Visual Studio Community

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

SQL Server Express

**Net Core Lenguaje C#**

Windows Forms

Fundamentos del Lenguaje. Clases, POO, Colecciones.

Inyección de dependencias

ADO NET 🡪 Linq y Entity Framework

Esta tecnología nos permite crear proyectos multiplataforma.

Misma app para Linux, Windows, MacOS.

Todo esta listo para su producción, es decir, las librerías y lo que realicemos se puede ejecutar SIN EL PROGRAMA VISUAL.

Existen proyectos de librerías y proyectos visuales. Las librerías son paquetes llamados Nuget. Dichos Nuget ya traen todo el contenido para trabajar con la plataforma deseada.

Posteriormente existen proyectos visuales como Windows Forms o MVC Net Core o Maui.

Metodología:

Tendremos un solo proyecto y múltiples formularios para trabajar.

Voy a trabajar con Github y mi código estará ahí para vosotros.

Nuestro IDE contiene lo siguiente:

App Principal: Pantalla Windows Forms.

Solution Explorer: El lugar donde vamos a localizar las clases.

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Toolbox: Barra de herramientas para nuestros componentes.

Comenzamos creando un nuevo proyecto Windows Forms llamado **Fundamentos2024**.

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

La clase Program es la que indica el inicio de nuestra App, es decir, indica qué formulario iniciará en nuestra App.

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

Todo son CLASES en Net y cada clase está compuesta por métodos y propiedades.

Un método es una acción.

Una propiedad es una característica.

Todo objeto debe ser instanciado antes de ser utilizado.

El lenguaje es Case Sensitive, diferencia entre mayúsculas y minúsculas.

Coche car = new Coche();

car.Arrancar(); **Método**

car.Color = Blue; **Propiedad**

Dentro del lenguaje, tenemos la ayuda que es **Intellisense.**

Las Propiedades son llaves inglesas y los métodos son como cubos morados.ç

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente Interfaz de usuario gráfica, Aplicación, Tabla

Descripción generada automáticamente

**Norma:** SOLAMENTE ESCRIBIMOS DENTRO DEL CODIGO DE LAS LLAVES

Comenzamos con controles gráficos. Necesitamos la barra de herramientas (toolbox)

Dentro de dicha barra, es arrastrar y soltar.

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Para poder modificar o trabajar con los controles necesitamos la ventana **Propiedades (F4).** Nos permite cambiar características de los objetos.

Interfaz de usuario gráfica

Descripción generada automáticamente

Cuando hablamos de objetos gráficos, debemos cambiar su **NAME** si interactuamos con dicho control en código.

Por ejemplo: Un botón, lo llamaremos como sea su descripción con un acrónimo (BTN)

En este entorno se escribe la primera palabra en minúsculas y la siguiente letra de la siguiente palabra en mayúscula.

**Name: btnPulsar**

Diagrama

Descripción generada automáticamente Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

Para escribir código, debemos “pensar” en CUANDO. Este tipo de Apps están enfocadas a Eventos. Por ejemplo, cuando pulsamos un botón, qué queremos hacer?

Para recuperar los eventos, doble click sobre el control que deseamos utilizar.

Texto

Descripción generada automáticamente

Para ejecutar nuestra App, simplemente pulsamos F5 o el botón del Play

Texto

Descripción generada automáticamente

Control de ERRORES:

Errores de compilación: Si tenemos un error en nuestro código, siempre pulsamos en **NO**

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Cuando digamos que NO, nos aparecerá una ventana con todos nuestros errores del proyecto.

Un proyecto es un CONJUNTO, si tenemos algún error en el conjunto, debemos solucionarlo, no podemos ejecutar ni compilar si existen errores en cualquier parte del proyecto.

Si pulsamos doble click en cualquier error, nos lleva al lugar del código de error.

Interfaz de usuario gráfica, Aplicación, Word

Descripción generada automáticamente

Tipos de Propiedades en objetos

1. New Object. Aunque no sepamos cómo es la creación, todos son iguales.



1. Propiedades con tipos primitivos o wrapper: Un primitivo o wrapper es un tipo de dato que no necesita ser creado (**new)**. Un Texto o un número



1. Propiedades Enumeradas: Son “herramientas” del lenguaje que nos permiten poder indicar de forma gráfica una propiedad. Truco: **AMARILLO**

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

**GITHUB**

Subir nuestros proyectos a Github.

En la parte inferior derecha de nuestro programa…

Interfaz de usuario gráfica, Aplicación, Teams

Descripción generada automáticamente

Interfaz de usuario gráfica, Aplicación, Correo electrónico

Descripción generada automáticamente

github.com/serraguti/Fundamentos2024

Cuando realicemos cambios en nuestro repo, simplemente veremos los cambios realizados en la parte inferior derecha

Interfaz de usuario gráfica, Aplicación, Teams

Descripción generada automáticamente

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

Descripción generada automáticamente

Utilizar el código de Github (Paco) en mi ordenador…

Abrimos Visual Studio y seleccionamos Clonar un repositorio.

Interfaz de usuario gráfica, Texto, Aplicación, Chat o mensaje de texto

Descripción generada automáticamente

Una vez abierto el proyecto, simplemente, haciendo **Pull** recuperamos los cambios que se vayan realizando en clase.

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

Descripción generada automáticamente

**TIPOS PRIMITIVOS O WRAPPER**

Un tipo Wrapper es un tipo de dato que no necesita ser instanciado (new), simplemente con la igualdad, nos basta para poder crear el objeto.

Declaración de variables:

**string texto = “Hola mundo”;**

**int numero = 14;**

TIPOS:

char: Representa un único carácter y se instancia con comillas simples.

byte: Enteros positivos y negativos hasta 255 máximo.

short: Entero con precisión hasta 27.000

int: Número entero de precisión media.

long: Número entero de precisión alta.

float, double, decimal: Son valores decimales de menor a mayor.

bool: True/False

DateTime: Indica una fecha y hora.

object: Cualquier objeto dentro de visual studio. Es la clase base.

**CONVERSION DE DATOS**

Dentro de la programación, necesitamos almacenar los tipos de dato en variables de su tipo. A veces, los tipos de dato no vienen en el formato que necesitamos y debemos convertirlos, por ejemplo, si preguntamos a alguien por un número, no lo dice en formato texto y nosotros debemos pasarlo a formato número para poder operar con él.

**Conversiones automáticas:**

Se utiliza cuando el tipo de dato a almacenar es **menor** que la variable que estamos utilizando. No HABLAMOS DEL VALOR, sino del tipo.

Texto

Descripción generada automáticamente con confianza media

**Casting de String a Primitivo**

Almacenar cualquier texto en cualquier primitivo.

Se utiliza un método llamado **Parse** que se llama desde el tipo de dato del Primitivo,

**TipoDato.Parse(string)**

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

**Convertir cualquier objeto a String**

Todas las clases contienen un método llamado **ToString()** para poder ser convertidas a su representación Text.

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

**Convertir entre primitivos compatibles**

Se realiza la conversión entre tipos de dato compatibles entre sí, por ejemplo, dos números.

Se utiliza la sintaxis: (TIPO DATO A CONVERTIR)valor

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Vamos a crear un nuevo formulario para sumar dos números.

Tendremos dos cajas y un botón y un Label para mostrar el resultado.

Creamos un nuevo Form llamado **Form01SumarNumeros**

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

Interfaz de usuario gráfica, Aplicación, Word

Descripción generada automáticamente con confianza media

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

**CODIGO FORMULARIO**

private void btnSumarNumeros\_Click(object sender, EventArgs e)

{

//CAPTURAMOS EN VARIABLES LOS NUMEROS DE LAS CAJAS

//EN LA CAJA VIENE UN TIPO DE DATO string

int numero1 = int.Parse(this.txtNumero1.Text);

int numero2 = int.Parse(this.txtNumero2.Text);

int suma = numero1 + numero2;

//ESCRIBIMOS EN EL LABEL EL RESULTADO DE LA SUMA

this.lblResultado.Text = suma.ToString();

}

CONDICIONALES

Un condicional es una pregunta o varias que hacemos en un programa

Operadores de comparación:

== IGUAL

!= DISTINTO

> MAYOR

< MENOR

>= MAYOR O IGUAL

<= MENOR O IGUAL

Sintaxis:

if (condición == A)

{

//INSTRUCCIONES A

}else if (condición == B){

//INSTRUCCIONES B

}else {

//INSTRUCCIONES C

}

Operadores relacionales

&& AND

|| OR

! NEGACION

Texto

Descripción generada automáticamente

Texto

Descripción generada automáticamente

Creamos un nuevo Formulario llamado **Form02MayorMenor**

Tendremos dos cajas para escribir un número en cada caja.

Al pulsar el botón, mostraremos qué número de los dos es mayor en un Label.

Si no escribimos números en las cajas, mostramos un mensaje informativo.

## **SUPER IMPORTANTE**

Control de código o depuración. En cualquier programa existen múltiples opciones en las cuales debemos realizar un seguimiento del código para “ver” qué sucede en ejecución.

Esto sirve para los errores lógicos. Cuando un programa funciona, no tenemos fallos, pero no hace lo que deseamos.

En la línea Gris de la izquierda del código se pone un punto de interrupción pulsando en la línea que deseemos o con la tecla F9

Texto

Descripción generada automáticamente

Cuando ejecutemos, veremos que se detiene en esa línea y podremos ver o cambiar el valor de nuestras variables en ejecución.

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Para ir “Paso a paso” en la ejecución del programa utilizamos la tecla F11

En la ventana inferior izquierda, podemos visualizar los valores que las variables van recuperando o podemos cambiar su valor en ejecución.

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

CLASE DATETIME

Es una clase para almacenar Fechas y Horas.

Es una clase Wrapper, con igualar un valor, se instancia.

DateTime fecha = new DateTime();

DateTime fecha = “30/09/2024”;

Métodos y propiedades de la clase DateTime

* AddDays(numero): Añaden días a una fecha
* AddMonths(numero): Añaden meses a una fecha
* ToShortDateString(): Convierten una fecha a formato corto.
* ToLongDateString(): Convierten una fecha a formato largo
* Year
* Month
* Minutes
* Seconds
* DayOfWeek: Devuelve una enumeración con el día de la semana.
* DayOfYear: Devuelve el número del día año

Métodos **STATIC**

Un método Static es un método de la clase que es una herramienta, no utiliza el valor de la clase.

Existe un método para saber si un año es bisiesto o no. **IsLeapYear(AÑO)**

DateTime fecha = “30/09/2024”;

Escribimos esto como ejemplo:

fecha.IsLeapYear(**2030**) Nos daría ERROR

El método static utiliza los parámetros que le mandamos dentro del método.

Se llama sin utilizar el objeto, se utiliza el NOMBRE DE LA CLASE

DateTime.IsLeapYear(2030); CORRECTO

Tenemos una propiedad static llamada **Now** que nos devuelve la fecha y hora actuales.

Creamos un nuevo form llamado **Form03DateTime**

La primera norma, no importa que tengamos clases gráficas o no.

Para ejecutar instrucciones al inicio de cualquier App debemos utilizar los constructores de las clases.

Un constructor es el primer lugar dónde se ejecuta cualquier código de la clase.

El constructor de un Form se llama como el nombre de la clase y debemos escribir siempre después de **InitilizeComponent()**

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

**CODIGO FORMULARIO**

public Form03DateTime()

{

InitializeComponent();

//QUIERO PONER LA FECHA ACTUAL EN LA CAJA

DateTime fecha = DateTime.Now;

this.txtFechaActual.Text = fecha.ToString();

}

private void chkCambiarFormato\_CheckedChanged(object sender, EventArgs e)

{

//CAPTURAMOS LA FECHA DE LA CAJA DE TEXTO

DateTime fecha = DateTime.Parse(this.txtFechaActual.Text);

//PREGUNTAMOS POR EL ESTADO DEL CHECKBOX

if (this.chkCambiarFormato.Checked == true)

{

this.txtFechaActual.Text = fecha.ToLongDateString();

}

else

{

this.txtFechaActual.Text = fecha.ToShortDateString();

}

}

private void btnIncrementar\_Click(object sender, EventArgs e)

{

//CAPTURAMOS LA FECHA ACTUAL DE LA CAJA

DateTime fecha = DateTime.Parse(this.txtFechaActual.Text);

int incremento = int.Parse(this.txtIncremento.Text);

if (this.rdbDias.Checked == true)

{

fecha = fecha.AddDays(incremento);

}else if (this.rdbAnyos.Checked == true)

{

fecha = fecha.AddYears(incremento);

}

else

{

fecha = fecha.AddMonths(incremento);

}

this.txtNuevaFecha.Text = fecha.ToString();

}

}

**CLASE CHAR**

La clase Char almacena una letra solamente. Se representa con la comilla simple.

Imagen que contiene Interfaz de usuario gráfica

Descripción generada automáticamente

Casi todos sus métodos son static, es decir, se llaman desde la clase y no desde el objeto.

Dichos métodos nos sirven para averiguar el contenido del char, es decir, el tipo de carácter que tenemos almacenado.

* char.IsLetter(carácter): Indica true/false si es letra
* char.IsDigit(carácter): Indica si es un número entero
* IsNumber(carácter): Indica si es un número entero o un símbolo numérico
* IsLetterOrDigit(carácter)
* IsSymbol(carácter)
* IsPunctuation(carácter)
* IsUpper(carácter)
* IsLower(carácter)
* ToUpper(carácter)
* ToLower(carácter)

Vamos a crear un nuevo form llamado **Form04Char** que recorrerá el código Ascii y nos separará los caracteres por su tipo.

Sintaxis Bucle FOR

Imagen que contiene Texto

Descripción generada automáticamente

Interfaz de usuario gráfica, Aplicación, Word

Descripción generada automáticamente

**CODIGO FORMULARIO**

private void btnIniciarAscii\_Click(object sender, EventArgs e)

{

//RECORREMOS TODOS LOS CARACTERES DEL CODIGO ASCII 0-255

for (int i = 0; i <= 255; i++)

{

//CONVERTIMOS CADA NUMERO ASCII EN LETRA/CARACTER

char caracter = (char) i;

//PREGUNTAR POR CADA VALOR DEL CARACTER

if (char.IsLetter(caracter) == true)

{

this.txtLetras.Text += caracter;

}

else if (char.IsNumber(caracter) == true)

{

this.txtNumeros.Text += caracter;

}

else if (char.IsSymbol(caracter) == true)

{

this.txtSimbolos.Text += caracter;

}

else if (char.IsPunctuation(caracter) == true)

{

this.txtPuntuacion.Text += caracter;

}

}

}

**CLASE STRING**

Es la clase para representar textos.

Todo en Net comienza en cero, es decir, Arrays, textos, colecciones

La primera letra de un Carácter comienza en CERO

string cadena = “VISUAL STUDIO”;

0 🡪 V

1. -> I
2. 🡪 S

Para acceder a la posición de un carácter dentro de un texto o un conjunto se utiliza el índice o propiedad indizada.

cadena[0] 🡪 V

cadena[1] 🡪 I

cadena[2] 🡪 S

**METODOS O PROPIEDADES DE LA CLASE STRING**

**Length**: Devuelve la longitud de caracteres de la cadena, en

nuestro ejemplo **13**.

* **Chars[Posición]**: Devuelve el carácter que se encuentra

en la posición indicada. Es una propiedad indizada y en el

lenguaje **C# NO** se pueden escribir el nombre de las

propiedades indizadas, se utilizan desde el objeto.

* **StartsWith(Texto)**: Indica si la cadena comienza con el

Texto que le indiquemos

* **EndsWith(Texto)**: Indica si la cadena finaliza con el

Texto que le indiquemos

* **IndexOf(Texto)**: Busca el texto en la cadena y

devuelve su posición. Si no encuentra el texto en la

cadena devuelve -1.

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

* **IndexOf(Texto, índice)**: Busca el texto en la cadena a partir

De la índice y devuelve su posición. Si no encuentra el texto en la

cadena devuelve -1.

Interfaz de usuario gráfica, Texto, Aplicación, Chat o mensaje de texto

Descripción generada automáticamente

* **LastIndexOf(Texto)**: Busca el texto en la cadena y

devuelve su posición, pero comienza a buscar desde

el final de la cadena. Si no encuentra el texto

en la cadena devuelve -1.

Interfaz de usuario gráfica, Texto, Aplicación, Chat o mensaje de texto

Descripción generada automáticamente

* **Contains(Texto)**: Busca el texto en la cadena y

devuelve True o False.

* **SubString(Inicio)**: Devuelve

una parte de la cadena desde la posición de inicio.

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

* **SubString(Inicio, Numero de Caracteres)**: Devuelve

una parte de la cadena indicando el inicio y la

longitud de caracteres que queremos recuperar.

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

* **ToUpper()**: Convierte el texto a Mayúsculas.
* **ToLower()**: Convierte el texto a Minúsculas.
* **Remove(Inicio, Numero de caracteres)**: Elimina

elementos de la cadena, indicando el inicio y el

número de caracteres a eliminar.

* **Insert(Posición, Texto)**: Inserta un texto en la

cadena a partir de una posición de inicio.

* **Trim()**: Elimina espacios a la izquierda y derecha

de un texto, no los que hubiera entre medias del texto.

* **Trim('@'):** Contiene una sobrecarga dónde podemos indicar

El carácter a eliminar (Char)

* **TrimStart()**: Elimina espacios a la izquierda del texto.
* **TrimEnd()**: Elimina espacios a la derecha del texto.

Creamos un nuevo Form llamado **Form05SumarTextoNumeros**

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

**CODIGO FORMULARIO**

private void btnSumar\_Click(object sender, EventArgs e)

{

//CAPTURAMOS EL TEXTO DE LA CAJA DE NUMEROS

string textoSuma = this.txtNumeros.Text;

int suma = 0;

//RECORREMOS TODOS LOS CARACTERES DEL TEXTO

//DE UNO EN UNO

for (int i = 0; i < textoSuma.Length; i++)

{

//CAPTURAMOS CADA CARACTER DEL TEXTO

char caracter = textoSuma[i]; //"1", "2"

//DEBEMOS CONVERTIR EL CARACTER A NUMERO

int numero = int.Parse(caracter.ToString());

//SUMAMOS CADA CARACTER A LA VARIABLE suma

suma = suma + numero;

}

//DIBUJAMOS LA SUMA EN EL LABEL DE RESULTADO

this.lblResultado.Text = suma.ToString();

}

Queremos validar un Email

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

Necesitamos verificar algunos elementos:

* Que exista una @
* Que exista un punto
* Que NO exista una @ al inicio
* Que NO exista una @ al final
* Dominio de 2 a 4 caracteres

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

**CODIGO FORMULARIO**

private void btnValidarEmail\_Click(object sender, EventArgs e)

{

string email = this.txtEmail.Text;

//NECESITAMOS BUSCAR UNA @

if (email.Contains("@") == false)

{

this.lblMensaje.Text = "No existe @ en el mail";

}

else

{

this.lblMensaje.Text = "Mail correcto";

}

//VUESTROS IF A PARTIR DE AQUI

if (email.StartsWith("@") == true)

{

this.lblMensaje.Text = "Email con @ al inicio";

}

else

{

this.lblMensaje.Text = "Mail correcto";

}

if (email.IndexOf("@") == 0)

{

this.lblMensaje.Text = "Email con @ al inicio";

}

else

{

this.lblMensaje.Text = "Mail correcto";

}

if (email.EndsWith("@") == true)

{

this.lblMensaje.Text = "@ al final del Email";

}

else

{

this.lblMensaje.Text = "Mail correcto";

}

if (email.IndexOf("@") == email.Length - 1)

{

this.lblMensaje.Text = "Email con @ al final";

}

else

{

this.lblMensaje.Text = "Mail correcto";

}

//pacoserranox@gmail.com

int ultimoPunto = email.LastIndexOf(".");

string dominio = email.Substring(ultimoPunto + 1);

if (dominio.Length >= 2 && dominio.Length <= 4)

{

this.lblMensaje.Text = "Mail correcto";

}

else

{

this.lblMensaje.Text = "El dominio debe ser de 2 a 4 caracteres";

}

}

**CONJUNTO DE OBJETOS**

Un String no deja de ser un conjunto de letras establecido por un índice por cada letra.

Visual Studio trabaja con conjuntos de elementos. Dichos conjuntos siempre estarán indizados.

A Net no le importa si es un conjunto de letras, de botones, de personas o cualquier tipo de conjunto.

Todos los grupos se manejan igual, lo que nos interesa de cada grupo es el individuo.

Tenemos dos tipos de grupos:

* 1. **Arrays:** Es un conjunto de objetos definidos por su índice, pero es un conjunto estático, es decir, que no puede crecer ni decrecer.

Si creamos un Array con 5 elementos, siempre tendrá espacio para 5, nunca menos ni más, aunque sus valores estén vacíos.

Estos tipos de objetos conjuntos no se utilizan en ciertos lenguajes porque consumen mucha memoria, para que quiero 5 sillas si luego vienen 2 personas?

* 1. **Colecciones:** Es un conjunto de objetos definidos por su índice, pero es dinámico. A medida que vayamos rellenando contenido, la colección crece o decrece, por lo que la memoria está perfectamente administrada.

Nos ofrece el tamaño REAL de los elementos que tenemos en su interior.

Propiedades o métodos de una colección. Todas las colecciones son iguales por su concepto de abstracción.

El concepto de abstracción quiere decir que puedo ser capaz de reconocer un objeto por sus características, aunque el objeto no venga en su forma nativa.

* Count: Devuelve el número de elementos de una colección
* Add(objeto): Añade un objeto a una colección
* Clear(): Elimina todos los objetos de una colección.
* Remove(objeto): Elimina un elemento de la colección indicando el objeto a eliminar
* RemoveAt(índice): Elimina un elemento de la colección indicando su índice
* Contains(objeto): Pregunta si un objeto existe dentro de la colección y devuelve true/false
* IndexOf(objeto): Pregunta si un objeto existe dentro de la colección y devuelve su posición. Si no lo encuentra, devuelve -1.

Vamos a comenzar con colecciones gráficas. Tenemos unos objetos que son Listbox o ComboBox que contienen múltiples elementos en forma de colección en su interior.

Dentro del Listbox tenemos algunas propiedades de tipo colección para trabajar con sus elementos.

* Items: Colección con todos los elementos del Listbox
* SelectedItems: Colección con los elementos seleccionados del Listbox
* SelectedIndices: Colección con los índices de los elementos seleccionados del Listbox.
* SelectedIndex: Devuelve el índice del elemento seleccionado
* SelectedItem: Devuelve el elemento seleccionado
* SelectionMode: Cambia el Listbox a selección múltiple

Creamos un nuevo formulario llamado **Form07ColeccionGrafica**

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

**CODIGO FORMULARIO**

private void btnInsertar\_Click(object sender, EventArgs e)

{

string elemento = this.txtElemento.Text;

this.lstElementos.Items.Add(elemento);

}

private void btnEliminar\_Click(object sender, EventArgs e)

{

//PARA PODER ELIMINAR UN ELEMENTO DE UNA COLECCION TENEMOS DOS

//OPCIONES:

//1) BORRAR POR EL ELEMENTO

//2) BORRAR POR EL INDICE

//RECUPERAMOS EL INDICE SELECCIONADO DE LA LISTA

int indiceSeleccionado = this.lstElementos.SelectedIndex;

//ELIMINAMOS EL ELEMENTO POR EL INDICE

this.lstElementos.Items.RemoveAt(indiceSeleccionado);

}

private void btnBorrarTodo\_Click(object sender, EventArgs e)

{

this.lstElementos.Items.Clear();

}

private void lstElementos\_SelectedIndexChanged(object sender, EventArgs e)

{

//PREGUNTAMOS SI EXISTE ELEMENTO SELECCIONADO

if (this.lstElementos.SelectedIndex != -1)

{

this.lblIndiceSeleccionado.Text = "Indice seleccionado " +

this.lstElementos.SelectedIndex;

this.lblItemSeleccionado.Text = "Item seleccionado " +

this.lstElementos.SelectedItem.ToString();

}

}

**BUCLES DE REFERENCIA**

Se utilizan para recorrer objetos conjuntos de una forma sencilla.

Nos permite recorrer uno a uno cada elemento solo con un Bucle con una variable de referencia.

Vamos a crear una aplicación en la que almacenamos números dentro de una Lista.

Al pulsar un botón, sumaremos cada número de la colección de la lista y mostraremos la suma dentro de un Label.

Creamos un form llamado **Form08SumaNumerosListbox**

Interfaz de usuario gráfica, Aplicación, Word

Descripción generada automáticamente

**CODIGO FORMULARIO**

Texto

Descripción generada automáticamente

A continuación, vamos a explicar un bucle de referencia.

¿Qué es una variable de referencia? Es una variable que hace REFERENCIA a un elemento de la colección. En nuestro ejemplo de código:

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

En nuestro ejemplo, estamos haciendo dos pasos para poder recuperar el elemento de la colección y convertirlo a su tipo de dato.

Un bucle de referencia nos permite recorrer los elementos de una colección con una variable y, a la vez, convertir esos elementos al tipo de dato que deseemos.

Estamos recorriendo NUMEROS.

Sintaxis:

foreach (variable REFERENCIA in COLECCIÓN) {

}

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

**CODIGO FORMULARIO**

Texto

Descripción generada automáticamente

**METODOS DE CLASE**

Un método es un conjunto de acciones. Dentro de dicha categoría tenemos tres tipos.

* 1. Métodos de acción: **void**

Estos métodos lo que hacen es un conjunto de acciones y nada más.

void LimpiarCajas() {

this.textBox1.Text = “”;

this.textBox2.Text = “”;

this.textBox3.Text = “”;

}

Para llamar a este método de forma explicita:

private void button1\_click(){

this.LimpiarCajas();

}

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

* 1. Métodos **return**: Realizan una serie de acciones y devuelven un valor al terminar las instrucciones.

Sintaxis:

TipoDato NombreMetodo(){

//ACCIONES

return valor al TipoDato;

}

Los métodos pueden recibir parámetros. Los parámetros son recibidos dentro de los paréntesis.

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

Petición al método

Interfaz de usuario gráfica, Texto, Chat o mensaje de texto

Descripción generada automáticamente

Independientemente que sean métodos void o return, son todos iguales.

La única diferencia está en que un método **return** debe devolver su tipo de dato con un valor siempre.

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

En la Petición almacenamos el valor que devuelve en una variable.

Texto

Descripción generada automáticamente con confianza media

* 1. Métodos de evento. Los métodos de evento son métodos asociados a una acción de un Control, por ejemplo, el CLICK de un botón.



Los métodos de evento reciben siempre dos variables:

* object sender: Es el objeto que ha realizado la llamada al método
* EventArgs e: Es la variable de argumentos del evento, dependiendo del tipo de evento, tiene información o no.

**Truco**: Si el evento es **EventArgs** no hacemos nada…

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

Descripción generada automáticamente

Para asociar un Control a un Evento de forma dinámica se hace de la siguiente forma.

Control.Accion += NombreEvento;

Creamos un form llamado **Form09EventosMetodos**

Cuando realizamos doble Click sobre cualquier control nos ofrece su evento por defecto, pero en los controles gráficos tenemos más eventos, por ejemplo, al escribir en una caja, al pulsar sobre un control o al pasar el ratón por encima…

Para recuperar esos eventos debemos hacerlo desde la ventana de propiedades y con el **rayo**.

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Interfaz de usuario gráfica

Descripción generada automáticamente

**CODIGO FORMULARIO**

public Form09EventosMetodos()

{

InitializeComponent();

this.button1.Click += BotonPulsado;

this.button2.Click += BotonPulsado;

this.button3.Click += BotonPulsado;

}

//CREAMOS UN METODO DE EVENTO

void BotonPulsado(object sender, EventArgs e)

{

//Y SI NECESITO SABER QUE BOTON HEMOS PULSADO????

//LA SOLUCION ES UTILIZAR SENDER

//sender SON BOTONES

//DEBEMOS CONVERTIR SENDER EN UN OBJETO Button

Button boton = (Button)sender;

boton.BackColor = Color.Yellow;

MessageBox.Show("Boton pulsado!!!!");

}

private void lblRaton\_Click(object sender, EventArgs e)

{

}

private void lblRaton\_MouseMove(object sender, MouseEventArgs e)

{

//SIEMPRE QUE SEA DISTINTO A EventArgs TENEMOS INFORMACION

//QUE PODRIA SER UTIL...

this.lblRaton.Text = "X: " + e.X + ", Y: " + e.Y;

}

Puede ser útil dependiendo del Evento….

Por ejemplo, quiero solamente escribir los caracteres que deseemos en las cajas.

Podríamos hacer la validación en el momento en el que el usuario escribe un carácter.

Aplicación

Descripción generada automáticamente con confianza media

Seleccionamos KeyPress

Interfaz de usuario gráfica, Aplicación, Tabla

Descripción generada automáticamente

Texto

Descripción generada automáticamente

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Vamos a realizar una práctica en la que tendremos múltiples botones.

Al iniciar nuestra App, vamos a recorrer todos los botones de la Pantalla y les pondremos un número aleatorio.

Al pulsar un botón, lo que haremos será ir Sumando el valor del botón pulsado.

Creamos un nuevo form llamado **Form10BotonesNumeros**

Tenemos una colección dentro de los Formularios llamada **Controls**

Dicha colección contiene todos los controles del Formulario.

Para generar números aleatorios se utiliza la clase **Random**

Mediante la clase **Control,** podemos recorrer todos los controles del formulario, posteriormente, debemos preguntar qué tipo de control estamos recorriendo.

Imagen de la pantalla de un celular con letras y números

Descripción generada automáticamente con confianza baja

**CODIGO FORMULARIO**

public partial class Form10BotonesNumeros : Form

{

public Form10BotonesNumeros()

{

InitializeComponent();

Random random = new Random();

//RECORREMOS TODOS LOS BOTONES (Button) DENTRO DEL

//FORMULARIO MEDIANTE LA COLECCION Controls

foreach (Control control in this.Controls)

{

if (control is Button)

{

int aleatorio = random.Next(1, 200);

control.Text = aleatorio.ToString();

control.BackColor = Color.LightGreen;

//CADA BOTON LEERA EL EVENTO SUMARNUMEROS()

control.Click += SumarNumeros;

}

}

}

void SumarNumeros(object sender, EventArgs e)

{

//CAPTURAMOS EL BOTON PULSADO MEDIANTE sender

Button boton = (Button)sender;

//CAPTURAMOS EL TEXTO NUMERICO DEL BOTON QUE HEMOS PULSADO

int numero = int.Parse(boton.Text);

//QUEREMOS IR SUMANDO CADA NUMERO AL ANTERIOR PULSADO

//CAPTURAMOS EL TEXTO DEL LABEL

int suma = int.Parse(this.lblSuma.Text);

//SUMAMOS EL NUMERO A LA SUMA Y LO MOSTRAMOS

suma = suma + numero;

this.lblSuma.Text = suma.ToString();

}

}

Creamos un nuevo form llamado **Form11TablaMultiplicar**

Al iniciar la aplicación, generamos números aleatorios en botones.

Al pulsar un botón, mostraremos la tabla de multiplicar del número pulsado en el control ListBox.

Interfaz de usuario gráfica

Descripción generada automáticamente

**CODIGO FORMULARIO**

public partial class Form11TablaMultiplicar : Form

{

public Form11TablaMultiplicar()

{

InitializeComponent();

Random random = new Random();

foreach (Control miControl in this.Controls)

{

if (miControl is Button)

{

int numeroAleatorio = random.Next(1, 40);

miControl.Text = numeroAleatorio.ToString();

miControl.Click += TablaMultiplicar;

}

}

}

void TablaMultiplicar(object sender, EventArgs e)

{

Button boton = (Button)sender;

int numeroBoton = int.Parse(boton.Text);

//LIMPIAMOS EL CONTENIDO DEL LISTBOX

this.lstTablaMultiplicar.Items.Clear();

//REALIZAMOS UN BUCLE DE 1 A 10

for (int i = 1; i <= 10; i++)

{

//REALIZAMOS LA OPERACION DE MULTIPLICAR EL NUMERO

//DEL BOTON POR LA VARIABLE i DEL BUCLE

int operacion = numeroBoton \* i;

//AÑADIMOS CADA NUMERO A LA COLECCION DEL LISTBOX

this.lstTablaMultiplicar.Items.Add(operacion);

}

}

}

**PROGRAMACION ORIENTADA A OBJETOS (POO)**

Todos los lenguajes son iguales para trabajar con POO. Este concepto lo que permite es crear nuestras propias clases personalizadas.

Es muy útil para cuando estamos leyendo datos de “algo” que no existe dentro de Visual.

Por ejemplo, si estamos leyendo nombres de una base datos, tenemos un tipo de dato string.

Por ejemplo, si estamos leyendo salarios de una base de datos, tenemos un tipo int.

Si estamos leyendo los dos datos a la vez, no existe ningún tipo de dato que sea string y a la vez que sea int, tendríamos que “inventarlo” y de ahí crear nuestra propia clase y utilizarla.

Dentro de POO tenemos unos conceptos iguales para todos los lenguajes:

* Abstracción: Todas las clases tienen una misma “familia” y podemos abstraernos para manejar los elementos de una misma forma.
* Encapsulación: No visualizar lo que no deseamos que no se vea para que no sea utilizado.
* Polimorfismo: Un objeto puede tener múltiples formas en un método.
* Herencia: Un objeto hereda de otro sus características e implementa dichas características desde ahí.

Las clases pueden ser creadas dentro de un mismo proyecto o creadas en otro proyecto distinto, no existe diferencia en “utilizar” las clases, pero sí en cómo las utilizamos.

Por ejemplo, si nosotros metemos una clase en un proyecto Windows Forms, dicha clase solamente la podremos utilizar dentro de proyectos de ese tipo.

Si necesitamos utilizar esa clase en un proyecto Web, no podremos, no son compatibles.

Tenemos un tipo de proyecto llamado **Class Library (Librería de clases)** que nos permite generar clases NO GRAFICAS y utilizarlas en cualquier tipo de proyecto.

**MODIFICADORES DE ACCESO**

Este concepto tiene que ver con la encapsulación. Con “palabras clave” podemos hacer que determinados objetos no sean visibles por otras clases.

Lo que estamos hablando ahora mismo es la comunicación entre clases y que deseamos que sea visible entre ellas.

Todo esto es declaración de variables a nivel de clase:

* public: Ámbito de acceso total, lo que quiere decir que cualquier variable será visible en todo momento.
* private: Este tipo de modificador impide ver las variables declaradas en una clase.
* internal: Ámbito de acceso para proyectos. Las variables declaradas como internal solamente serán visibles entre las clases un UN MISMO PROYECTO.
* protected: Es un ámbito para la herencia. Las variables solamente serán visibles para los objetos que hereden de otros objetos.

Comenzamos creando un nuevo proyecto de Librería de clases llamado **ProyectoClases**

En el momento de crear un proyecto nuevo, tenemos dos formas:

* 1. Proyecto independiente: Tendremos abierto solamente este proyecto.
  2. Proyecto dentro de la solución: Tendremos abierto los dos proyectos a la vez.

Nosotros vamos a agregar un proyecto nuevo a nuestra solución.

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Cuando creamos un proyecto nuevo, se crea lo que se llama un **namespace** para dicho proyecto.

Un namespace es una “ruta” para acceder a las clases que tengamos en dicho proyecto.

Cuando hablamos de librerías, existen namespace para acceder a cada librería.

Por ejemplo, si queremos trabajar con datos, tenemos un namespace que se llama System.Data

Por ejemplo, si queremos trabajar con ficheros, tenemos un namespace que se llama System.IO

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Cuando tengamos un namespace propio, siempre debo poner **using** para acceder a nuestras librerías creadas propias.

De forma que si deseamos utilizar las clases del proyecto de librerías en el proyecto de formulario o cualquier otro debemos realizar dos características:

* 1. Agregar el ensamblado del proyecto de clases
  2. Incluir el namespace de las clases

TODO EL TIEMPO HABLO DEL PROYECTOCLASES

Creamos una nueva clase llamada **Persona**

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Una persona está compuesta por Nombre, Apellidos y Edad, que son sus propiedades.

Vamos a utilizar esta Clase persona desde un Formulario del otro proyecto y sacar conclusiones.

Para poder utilizar una librería en otro proyecto, lo primero es agregar dicha librería.

Sobre el proyecto de formularios Fundamentos2024 realizaremos la función de agregar una **Referencia de Proyecto**

Interfaz de usuario gráfica, Aplicación, Word

Descripción generada automáticamente

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

Para nuestras pruebas, vamos a utilizar un formulario llamado **Form12TestClases**

Para poder utilizar una clase Persona en el formulario, debemos utilizar **using** con el namespace del proyecto de clases.

Interfaz de usuario gráfica, Texto, Aplicación, Word, Correo electrónico

Descripción generada automáticamente

Texto

Descripción generada automáticamente

Cuando diseñamos nuestras clases, NOSOTROS decidimos QUE deseamos hacer y que deseamos que NO se pueda hacer.

Si tenemos una Persona, no deberíamos dejar que le pusieran una edad negativa, pero eso solo es un CONCEPTO.

Necesitamos un CODIGO para poder “validar” la edad de una persona, es decir, en algún “sitio” poder indicar que no puedan poner valores negativos a la edad.

Necesitamos crear una **propiedad extendida**, es decir, una propiedad que contenga CODIGO para yo poder hacer un IF.

Una propiedad extendida está compuesta por 3 elementos.

* Un campo de propiedad: Una variable para trabajar con la propiedad
* GET: Cuando se recupera un valor de la propiedad
* SET: Cuando se establece el valor de la propiedad

miPersona.Edad = 45;

int dato = miPersona.Edad;

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

Descripción generada automáticamente

Ejemplo de comprobación de Edad

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Cuando queremos comprobar algo, podemos ser “suaves” si no nos gusta o dar un toque grande de atención, es decir, lanzar una Excepción.

Una excepción hace que el programa se detenga.

Para lanzar una excepción se hace con **throw new Exception(“MENSAJE A LANZAR”)**

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

**PROPIEDADES AUTOIMPLEMENTADAS**

Siempre que hablamos de clases, estamos hablando de Propiedades.

Tenemos dos formas de crear propiedades: Extendidas y Autoimplementadas.

Una propiedad autoimplementada se utiliza cuando NO queremos validar nada, simplemente crear una propiedad.



Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

**PROPIEDADES ENUMERADAS**

Una propiedad enumerada nos ofrece una serie de valores para que el programador pueda seleccionar el que necesite.

Para crear una propiedad enumerada, lo primero es crear la enumeración con las diferentes posibilidades que tengamos.

Se crean de la siguiente forma y dentro del namespace (Es la única línea que va escrita ahí)

public enum NOMBREENUMERACION { NORTE, SUR, ESTE, OESTE }

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Una vez creada la enumeración, creamos la propiedad con el TIPO de enumeración

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

**METODOS**

Un método son un conjunto de acciones de una clase. Pueden devolver valor o no (void)

Pongamos que deseamos mostrar el nombre completo de una persona, actualmente estamos concatenando el nombre y el apellido dentro del Listbox para dibujarlo.

Podríamos crearnos un método que nos devuelva el nombre y los apellidos para simplificar esto.

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

Descripción generada automáticamente

**POLIMORFISMO**

Este concepto tiene que ver con los métodos. Podemos tener múltiples métodos en una clase.

¿Qué sucede si algún método queremos que realice la misma acción más o menos?

Por ejemplo: Queremos que al mostrar el nombre completo, el programador tenga la posibilidad de recuperar el nombre y los apellidos o los apellidos y el nombre para el dibujo.

Posibilidades:

* GetNombreCompleto(): Devuelve el nombre y apellidos
* GetNombreCompletoInverso(): Devuelve los apellidos y el nombre
* GetNombreCompletoMayusculas(): Devuelve el nombre y apellidos en mayúsculas
* GetNombreCompletoMinusculas(): Devuelve el nombre y apellidos en minúsculas.

En nuestro coche tenemos un pedal de freno y podemos frenar de distintas formas:

* FrenarMucho()
* FrenarPoco()
* FrenarCompleto()

No tenemos en el coche tres pedales de freno. Tenemos un solo pedal y, dependiendo como apliquemos la acción se comporta de una forma u otra.

El polimorfismo nos permite tener un mismo nombre de método y, dependiendo de sus parámetros recibidos, se comportará de una forma o de otra forma.

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

**CONSTRUCTORES DE CLASE**

Un constructor es el primer lugar/código dónde se ejecuta una clase.

Sirve para “iniciar” todos los elementos que podríamos tener en una clase.

Es el primer código que se ejecuta.

Define cómo se crea cualquier clase.

Cuando el programador crea un objeto, estamos llamando al constructor, aunque NO lo hayamos escrito en el código. Cuando ponemos **new** se crea el objeto y podemos tener un código para trabajar con dicho objeto

Interfaz de usuario gráfica

Descripción generada automáticamente con confianza media

Un constructor debe llamarse como la Clase.

Un constructor puede tener sobrecarga.

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Un constructor puede obligar a como crear una clase determinada.

Si solamente tenemos una forma de crear la clase, estamos obligando a quién utilice dicha clase a utilizar dicha forma.

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

**ACCESO FICHEROS**

El namespace de acceso a ficheros es **System.IO**

Tenemos un montón de clases: DirectoryInfo, FileInfo

Tenemos clases especializadas en leer/escribir contenido en ficheros.

TextReader, TextWriter

StreamReader, StreamWriter

Cuando estamos leyendo un fichero, sabemos el tiempo que nuestro programa tardará en leerlo???

Para leer ficheros debemos hacerlo de forma asíncrona, es decir, cuando leemos un fichero sabemos CUANDO comenzamos, pero no sabemos cuando finaliza.

Para poder utilizar métodos asíncronos debemos utilizar las siguiente palabra: **await** y **async**

Son dos instrucciones que tenemos que utilizar a la vez.

ASYNC se utiliza dentro de la definición del método

AWAIT se utiliza en la llamada al método asíncrono.

**async** void LeerFichero() {

string contenido = “”;

File fichero = new Fichero(“C:\mifichero.txt”);

contenido = **await** fichero.Read();

this.txtContenido.Text = contenido;

}

**USING EN BLOQUE**

Using es una estructura de código que nos permite asegurarnos que un objeto es creado/utilizado en un bloque de código.

También optimiza la memoria, es decir, es destruido después del bloque using

Sintaxis:

using (OBJETO){

//ACCESO A DICHO OBJETO

}

//AQUI EL OBJETO ESTA DESTRUIDO

Pongamos el siguiente ejemplo de código:

public class AccesoDatos{

public AccesoDatos(string cadenaConexion){

//AQUÍ NOS CONECTAMOS

}

public string LeerDatoBaseDatos(){

//LEEMOS DATOS Y DEVOLVEMOS UN VALOR

}

}

Cuando vayamos a crear la clase, debería de conectar con el servidor de base de datos y después, debería Leer los datos de la base de datos.

AccesoDatos data = new AccesoDatos(“server: https:/…”);

//ESTAMOS RECUPERANDO DATOS SIN SABER SI SE HA CONECTADO TODAVIA

string contenido = data.LeerDatoBaseDatos();

Con using nos aseguramos que un objeto ha sido creado antes de continuar. Dicho bloque se asegura que ha creado el objeto.

using (AccesoDatos data = new AccesoDatos(“server:https://…”)){

//PARA LLEGAR A ESTA LINEA, HA TENIDO QUE CREAR EL OBJETO

string contenido = data.LeerDatoBaseDatos();

}

Creamos un nuevo Form llamado **Form13Files**

Texto

Descripción generada automáticamente

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

**CODIGO FORMULARIO**

public partial class Form13Files : Form

{

string rutaFichero;

public Form13Files()

{

InitializeComponent();

//INICIAMOS LA RUTA DENTRO DEL CONSTRUCTOR

//EL SIMBOLO DE \ ES UN SIMBOLO DEL LENGUAJE, POR EJEMPLO

//ES UTILIZADO PARA DAR UN ENTER EN UN STRING \n

//TENEMOS DOS OPCIONES:

//1) PONER DOBLE CONTRABARRA

this.rutaFichero = "C:\\Users\\Serra\\Documents\\test.txt";

//2) UTILIZAR @ ANTES DEL STRING

this.rutaFichero = @"C:\Users\Serra\Documents\test.txt";

}

private async void btnLeerFichero\_Click(object sender, EventArgs e)

{

//NECESITAMOS LA RUTA QUE YA LA TENEMOS A NIVEL DE CLASE

FileInfo file = new FileInfo(this.rutaFichero);

//CON LA CLASE TextReader LEEMOS TEXTO PLANO DE UN FICHERO

using (TextReader reader = file.OpenText())

{

//SI TENEMOS METODOS Async PARA LEER, DEBEMOS UTILIZARLOS

//LOS METODOS ASYNC FINALIZAN CON ESA PALABRA

//LEEMOS EL CONTENIDO DEL FILE

string contenido = await reader.ReadToEndAsync();

//DEBEMOS CERRAR EL FICHERO

reader.Close();

this.txtContenido.Text = contenido;

}

}

private async void btnGuardarFichero\_Click(object sender, EventArgs e)

{

//NECESITAMOS LA RUTA AL FICHERO

FileInfo file = new FileInfo(this.rutaFichero);

using (TextWriter writer = file.CreateText())

{

//RECUPERAMOS EL CONTENIDO DE LA CAJA

string contenido = this.txtContenido.Text;

//ESCRIBIMOS EL CONTENIDO EN EL FILE

await writer.WriteAsync(contenido);

//SIEMPRE QUE ESCRIBAMOS FILES, DEBEMOS LLAMAR AL

//METODO FLUSH PARA QUE HAGA EL VACIADO DE MEMORIA

await writer.FlushAsync();

writer.Close();

MessageBox.Show("Datos almacenados");

}

}

}

Vamos a realizar una modificación de código y las rutas no serán estáticas.

Podremos seleccionar el fichero que deseemos para leerlo.

Existe una clase de Forms llamada **OpenFileDialog** que es para abrir ficheros.

Interfaz de usuario gráfica, Aplicación, Word

Descripción generada automáticamente

Contiene un método llamado **ShowDialog()** para mostrar la ventana.

El objeto contiene una propiedad para recuperar la ruta del fichero seleccionado llamada **FileName**

Al igual que tenemos OpenFileDialog, tenemos también otra clase llamada **SaveFileDialog** que hace lo mismo igual, pero en el título pone Guardar Como…

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

Modificamos el código de los botones y quitamos el código del constructor

private async void btnLeerFichero\_Click(object sender, EventArgs e)

{

//NECESITAMOS LA RUTA DEL FICHERO

this.openFileDialog1.ShowDialog();

string path = this.openFileDialog1.FileName;

FileInfo file = new FileInfo(path);

//CON LA CLASE TextReader LEEMOS TEXTO PLANO DE UN FICHERO

using (TextReader reader = file.OpenText())

{

//SI TENEMOS METODOS Async PARA LEER, DEBEMOS UTILIZARLOS

//LOS METODOS ASYNC FINALIZAN CON ESA PALABRA

//LEEMOS EL CONTENIDO DEL FILE

string contenido = await reader.ReadToEndAsync();

//DEBEMOS CERRAR EL FICHERO

reader.Close();

this.txtContenido.Text = contenido;

}

}

private async void btnGuardarFichero\_Click(object sender, EventArgs e)

{

//NECESITAMOS LA RUTA AL FICHERO

this.saveFileDialog1.ShowDialog();

string path = this.saveFileDialog1.FileName;

FileInfo file = new FileInfo(path);

using (TextWriter writer = file.CreateText())

{

//RECUPERAMOS EL CONTENIDO DE LA CAJA

string contenido = this.txtContenido.Text;

//ESCRIBIMOS EL CONTENIDO EN EL FILE

await writer.WriteAsync(contenido);

//SIEMPRE QUE ESCRIBAMOS FILES, DEBEMOS LLAMAR AL

//METODO FLUSH PARA QUE HAGA EL VACIADO DE MEMORIA

await writer.FlushAsync();

writer.Close();

MessageBox.Show("Datos almacenados");

}

}

### **COLECCIONES NO GRAFICAS**

Hemos visto hasta ahora colecciones utilizando controles como, por ejemplo, Items de un Listbox.

La cuestión es NO necesitar un control visual para poder almacenar conjuntos.

Existen colecciones no gráficas, que nos permiten almacenar información de conjuntos sin necesidad de dibujar dichos conjuntos.

Por ejemplo, si queremos cargar los datos de múltiples clases, por ejemplo, un conjunto de Personas.

Una colección nos permite almacenar conjuntos de elementos sin necesidad de mostrarlos, cuando lo deseemos los mostraremos.

Tenemos la colección **List<T>**, que nos permite almacenar objetos de tipo T. Todos los objetos deben ser del mismo tipo de T. Si T son string, todos los objetos a almacenar deben ser de ese tipo.

Como toda colección, tiene métodos Add, Clear, Remove…

Vamos a almacenar un conjunto de nombres en memoria (List). Posteriormente, cuando lo necesitemos, pintaremos los nombres en un Listbox.

Por último, en ese ejemplo lo que haremos será guardar dichos nombres en un fichero físico.

Creamos un nuevo form llamado **Form14ColeccionesNoGraficas**

Para tener una colección, la crearemos a nivel de clase, declarada en el formulario.

Declaramos la variable de colección a nivel de clase y la creamos en el Constructor de la clase.

Interfaz de usuario gráfica, Aplicación, Word

Descripción generada automáticamente

CODIGO FORMULARIO

public partial class Form14ColeccionesNoGraficas : Form

{

List<string> coleccionNombres;

public Form14ColeccionesNoGraficas()

{

InitializeComponent();

this.coleccionNombres = new List<string>();

}

private void brtnGuardarNombre\_Click(object sender, EventArgs e)

{

string nombre = this.txtNombre.Text;

this.coleccionNombres.Add(nombre);

this.lblMensaje.Text = "Nombres almacenados: " + this.coleccionNombres.Count;

this.txtNombre.Text = "";

}

private void btnMostrarNombres\_Click(object sender, EventArgs e)

{

this.lstNombres.Items.Clear();

foreach (string name in this.coleccionNombres)

{

this.lstNombres.Items.Add(name);

}

}

}

En lugar de nombres y texto string, lo que vamos a realizar será almacenar Personas.

Tenemos una clase llamada Persona.

Cada vez que pulsemos el botón, lo que haremos será guardar una persona.

Cuando pulsemos sobre una Lista visual, mostraremos los datos de la persona.

Creamos un nuevo Form llamado **Form15ColeccionPersonas**