>Oficio: </label>
    <input type="text" name="oficio" class="form-control"/>
    <label>Salario: </label>
    <input type="text" name="salario" class="form-control"/>
    <button class="btn btn-info">
        Buscar empleados
    </button>
</form>
@if (Model != null){
    <table class="table table-bordered table-active">
        <thead>
            <tr>
                <th>Apellido</th>
                <th>Oficio</th>
            </tr>
        </thead>
        <tbody>
            <tr>
                <td>GIL</td>
                <td>ANALISTA</td>
            </tr>
            <tr>
                <td>GUTIERREZ</td>
                <td>ANALISTA</td>
            </tr>
        </tbody>
    </table>
}
```

```

        <th>Salario</th>
    </tr>
</thead>
<tbody>
    @foreach (Empleado empleado in Model){
        <tr>
            <td>@empleado.Apellido</td>
            <td>@empleado.Oficio</td>
            <td>@empleado.Salario</td>
        </tr>
    }
</tbody>
</table>
}

```

Por norma, si no encuentra datos, debemos devolver siempre **NULL** dentro de un **Repository**

Vamos a modificar el código del Repo de forma que devuelva NULL en el caso de no encontrar Datos. Para ello, debemos utilizar el método **Lambda** llamado **Count()** de consulta

REPOSITORYEMPLEADOS

```

public List<Empleado> GetEmpleadosOficioSalario
    (string oficio, int salario)
{
    var consulta = from datos in this.tablaEmpleados.Enumerable()
                   where datos.Field<string>("OFICIO") == oficio
                   && datos.Field<int>("SALARIO") >= salario
                   select datos;
    //DEBEMOS COMPROBAR SI TENEMOS DATOS O NO...
    if (consulta.Count() == 0)
    {
        return null;
    }
    else
    {
        List<Empleado> empleados = new List<Empleado>();
        foreach (var row in consulta)
        {
            Empleado empleado = new Empleado
            {
                IdEmpleado = row.Field<int>("EMP_NO"),
                Apellido = row.Field<string>("APELLIDO"),
                Oficio = row.Field<string>("OFICIO"),
                Salario = row.Field<int>("SALARIO"),
                IdDepartamento = row.Field<int>("DEPT_NO")
            };
            empleados.Add(empleado);
        }
        return empleados;
    }
}

```

Este código nos permite, desde nuestro Controller saber si tenemos datos o no y actuar en Consecuencia.

EMPLEADOSCONTROLLER

```

[HttpPost]
public IActionResult BuscadorEmpleados(string oficio, int salario)
{
    List<Empleado> empleados = this.repo
        .GetEmpleadosOficioSalario(oficio, salario);
    if (empleados == null)
    {
        ViewData["MENSAJE"] = "No existen empleados con oficio "
            + oficio + " y salario superior a " + salario;
        return View();
    }
    else
    {
        return View(empleados);
    }
}

```

Modificamos la vista y dibujamos ViewData

```

<h2 style="color:red">
    @ViewData["MENSAJE"]
</h2>

@if (Model != null){
    <table class="table table-bordered table-active">
        <thead>

```

Buscador Linq To SQL

Oficio:

DIRECTORRRR

Salario:

200000

Buscar empleados

No existen empleados con oficio DIRECTORRRR y salario superior a 250000

PRACTICA

Realizaremos una aplicación para visualizar **Enfermo**

Tendremos una vista donde mostraremos todos los enfermos.
Podremos visualizar los detalles de un Enfermo seleccionado.
En la vista Index podremos eliminar un enfermo.

Normas:

- Consultas Selección: LINQ
- Consultas de acción: ADO NET

Fernández M.	Details	Delete
Serrano V.	Details	Delete
Cervantes M.	Details	Delete

Details

[Back to list](#)

Apellido: Fernández M.

Inscripción: 14024

Fecha nacimiento: sábado, 21 de mayo de 1960

Dirección: Recoletos 50

Sobre Models creamos una nueva clase llamada **Enfermo**

```
public class Enfermo
{
    5 references
    public string Inscripcion { get; set; }
    4 references
    public string Apellido { get; set; }
    3 references
    public string Direccion { get; set; }
    3 references
    public DateTime FechaNacimiento { get; set; }
}
```

Sobre Repositories creamos una nueva clase llamada **RepositoryEnfermos**

```
REPOSITORYENFERMOS

public class RepositoryEnfermos
{
    private DataTable tablaEnfermos;
    private SqlConnection cn;
    private SqlCommand com;

    public RepositoryEnfermos()
    {
        string connectionString = @"Data Source=LOCALHOST
\DESARROLLO;Initial Catalog=HOSPITAL;Persist Security Info=True;User ID=SA;Encrypt=True;Trust Server Certificate=True";
        this.tablaEnfermos = new DataTable();
        string sql = "select * from ENFERMO";
        SqlDataAdapter ad = new SqlDataAdapter(sql, connectionString);
        ad.Fill(this.tablaEnfermos);
        this.cn = new SqlConnection(connectionString);
        this.com = new SqlCommand();
        this.com.Connection = this.cn;
    }

    public List<Enfermo> GetEnfermos()
    {
        ...
    }
}
```

```

var consulta = from datos in this.tablaEnfermos.AsEnumerable()
    select datos;
List<Enfermo> enfermos = new List<Enfermo>();
foreach (var row in consulta)
{
    Enfermo enfermo = new Enfermo
    {
        Inscripcion = row.Field<string>("INSCRIPCION"),
        Apellido = row.Field<string>("APELLIDO"),
        Direccion = row.Field<string>("DIRECCION"),
        FechaNacimiento = row.Field<DateTime>("FECHA_NAC")
    };
    enfermos.Add(enfermo);
}
return enfermos;
}

public Enfermo FindEnfermo(string inscripcion)
{
    var consulta = from datos in this.tablaEnfermos.AsEnumerable()
        where datos.Field<string>("INSCRIPCION") == inscripcion
        select datos;
    if (consulta.Count() == 0)
    {
        return null;
    }
    else
    {
        var row = consulta.First();
        Enfermo enfermo = new Enfermo
        {
            Inscripcion = row.Field<string>("INSCRIPCION"),
            Apellido = row.Field<string>("APELLIDO"),
            Direccion = row.Field<string>("DIRECCION"),
            FechaNacimiento = row.Field<DateTime>("FECHA_NAC")
        };
        return enfermo;
    }
}

public void DeleteEnfermo(string inscripcion)
{
    string sql = "delete from ENFERMO where INSCRIPCION=@inscripcion";
    this.com.Parameters.AddWithValue("@inscripcion", inscripcion);
    this.com.CommandType = CommandType.Text;
    this.com.CommandText = sql;
    this.cn.Open();
    this.com.ExecuteNonQuery();
    this.cn.Close();
    this.com.Parameters.Clear();
}
}

```

Sobre Controllers creamos un nuevo controlador llamado **EnfermosController**

ENFERMOSCONTROLLER

```

public class EnfermosController : Controller
{
    private RepositoryEnfermos repo;

    public EnfermosController()
    {
        this.repo = new RepositoryEnfermos();
    }

    public IActionResult Index()
    {
        List<Enfermo> enfermos = this.repo.GetEnfermos();
        return View(enfermos);
    }

    public IActionResult Details(string inscripcion)
    {
        Enfermo enfermo = this.repo.FindEnfermo(inscripcion);
        return View(enfermo);
    }

    public IActionResult Delete(string inscripcion)
    {
        this.repo.DeleteEnfermo(inscripcion);
        return RedirectToAction("Index");
    }
}

```

Sobre Views creamos una carpeta llamada **Enfermos** y una vista llamada **Index.cshtml**

INDEX.CSHTML

```

@model List<Enfermo>
<h1>Enfermos Linq To Sql</h1>
<ul class="list-group">
    @foreach (Enfermo enfermo in Model){
        <li class="list-group-item">
            @enfermo.Apellido
            <a href="#" asp-controller="Enfermos"
                asp-action="Details"
                asp-route-inscripcion="@enfermo.Inscripcion"
                class="btn btn-dark">
                Details
            </a>
            <a href="#" asp-controller="Enfermos"
                asp-action="Delete"
                asp-route-inscripcion="@enfermo.Inscripcion"
                class="btn btn-danger">
                Delete
            </a>
        </li>
    }
</ul>

```

Sobre Views/Enfermos creamos una nueva vista llamada **Details.cshtml**

DETAILS.CSHTML

```
@model Enfermo
```

```

<h1 style="color:blue">Details</h1>
<p>
  <a asp-controller="Enfermos"
     asp-action="Index">Back to list</a>
</p>
<ul class="list-group">
  <li class="list-group-item list-group-item-primary">
    Apellido: @Model.Apellido
  </li>
  <li class="list-group-item list-group-item-danger">
    Inscripción: @Model.Inscripcion
  </li>
  <li class="list-group-item list-group-item-info">
    Fecha nacimiento: @Model.FechaNacimiento.ToString()
  </li>
  <li class="list-group-item list-group-item-success">
    Dirección: @Model.Direccion
  </li>
</ul>

```

EXPRESIONES LAMBDA

Una expresión Lambda se utiliza para poder obtener información de un conjunto.

Nos puede permitir filtrar, ordenar, buscar o recuperar información exacta de un grupo Determinado.

La expresión Lambda se representa mediante =>

Dentro de la expresión se genera una variable para acceder a cada elemento del conjunto Y poder utilizar dicha variable para nuestra expresión

```

List<int> numeros = [2,4,2,6,7,78,8];
numeros.Sum(x => x)
numeros.Max(z => z)

```

```

List<Persona> personas= {[Nombre, Edad], {Nombre, Edad}, {Nombre, Edad}};
personas.Sum(x => x.Edad)
personas.Max(x => x.Edad)
personas.OrderBy(z => z.Nombre)

```

Vamos a realizar una aplicación en la que mostraremos los empleados por un Determinado Oficio.

Tendremos un método que nos devolverá los diferentes oficios de los Empleados Además de los empleados, mostraremos las Personas, el Máximo y el mínimo

Lambda expression Linq

Oficio:

Buscar empleados

Personas: 2

Máximo salario: 390001

Media salarial: 304501

Apellido	Oficio	Salario
GUTIERREZ	ANALISTA	219001
GIL	ANALISTA	390001

Comenzaremos primero con una caja y luego hacemos lo de un desplegable para los Oficios

Sobre **Models** creamos un nuevo modelo para el resumen de los empleados y de sus Datos del conjunto. Creamos una clase llamada **ResumenEmpleados**

```

public class ResumenEmpleados
{
  0 references | 0 changes | 0 authors, 0 changes
  public int Personas { get; set; }
  0 references | 0 changes | 0 authors, 0 changes
  public int MaximoSalario { get; set; }
  0 references | 0 changes | 0 authors, 0 changes
  public double MediaSalarial { get; set; }
  0 references | 0 changes | 0 authors, 0 changes
  public List<Empleado> Empleados { get; set; }
}

```

Agregamos un nuevo método dentro de **RepositoryEmpleados**

```

REPOSITORYEMPLEADOS
public ResumenEmpleados GetEmpleadosOficio(string oficio)
{

```

```

var consulta = from datos in this.tablaEmpleados.AsEnumerable()
    where datos.Field<string>("OFICIO") == oficio
    select datos;
//QUISEIRA ORDENAR LOS EMPLEADOS POR SU SALARIO
consulta = consulta.OrderBy(z => z.Field<int>("SALARIO"));
int personas = consulta.Count();
int maximo = consulta.Max(x => x.Field<int>("SALARIO"));
double media = consulta.Average(x => x.Field<int>("SALARIO"));
List<Empleado> empleados = new List<Empleado>();
foreach (var row in consulta)
{
    Empleado empleado = new Empleado
    {
        IdEmpleado = row.Field<int>("EMP_NO"),
        Apellido = row.Field<string>("APELLIDO"),
        Oficio = row.Field<string>("OFICIO"),
        Salario = row.Field<int>("SALARIO"),
        IdDepartamento = row.Field<int>("DEPT_NO")
    };
    empleados.Add(empleado);
}
ResumenEmpleados resumen = new ResumenEmpleados();
resumen.Personas = personas;
resumen.MaximoSalario = maximo;
resumen.MediaSalarial = media;
resumen.Empleados = empleados;
return resumen;
}

```

El siguiente paso es ir al controller de empleados y agregar un par de métodos

EMPLEADOSCONTROLLER

```

public IActionResult EmpleadosOficio()
{
    return View();
}

[HttpPost]
0 references | 0 changes | 0 authors, 0 changes
public IActionResult EmpleadosOficio(string oficio)
{
    ResumenEmpleados resumen = this.repo.GetEmpleadosOficio(oficio);
    return View(resumen);
}

```

Sobre Views/Empleados agregamos una nueva vista llamada `EmpleadosOficio.cshtml`

EMPLEADOSOFICIO.CSHTML

```

@model ResumenEmpleados


# Lambda expression Linq


<form method="post">
    <label>Oficio: </label>
    <input type="text" name="oficio"
           class="form-control"/>
    <button class="btn btn-info">
        Buscar empleados
    </button>
</form>
@if (Model != null){
    <ul class="list-group">
        <li class="list-group-item">
            Personas: @Model.Personas
        </li>
        <li class="list-group-item">
            Máximo salario: @Model.MaximoSalario
        </li>
        <li class="list-group-item">
            Media salarial: @Model.MediaSalarial
        </li>
    </ul>
    <table class="table table-striped">
        <thead>
            <tr>
                <th>Apellido</th>
                <th>Oficio</th>
                <th>Salario</th>
            </tr>
        </thead>
        <tbody>
            @foreach (Empleado empleado in Model.Empleados){
                <tr>
                    <td>@empleado.Apellido</td>
                    <td>@empleado.Oficio</td>
                    <td>@empleado.Salario</td>
                </tr>
            }
        </tbody>
    </table>
}

```

El siguiente paso es implementar un desplegable con los diferentes oficios.

Lambda expression Linq

Oficio:

VENDEDOR

Buscar empleados

Personas: 6

Máximo salario: 208000

Media salarial: 172733,3333333334

Apellido	Oficio	Salario
FORD	VENDEDOR	129000
SALA	VENDEDOR	156500

Vamos a utilizar un `Distinct()` con Lambda y Linq

Tenemos un método llamado `Distinct()` para poder filtrar los datos y dicho método

Lo podemos realizar de dos formas.

- 1) Utilizando Lambda una vez que tenemos la consulta

```
var consulta = from datos in....;
var filtro = consulta.Distinct(z => z.Propiedad)
```

- 2) Podemos filtrar directamente en la propia consulta.

```
var consulta = (from datos in ... ).Distinct()
```

Agregamos un nuevo método dentro de `RepositoryEmpleados`

REPOSITORYEMPLEADOS

```
public List<string> GetOficios()
{
    var consulta = (from datos in this.tablaEmpleados.AsEnumerable()
                  select datos.Field<string>("OFICIO")).Distinct();
    return consulta.ToList();
}
```

Implementamos la nueva funcionalidad dentro del Controller

EMPLEADOSCONTROLLER

```
public IActionResult EmpleadosOficio()
{
    ViewData["OFICIOS"] = this.repo.GetOficios();
    return View();
}

[HttpPost]
0 references | 0 changes | 0 authors, 0 changes
public IActionResult EmpleadosOficio(string oficio)
{
    ViewData["OFICIOS"] = this.repo.GetOficios();
    ResumenEmpleados resumen = this.repo.GetEmpleadosOficio(oficio);
    return View(resumen);
}
```

Dibujamos un desplegable en lugar del input dentro de `EmpleadosOficio.cshtml`

EMPLEADOSOFICIO.CSHMTL

```
@model ResumenEmpleados
@{
    List<string> oficios =
        ViewData["OFICIOS"] as List<string>;
}


# Lambda expression Linq


<form method="post">
    <label>Oficio: </label>
    <select name="oficio" class="form-control">
        @foreach (string ofi in oficios){
            <option value="@ofi">@ofi</option>
        }
    </select>
    <button class="btn btn-info">
```

```

        Buscar empleados
    </button>
</form>

@if (Model != null){
    <ul class="list-group">
        <li class="list-group-item">
            Personas: @Model.Personas
        </li>
        <li class="list-group-item">
            Máximo salario: @Model.MaximoSalario
        </li>
        <li class="list-group-item">
            Media salarial: @Model.MediaSalarial
        </li>
    </ul>
    <table class="table table-striped">
        <thead>
            <tr>
                <th>Apellido</th>
                <th>Oficio</th>
                <th>Salario</th>
            </tr>
        </thead>
        <tbody>
            @foreach (Empleado empleado in Model.Empleados){
                <tr>
                    <td>@empleado.Apellido</td>
                    <td>@empleado.Oficio</td>
                    <td>@empleado.Salario</td>
                </tr>
            }
        </tbody>
    </table>
}

```

PRACTICA FINAL

Necesito un CRUD de Doctores aplicando todo lo que hemos visto.

Necesito un buscador de doctores por especialidad con un desplegable.