**FUNDAMENTOS VISUAL STUDIO**

Software:

* **Visual Studio Community** (IDE)

[Visual Studio 2022 Community Edition: descargar la versión gratuita más reciente](https://visualstudio.microsoft.com/es/vs/community/)

Lo descargamos y, en la instalación, debemos seleccionar lo siguiente:



Mi entorno está en inglés. Podéis perfectamente descargar también el paquete de idioma

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

Descripción generada automáticamente

* Cuenta de **GitHub**. GitHub es un repositorio de código. Es de Microsoft, pero no tiene código solo de Microsoft, contiene de todo, desde código Front, React, Angular o Java que es Back.

Si no tenemos cuenta, nos hacemos una cuenta para poder trabajar con nuestros proyectos.

[GitHub · Build and ship software on a single, collaborative platform · GitHub](https://github.com/)

* **SQL Server Express**: Es la base de datos de Microsoft. Al instalar, seleccionamos Basic y ya está.

[Descargas de SQL Server | Microsoft](https://www.microsoft.com/es-es/sql-server/sql-server-downloads)



* Después de instalar SQL Server Express, nos pedirá, en la misma instalación, al finalizar, instalar **SQL Server Management Studio**.

Lo instalamos también con siguiente 🡪 siguiente.

<https://learn.microsoft.com/es-es/sql/ssms/download-sql-server-management-studio-ssms?view=sql-server-ver16#download-ssms>

Temario: **NET CORE LENGUAJE C#**

* **Windows Forms**. Será nuestro entorno de trabajo Visual. Iremos creando múltiples formularios para entender cada característica del lenguaje.
* **Fundamentos del lenguaje**: Iteraciones, sintaxis, Clases, POO, Colecciones.
* **Inyección de dependencias**:Nos permite tener todas las clases y elementos administrados en un solo punto de un proyecto.
* **Ado Net**:Es la tecnología de acceso a datos de Microsoft (antigua). Se sigue utilizando, pero le ha ganado la batalla otra tecnología.
* **LINQ y Entity Framework**: Es la tecnología de acceso a datos actual (2024) de Microsoft.

El temario que tenéis delante es un temario que se llama Net Framework. Es un temario dónde se hacían solamente aplicaciones para Windows y solo con Windows.

¿Qué es Net Core? Esta tecnología nos permite crear aplicaciones del fabricante Microsoft para distintas plataformas.

Con una aplicación, puede distribuirla en un Windows, Linux un MacOS.

Si montamos una aplicación Web, podemos ponerla en un servidor Apache, IIS, Nginx.

Todo lo que hagamos, estará listo para producción, es decir, cuando compilemos un programa, podremos utilizarlo en cualquier PC, sin necesidad del programa de Visual Studio.

Por ejemplo, si hacemos una calculadora, podría ejecutarla directamente sin necesidad de Visual, simplemente ejecutando su EXE.

Tenemos diferentes proyectos dentro de Visual Studio.

* Librerías: Son proyectos dónde solamente tenemos funcionalidades, no tenemos parte gráfica, solo lógica.
* Visuales (Forms): En nuestro caso, tenemos proyectos de Windows, pero, también tenemos proyectos Web visuales o proyectos para móviles.

**Metodología de trabajo**

Tendremos un solo proyecto y múltiples formularios.

Iremos generando más proyectos a medida que vayamos aprendiendo, pero un solo proyecto para cada característica.

GITHUB será el centro de trabajo para vosotros y para mí.

Vamos a crear un nuevo proyecto llamado **FundamentosOctubreNetCore** de tipo **Windows Forms**.

**Nota:** Aunque no tengamos todavía Visual Studio instalado, no pasa nada, lo vamos haciendo a lo largo de la clase y podéis coger mis recursos.

Podemos clonar un repositorio de GitHub (Paco). Si todavía no lo tenemos instalado, podemos recuperar lo que haya puesto el profe en la nube.

Interfaz de usuario gráfica, Texto, Aplicación, Word

Descripción generada automáticamente

Dentro de GitHub, simplemente copiamos el código que pone **.git**

Debemos estar seguros que hemos seleccionado el lenguaje **C#**.

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

A continuación, nos pedirá un nombre y una ubicación.

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

Descripción generada automáticamente

Lo siguiente que nos pregunta son las librerías de Net Core a utilizar.

La última es la versión 8.0.

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Voy a subir la documentación y el nuevo proyecto a GitHub para que podáis tenerlo en casa.

Sobre la esquina inferior derecha, seleccionamos **Add to Source Control**

Interfaz de usuario gráfica, Aplicación, Teams

Descripción generada automáticamente

Nos abrirá una ventana para poder subir el proyecto a GitHub

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

Y podremos visualizar nuestro proyecto subido a la nube de GitHub.

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

Descripción generada automáticamente

Mi proyecto de Github está en la siguiente dirección

<https://github.com/serraguti/FundamentosOctubreNetCore/>

Una vez que tenemos los elementos subidos en Github, debemos descargar los posibles cambios que vayamos haciendo en el programa y los cambios que irá haciendo Paco en el proyecto.

Tenemos dos opciones:

1. **Descargar todo el proyecto de GitHub desde CERO**.

Desde la página de GitHub copiamos el código GIT.

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

Descripción generada automáticamente

Clonamos el repositorio desde Visual Studio

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

Descripción generada automáticamente

Le damos un nombre al proyecto para almacenar la carpeta.

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

Descripción generada automáticamente

1. **Implementar los cambios** que se han realizado en el proyecto, una vez que lo hemos descargado.

En este caso, suponemos que ya tenemos el proyecto descargado. Dicho proyecto, desde el origen, ha tenido cambios y YO también he realizado cambios.

Por un lado, está mi proyecto (Alumno) y, por otro lado, está el proyecto de Profesor (Origen)

Debemos ir a una ventana llamada **Git Changes**

Ponemos nuestro mensaje (lo que sea) y pulsamos en **Commit All**



Una vez que hemos salvado todo, indicamos **Pull** sobre la flecha

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

1. **Cuando creemos un formulario, con el nombre que diga Paco, pues le agregamos nuestro nombre o nuestras iniciales y así tenemos ficheros distintos, uno para mis cosas y otro para la explicación y los Pull y así no entran en conflicto.**

Es lo mismo, debemos hacer un Commit para guardar los cambios en local y después, hacer un **Pull**

# IDE VISUAL STUDIO

**Importante:** Todas las ventanas que voy describiendo a continuación, estarán en el Menú **View/Ver**

* **App Principal:** Es la parte central de nuestra Aplicación, en nuestro ejemplo es un Formulario.

No podemos editar el formulario ni nada si estamos en ejecución.

Interfaz de usuario gráfica, Aplicación, Word

Descripción generada automáticamente con confianza media

* **Explorador de soluciones/Solution Explorer**: El explorador de soluciones nos permitirá acceder a nuestras clases y características que tengamos en nuestra App. Es decir, accedemos a los ficheros del proyecto.

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

Descripción generada automáticamente

* **Barra de herramientas/ToolBox**: Solamente vamos a visualizar la barra de herramientas cuando estemos en MODO DISEÑO/DESIGN

En dicha barra, tendremos todos los controles para diseñar nuestro formulario.

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

* **Propiedades/Properties:** Solamente se visualiza en modo Diseño/Design y nos permite modificar el aspecto Visual de los controles.

Los nombres de los controles (Button, Label, TextBox) debemos indicarlos si vamos a trabajar con ellos en código. Modificaremos su propiedad **name** para acceder a ella por código.

Dependiendo del control, asignaremos un nombre u otro

Ejemplo:

Button: **btnDescripcion** de la acción.

TextBox: txtDescripcion del contenido de la caja

Label: lblDescripcion de lo que tenga el Label.

En este lenguaje, se diferencia mayúsculas de minúsculas.

**Nota:** Ordenamos la ventana de propiedades de forma alfabética para visualizar mejor todo por su nombre.

Interfaz de usuario gráfica, Tabla

Descripción generada automáticamente

Por ejemplo, si vamos a poner un botón para Pulsar para algo

**Name: btnPulsar**

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

Otro ejemplo con una Caja de texto

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

Descripción generada automáticamente

Por otro lado, tenemos el código lógico, es decir, qué queremos hacer con los dibujos.

Este tipo de arquitectura de Windows Forms funciona y está orientada a eventos. Un evento es un **CUANDO** queremos hacer algo.

Dentro de dicho CUANDO es dónde escribiremos.

Para recuperar un EVENTO, necesitamos realizar **doble click** sobre el control que deseemos, por ejemplo, en este caso, en un botón.

Texto, Aplicación

Descripción generada automáticamente

**CONTROL DE ERRORES VISUAL STUDIO**

Existen multitud de errores dentro de un programa.

Vamos a comenzar por visualizar el más sencillo y cómo podemos solucionarlo con nuestro entorno gráfico.

**Error de compilación**. Dicho error sucede cuando tenemos algo mal escrito en nuestro código y no nos permite compilar. Compilar es ejecutar el proyecto.

Si tenemos un error, siempre diremos que NO en la ventana que nos aparecerá.

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

Descripción generada automáticamente

Estamos en un proyecto, para poder ejecutar el proyecto, todo debe estar bien escrito, no importa el lugar dónde esté escrito. Si tenemos errores, no podremos continuar.

Al pulsar en NO, vemos que nos abre una ventana de errores en la parte inferior.

Dicha ventana de errores nos irá llevando a todos los errores que tengamos en el proyecto para poder ejecutarlo

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

Descripción generada automáticamente

Bastará con pulsar sobre CADA ERROR y solucionarlo.

# CODIGO DE OBJETOS EN EVENTOS

Cuando escribamos código dentro de un Evento (Click), debemos acceder a los objetos del formulario con la palabra clave **this**

Por ejemplo, tenemos una caja de texto ahora mismo y para acceder a ella en nuestro código, pondremos this.

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Tipos de propiedades en objetos y clases. Todo es una clase, pero hablamos ahora mismo de clases gráficas.

Para saber si estamos accediendo a una propiedad, debemos mirar la llave inglesa.

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Tenemos varios tipos de propiedad en código. Por ahora, nos da igual lo que hagan dichas propiedades.

1. **Propiedades de Tipos primitivos (números, letras):** Podemos representar directamente los números o las letras. Las letras se escriben entre comillas dobles y todo termina con punto y coma.

Interfaz de usuario gráfica

Descripción generada automáticamente con confianza media

1. Propiedades de **New Object:** No sabemos el tipo que tiene la propiedad (ni idea). Lo que se escribe es **new** y lo que nos diga la ayuda de Visual Studio.

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente



1. **Propiedades enumeradas:** Son “herramientas” para el lenguaje y el programador. Son una ayuda dónde me ofrecen todas las posibilidades que la propiedad necesite.

**Lo único que tenemos que visualizar es que sean de color AMARILLO.**

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente



### DECLARACION DE VARIABLES TIPOS PRIMITIVOS

Una variable es un espacio de memoria para almacenar algún dato de nuestro programa.

Por ejemplo, podemos pedir dos números al usuario y mostrarle la suma.

En cualquier programa, necesitamos almacenar elementos:

1. Necesitamos almacenar el numero 1.
2. Necesitamos almacenar el numero 2
3. Necesitamos almacenar la suma de los dos numeros

Declaración de las variables se realiza con la siguiente sintaxis:

TIPODATO nombreVariable;

Por ejemplo:

string texto = “Hola mundo”;

int numero = 17;

Vamos a comenzar viendo la teoría de los tipos primitivos. Un tipo primitivo o Wrapper se iguala directamente a un valor para poder ser utilizado.

El valor puede ser representado gráficamente.

PRIMITIVOS

* **char**: Representa un carácter. Se escribe entre comillas simples.
* **byte**: Almacena números de 0-255.
* **short**: Almacena números hasta 27.900
* **int**: Almacena más precisión
* **long**: El mayor número de precisión de un entero.
* **float, double, decimal**: Representan números decimales
* **bool**: true/false
* **DateTime**: Representa fechas.
* **object**: En realidad es cualquier elemento que tengamos dentro de Visual Studio. Podemos almacenar cualquier clase dentro de esta variable.

**CONVERSION DE TIPOS**

Dentro de cualquier lenguaje de programación, necesitamos almacenar valores según su tipo.

En ocasiones, dichos valores NO vienen en el formato que necesitamos para trabajar, por ejemplo, un número podría venir como texto, por lo que tendríamos que convertirlo para poder trabajar con operaciones matemáticas.

**Conversiones automáticas**

Estas conversiones se realizan cuando el tipo de dato a almacenar es **menor** que el tipo que estamos utilizando.

No hablamos del valor, estamos hablando del TIPO.

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

**Conversiones entre tipos compatibles**

Se realiza la conversión cuando los tipos son compatibles (número con número) y el tipo es **mayor** a lo que vamos a almacenar.

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

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

**Convertir de String a cualquier primitivo**

Esta conversión nos permitirá almacenar cualquier texto en un tipo primitivo deseado (números, booleans, fechas…)

El valor del texto debe ser compatible, esto no hace milagros, es decir, no podemos convertir el texto “Lunes” a un número.

Se utiliza un método que tienen TODOS los primitivos llamado **Parse**

Dicho método se llama desde la clase que deseamos convertir y aplicando el método **Parse**

**TipoDato.Parse(string)**

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

**Convertir cualquier clase a String**

Todas las clases contienen un método llamado **ToString()** para convertir a su representación de string, es decir, para almacenar su valor en un string.

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Vamos a realizar un nuevo formulario para sumar dos números.

En el momento de crear un formulario, iremos dando diferentes nombres con una numeración para saber el orden que hemos ido siguiendo.

**Nota:** Si estáis con mi proyecto y necesitáis “jugar”, llamar al formulario de otra forma, es decir, con vuestra inicial al final, por ejemplo.

En el **explorador de soluciones**, sobre el **proyecto** (negrita), botón derecho y **Agregar Nuevo Formulario**.

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

Llamaremos a nuestro formulario **Form01SumarNumeros**

Cuando agregamos un nuevo formulario al proyecto, no se incluye por defecto como formulario de inicio, tengo que indicarlo de forma explícita

Para indicar el formulario de inicio, lo haremos en la clase **Program**

**PROGRAM**

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

FORMULARIO

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

**CODIGO FORMULARIO**

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

{

    //CAPTURAMOS LOS DATOS DE CADA CAJA DE TEXTO EN UNA

    //VARIABLE

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

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

    //DECLARAMOS UNA VARIABLE PARA ALMACENAR LA SUMA

**int** suma **=** numero1 **+** numero2;

    //MOSTRAMOS EL RESULTADO EN EL LABEL DEL FORMULARIO

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

}

**CONDICIONALES**

Un condicional es una instrucción que hacemos en un programa para poder evaluar unas condiciones en nuestro código.

Por ejemplo, ahora mismo, si no escribimos nada en las cajas, nuestro programa podría fallar.

Mediante un condicional podemos hacer que nuestro programa se comporte de forma distinta.

Operadores de comparación

== IGUAL

!= DISTINTO

> MAYOR

< MENOR

>= MAYOR O IGUAL

<= MENOR O IGUAL

Sintaxis:

if (condicion == true)

{

//CONDICION TRUE

}

Podemos realizar más preguntas como, un IF-ELSE:

if (condicion == true){

//CONDICION TRUE

}else{

//CONDICION FALSE

}

Tenemos otro código para hacer más preguntas dentro de un mismo IF:

if (condicionA == true) {

//TRUE CONDICION A

}else if (condicionB == true) {

//TRUE CONDICION B

}else {

//SI NO SE CUMPLEN EL RESTO DE PREGUNTAS

}

**Operadores relacionales**: Estos operadores nos permiten agrupar más preguntas dentro de una sola condición

&& AND: Cada pregunta debe cumplirse

|| OR: Cualquier pregunta entra dentro de la condición IF

! NOT: Negar una condición

Por ejemplo, podemos modificar el código del ejemplo anterior y que muestre un mensaje en el caso de que no tenga contenido ninguna de las cajas.

**MODIFICACION DEL EJEMPLO ANTERIOR**

Texto

Descripción generada automáticamente

También podemos preguntar por las dos cajas a la vez. Si no ha escrito nada en ninguna de las cajas, que muestre un mensaje.

Texto

Descripción generada automáticamente con confianza media

Creamos un nuevo formulario llamado **Form02PositivoNegativoCero**

* El formulario tendrá una caja de texto para pedir un número.
* Tendremos un botón para evaluar dicho número.
* Cuando pulsemos el botón, mostraremos en un LABEL si el número es positivo, negativo o es cero.

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

**CODIGO FORMULARIO**

**private** **void** **btnEvaluarNumero\_Click**(**object** sender, EventArgs e)

{

    //CAPTURAMOS EL NUMERO DE LA CAJA DE TEXTO

**int** numero **=** **int.Parse**(**this.**txtNumero**.**Text);

    //DEBEMOS PREGUNTAS POR LAS DIFERENTES OPCIONES QUE

    //TENEMOS EN EL PROGRAMA: POSITIVO, NEGATIVO O CERO

**if** (numero **>** 0)

    {

**this.**lblResultado**.**Text **=** "POSITIVO";

    }

**else** **if** (numero **<** 0) {

**this.**lblResultado**.**Text **=** "NEGATIVO";

    }

**else**

    {

**this.**lblResultado**.**Text **=** "CERO";

    }

}

Realizar una práctica en la que evaluaremos dos números.

Creamos un nuevo formulario llamado **Form03NumerosMayorMenor**

* Tendremos dos cajas de texto para introducir dos números.
* Un botón para evaluar la comparación de los números.
* Al pulsar el botón, debemos mostrar en un LABEL qué número es mayor al otro

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

**CODIGO FORMULARIO**

**private** **void** **btnMostrarMayor\_Click**(**object** sender, EventArgs e)

{

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

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

    //DECLARAMOS UNA VARIABLE PARA SABER EL NUMERO MAYOR

**int** mayor **=** 0;

**if** (numero1 **>** numero2)

    {

        mayor **=** numero1;

    }

**else**

    {

        mayor **=** numero2;

    }

**this.**lblResultado**.**Text **=** "El número mayor es " **+** mayor;

}

Vamos a crear otro formulario llamado **Form04MayorTresNumeros**

* Necesitamos tres cajas de texto para pedir tres números.
* Un botón para mostrar una serie de datos.
* Al pulsar el botón, debemos mostrar los siguientes resultados:
  + Mayor
  + Menor
  + Intermedio

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

**CODIGO FORMULARIO**

**private** **void** **btnMostrarResultados\_Click**(**object** sender, EventArgs e)

{

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

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

**int** numero3 **=** **int.Parse**(**this.**txtNumero3**.**Text);

**int** mayor **=** 0;

**int** menor **=** 0;

**int** intermedio **=** 0;

    //DEBEMOS COMPARAR EL NUMERO 1 CON EL RESTO DE NUMEROS QUE TENGAMOS

    //VOY A UTILIZAR EN LA COMPARACION TAMBIEN SI SON IGUALES >=

**if** (numero1 **>=** numero2 **&&** numero1 **>=** numero3)

    {

        mayor **=** numero1;

    }**else** **if** (numero2 **>=** numero1 **&&** numero2 **>=** numero3)

    {

        mayor **=** numero2;

    }

**else**

    {

        mayor **=** numero3;

    }

    //COMPARAMOS LOS NUMEROS ENTRE SI CON EL OPERANDO DE <=

**if** (numero1 **<=** numero2 **&&** numero1 **<=** numero3)

    {

        menor **=** numero1;

    }**else** **if** (numero2 **<=** numero1 **&&** numero2 **<=** numero3)

    {

        menor **=** numero2;

    }

**else**

    {

        menor **=** numero3;

    }

    //PARA EL INTERMEDIO, VAMOS A SUMAR TODOS LOS NUMEROS QUE TENEMOS

**int** suma **=** numero1 **+** numero2 **+** numero3;

    //EL INTERMEDIO SERA LA SUMA MENOS EL MAYOR Y EL MENOR

    intermedio **=** suma **-** mayor **-** menor;

**this.**lblMayor**.**Text **=** "Mayor: " **+** mayor;

**this.**lblMenor**.**Text **=** "Menor: " **+** menor;

**this.**lblIntermedio**.**Text **=** "Intermedio " **+** intermedio;

}

**CALCULAR FECHA NACIMIENTO DIA SEMANA**

Creamos un nuevo formulario llamado **Form05DiaNacimiento**

* 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.

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

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



Debemos pedir el **día**, el número de **mes** y el **año** que el usuario haya nacido.

A partir de esto 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. **Solamente para estos dos meses**

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 🡪 4

1. Dividir el año entre 4

1997 / 4 🡪 499

1. Dividir el año entre 100

1997 / 100 🡪 19

1. Dividir el año entre 400

1997 / 400 🡪 4

1. Sumar el día, 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 🡪 2514

1. Dividir el resultado anterior entre 7.

2514 / 7 🡪 359

1. Restar el número del paso 5 con el número del paso 6 por 7.

2514 – (359 \* 7) 🡪 1

1. Miramos la tabla y vemos que el número 1 corresponde a **DOMINGO**

**CODIGO 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);

    //15/06/1997

**if** (mes **==** 1)

    {

        mes **=** 13;

        anyo **=** anyo **-** 1;

    }**else** **if** (mes **==** 2)

    {

        mes **=** 14;

        anyo **=** anyo **-** 1;

    }

    //COMENZAMOS A REALIZAR LAS OPERACIONES MATEMATICAS

    //Multiplicar el Mes más 1 por 3 y dividirlo entre 5

**int** op1 **=** ((mes **+** 1) **\*** 3) **/** 5;

    //Dividir el año entre 4

**int** op2 **=** anyo **/** 4;

    //Dividir el año entre 100

**int** op3 **=** anyo **/** 100;

    //Dividir el año entre 400

**int** op4 **=** anyo **/** 400;

    //Sumar el día, 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.

**int** op5 **=** dia **+** (mes **\*** 2) **+** anyo **+** op1 **+** op2 **-** op3 **+** op4 **+** 2;

    //Dividir el resultado anterior entre 7.

**int** op6 **=** op5 **/** 7;

    //Restar el número del paso 5 con el número del paso 6 por 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 **=** "Tenemos un error, Houston";

    }

}

# **SUPER IMPORTANTE EN DESARROLLO**

Control de código o depuración. En cualquier programa, existen múltiples tipos de error.

Hemos visto que podemos controlar dos errores:

* **Errores de compilación**: Este tipo de errores, suceden cuando tenemos algún problema con la sintaxis del programa. Simplemente, al ejecutar, indicamos que NO y solucionamos el problema de sintaxis.
* **Errores en ejecución**: Suceden cuando el usuario realiza acciones que no hemos controlado por código, por ejemplo, si estamos pidiendo números en una caja de texto y el usuario escribe letras o deja la caja vacía. Estos errores se controlan mediante preguntas IF.

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

* **Errores lógicos**: Este tipo de errores son los más importantes y los que más difícil solución tienen. El programa funciona, todo está estupendo y no tenemos ningún error, pero no hace lo que deseamos, se comporta de forma distinta.

La única forma que tenemos de controlar estos errores es “visualizar” qué está sucediendo en nuestro código y con nuestras variables.

La forma de solucionar este problema es ir paso a paso dentro de la ejecución de nuestro programa y VER los resultados de las variables o lo que estamos haciendo.

En la línea gris que tenemos a la izquierda del código, podemos poner **Puntos de interrupción** para que el programa se detenga en ese momento exacto.

Si pulsamos, nos pondrá una línea roja indicando la parada. También podemos utilizar la tecla F9.

Texto

Descripción generada automáticamente

Cuando ejecutemos el programa, en algún momento se detendrá en esa línea o en otras que hayamos marcado.

Podemos visualizar nuestras variables en ejecución o ir paso a paso con la tecla F11.

Si ponemos el ratón encima de las variables, veremos su valor una vez que hayan pasado el punto de interrupción.

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente



Mientras estamos en ejecución, podemos visualizar el valor de las variables locales del programa en una ventana inferior llamada **Locals**

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

Descripción generada automáticamente

Una vez que hemos terminado la ejecución y hemos encontrado el problema, podemos quitar el punto de interrupción y ejecutar el programa sin depurar.

Todo en Visual Studio son clases. Tenemos clases gráficas como, por ejemplo, Button, TextBox o clases no gráficas que nos permiten realizar acciones dependiendo de su tipo. (int, float, string)

Cada clase tendrá una serie de Propiedades o métodos con los que podremos trabajar, todo depende de la clase. Dichos métodos son herramientas para facilitar el trabajo en el momento de realizar tareas.

### Clase DateTime

Es una clase en la que podemos utilizar y almacenar Fechas y horas.

Es una clase wrapper, lo que quiere decir que no es necesario que pongamos new para crear un valor, podemos perfectamente igualar su valor a una fecha.

DateTime fecha = “29/10/2024”;

Utilizando **new**

DateTime fecha = **new** DateTime(“29/10/2024”);

Métodos y propiedades de la clase DateTime.

* AddDays(número): Añade un número de días a una fecha
* AddMonths(número): Añade un número de meses a una fecha
* AddYears(número): Añade un número de años a una fecha
* ToLongDateString(): Devuelve la fecha en formato string y nos la devuelve en su formato largo: **Martes 29 de octubre de 2024**
* ToShortDateString(): Devuelve la fecha en formato string y nos la devuelve en su formato corto: **29/10/2024**
* Year: Devuelve el año de una fecha
* Minutes: Devuelve los minutos de una fecha
* Month: Devuelve los meses de una fecha
* Seconds: Devuelve los segundos de una fecha
* DayOfWeek: Devuelve una enumeración con el día de la semana en inglés.
* DayOfYear: Indica el día del año actual.
* Parse(string): Convierte un string al tipo de dato DateTime

Métodos de tipo **static.** Este concepto no tiene que ver con la clase DateTime.

Un método static es un método que pertenece a la clase y es una herramienta.

Dicha herramienta NO utiliza el valor del objeto de la clase.

Por ejemplo, dentro de DateTime tenemos un método llamado **IsLeapYear(AÑO)** que nos devuelve un true/false si el año es bisiesto o no. Este método es **static**

DateTime fecha = “29/10/2024”;

Escribimos el siguiente código:

fecha.IsLeapYear(2030);

El método static utiliza el valor del parámetro que le enviamos, no utiliza el valor del objeto (2024).

De hecho, dicho método NO se llama desde el objeto (fecha), se llama desde el nombre de la clase (DateTime).

DateTime.IsLeapYear(2030);

Tenemos una propiedad muy útil llamada **Now** que nos indica la fecha y la hora actual para un tipo DateTime.

DateTime fecha = DateTime.**Now**;

Para probar esta teoría, vamos a crear un nuevo formulario llamado **Form06DateTime**

Cuando estamos con clases y queremos hacer algo al inicio de dicha clase, se utilizan los constructores.

Un constructor es el lugar dónde se inicializan las variables o los elementos que deseemos en una clase, por ejemplo, en una clase Formulario.

Un constructor se llama exactamente igual que la clase en la que estemos trabajando.

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

Descripción generada automáticamente

En nuestro ejemplo, necesitamos dibujar la fecha actual dentro de la caja de texto Fecha Actual y lo haremos en el constructor (Al inicio de la clase)

**Nota:** Tenemos un método llamado InitializeComponent() que no debemos tocar. Dicho método lo utiliza nuestro proyecto Net para los dibujos que hacemos.

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

**CODIGO FORMULARIO**

**public** **partial** **class** Form06DateTime : Form

{

**public** Form06DateTime()

    {

**InitializeComponent**();

        //SIEMPRE ESCRIBIREMOS A PARTIR DE ESTA INSTRUCCION

        //NECESITAMOS RECUPERAR LA FECHA ACTUAL

        DateTime fechaActual **=** DateTime**.**Now;

        //DIBUJAMOS EN LA CAJA LA FECHA ACTUAL

**this.**txtFechaActual**.**Text **=** fechaActual**.ToString**();

    }

**private** **void** **chkFormato\_CheckedChanged**(**object** sender, EventArgs e)

    {

        //NECESITAMOS CAPTURAR LA FECHA QUE TENEMOS EN LA CAJA

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

        //EL CONTROL CHECKBOX TIENE UNA PROPIEDAD LLAMADA Checked

        //QUE INDICA SI ESTA CHEQUEADO O NO

**if** (**this.**chkFormato**.**Checked **==** **true**)

        {

            //DIBUJAMOS EN LA CAJA EL FORMATO DE FECHA LARGO

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

        }

**else**

        {

            //DIBUJAMOS EN LA CAJA EL FORMATO DE FECHA CORTO

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

        }

    }

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

    {

        //CAPTURAMOS LA FECHA DE LA CAJA

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

        //AÑADIMOS 5 DIAS A LA FECHA

        //AUNQUE APLIQUEMOS UN METODO A UN OBJETO, DICHO OBJETO NO CAMBIA

        //DE VALOR.  SOLAMENTE SI IGUALAMOS EL OBJETO, CAMBIARA DE VALOR

        //fecha = fecha.AddDays(5);

        //DIBUJAMOS EN LA CAJA INFERIOR EL NUEVO VALOR

        //CAPTURAMOS EL INCREMENTO DE LA CAJA DE TEXTO

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

        //PREGUNTAMOS POR LOS RADIO BUTTONS

**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**();

**this.**lblDayOfWeek**.**Text **=** fecha**.**DayOfWeek**.ToString**();

**this.**lblDayOfYear**.**Text **=** fecha**.**DayOfYear**.ToString**();

    }

}

**SINTAXIS DE BUCLES**

Los bucles son interaciones que se realizan hasta que se cumple una condición que nosotros indiquemos.

Por supuesto, podemos realizar bucles infinitos.

Es un código que hace que una secuencia se repita N veces.

Tenemos dos tipos de bucles:

* **FOR**: Es un bucle numérico que se repite desde un inicio hasta un final. Debemos indicar el inicio, cuando terminará y el incremento que tendrá en cada vuelta del bucle.

for (inicio; final; incremento){

//TODAS LAS ITERACIONES DEL BUCLE

}

Texto

Descripción generada automáticamente

* **WHILE**: Es un bucle peligroso porque podemos hacer que sea infinito. Es un bucle condicional, se mantiene dentro del bucle mientras que la condición del while no cambie.

Debemos indicar, dentro del bucle, que finalizará en algún momento.

while (condición == true){

//LAS CONDICIONES DENTRO DEL BUCLE

//DEBEMOS DE CAMBIAR LA CONDICION PARA QUE FINALICE EL //BUCLE

}

Texto

Descripción generada automáticamente

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

**Nota:** Para comentar código dentro del programa, tenemos una combinación de teclas muy útil.

* **COMENTAR CODIGO**: CONTROL + K + C
* **DESCOMENTAR CODIGO**: CONTROL + K + U

Creamos un nuevo formulario llamado **Form07Bucles**

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

**CODIGO FORMULARIO**

**private** **void** **btnEjemploFor\_Click**(**object** sender, EventArgs e)

{

    //EN UN BUCLE FOR, DEBEMOS INDICAR UNA VARIABLE QUE SERA

    //EL INICIO.  DICHA VARIABLE, SOLAMENTE LA PODEMOS UTILIZAR

    //DENTRO DEL BUCLE { ... }

    //HACEMOS UN BUCLE DE 1 A 7

**string** resultado **=** "";

**for** (**int** i **=** 1; i **<=** 7; i**++**)

    {

        resultado **=** resultado **+** i **+** ", ";

        //resultado += i + ", ";

    }

**this.**txtResultado**.**Text **=** resultado;

}

**private** **void** **btnEjemploWhile\_Click**(**object** sender, EventArgs e)

{

**string** resultado **=** "";

**int** contador **=** 1;

**while** (contador **<=** 7)

    {

        resultado **+=** contador **+** ", ";

        //DEBEMOS HACER QUE SALGA DEL BUCLE ALGUNA VEZ

        contador **+=** 1;

    }

**this.**txtResultado**.**Text **=** resultado;

    //PODEMOS REALIZAR LO MISMO EN UN BUCLE WHILE QUE EN UN FOR

    //POR EJEMPLO, EL CODIGO QUE HEMOS UTILIZADO DE 1 - 7

    //LOS BUCLES WHILE FUNCIONAN CON UNA CONDICION

    //NECESITAMOS "ALGO" QUE INDIQUE NUESTRA CONDICION

    //Y, DENTRO DEL BUCLE, DEBEMOS CAMBIAR DICHA CONDICION

    //bool respuesta = true;

    //while (respuesta == true)

    //{

    //    //SE VA A QUEDAR AQUI DENTRO

    //    this.txtResultado.AppendText("@");

    //    //DEBEMOS CAMBIAR LA CONDICION

    //    //SI LA LONGITUD DEL TEXTO DE LA CAJA LLEGA A 100

    //    //DENTRO DE .Text DE UNA CAJA TENEMOS UNA PROPIEDAD LLAMADA

    //    //.TextLength QUE NOS INDICA LA LONGITUD DE UN TEXTO

    //    int longitud = this.txtResultado.TextLength;

    //    if (longitud == 100)

    //    {

    //        respuesta = false;

    //    }

    //}

}

También podemos utilizar bucles **for** de forma inversa.

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

Vamos a realizar un formulario para mostrar los números pares que tenemos entre 1 y 50.

Mostraremos los datos dentro de una caja, como en el ejemplo anterior.

El símbolo del resto en este lenguaje es el tanto por ciento: **%**

Creamos un nuevo formulario llamado **Form08NumerosPares**

Texto

Descripción generada automáticamente

**SOLUCION 1**

**private** **void** **btnNumerosPares\_Click**(**object** sender, EventArgs e)

{

    //DECLARAMOS UNA VARIABLE PARA MOSTRAR EL RESULTADO DE LOS PARES

**string** resultado **=** "";

**for** (**int** i **=** 1; i **<=** 50; i**++**)

    {

        //DEBEMOS AVERIGUAR SI EL NUMERO ES PAR

        //PREGUNTAMOS SI SU RESTO ENTRE 2 ES CERO

**if** (i **%** 2 **==** 0)

        {

            //PARES

            resultado **+=** i **+** ", ";

        }

    }

**this.**txtResultado**.**Text **=** resultado;

}

Otra posible solución será hacer el recorrido desde **2** e incrementar el bucle de 2 en 2. Así no preguntamos nada dentro del bucle.

**SOLUCION 2**

Imagen que contiene Gráfico de dispersión

Descripción generada automáticamente

Vamos a realizar otra lógica para comprobar la Conjetura de Collatz.

La teoría de Collatz indica lo siguiente:

“Todo número natural será siempre **1** siguiendo dos reglas”:

* Si el número es par, dividimos entre 2.
* Si el número es impar, multiplicamos por tres y sumamos 1.

Por ejemplo, pongamos la secuencia del número 6:

6, 3, 10, 5, 16, 8, 4, 2, 1

Creamos un nuevo formulario llamado **Form09Collatz**

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

**CODIGO FORMULARIO**

**private** **void** **btnCollatz\_Click**(**object** sender, EventArgs e)

{

**string** resultado **=** "";

    //NECESITAMOS EL NUMERO INICIAL

**int** numero **=** **int.Parse**(**this.**txtNumero**.**Text);

    //NUESTRA CONDICION SERA QUE MIENTRAS QUE NUMERO NO SEA 1

**while** (numero **!=** 1)

    {

        //COMPROBAMOS SI EL NUMERO ES PAR

**if** (numero **%** 2 **==** 0)

        {

            //PAR

            numero **=** numero **/** 2;

        }

**else**

        {

            //IMPAR

            numero **=** numero **\*** 3 **+** 1;

        }

        resultado **+=** numero **+** ", ";

    }

**this.**txtResultado**.**Text **=** resultado;

}

Vamos a crear un nuevo formulario llamado **Form10PracticasBucles**

En el mismo formulario, dibujamos lo siguiente:

* 1. Tabla Multiplicar.

Tendremos una caja de texto para pedir un número.

Tendremos un botón para representar la tabla de multiplicar.

Tendremos una caja para mostrar el resultado.

Al pulsar el botón, mostramos los resultados de la tabla de multiplicar del número que el usuario ha introducido.

5,10,15,20,…50

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

**private** **void** **btnTablaMultiplicar\_Click**(**object** sender, EventArgs e)

{

    //CAPTURAMOS EL NUMERO DE LA CAJA

**int** numero **=** **int.Parse**(**this.**txtNumero**.**Text);

**string** resultado **=** "";

    //REALIZAMOS UN BUCLE DE 1 A 10

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

    {

**int** operacion **=** numero **\*** i;

        resultado **+=** operacion **+** ", ";

    }

**this.**txtResultado**.**Text **=** resultado;

}

**VERSION 2**

**private** **void** **btnTablaMultiplicar\_Click**(**object** sender, EventArgs e)

{

    //PODEMOS COMPROBAR SI LA CAJA DE TEXTO ESTA VACIA O NO

**if** (**this.**txtNumero**.**Text **==** "")

    {

        MessageBox**.Show**("El número no puede estar vacío");

    }

**else**

    {

        //RECUPERAMOS EL TEXTO DE LA CAJA

**string** textoCaja **=** **this.**txtNumero**.**Text;

        //DECLARAMOS EL NUMERO PARA LA CAJA

**int** numero;

        //MEDIANTE TRYPARSE, PREGUNTAMOS SI PUEDE HACER LA CONVERSION O NO

**if** (**int.TryParse**(textoCaja, **out** numero))

        {

**string** resultado **=** "";

            //REALIZAMOS UN BUCLE DE 1 A 10

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

            {

**int** operacion **=** numero **\*** i;

                resultado **+=** operacion **+** ", ";

            }

**this.**txtResultado**.**Text **=** resultado;

        }

**else**

        {

            MessageBox**.Show**("Solo números en la caja");

        }

    }

}

* 1. Números impares.

Tendremos una caja para pedir el inicio y el final de los números impares.

Debemos mostrar los números impares entre dicho inicio y final.

Tendremos un botón para probar la ejecución y dibujamos los datos en Resultado.

Modificamos este ejemplo y comprobamos que tengamos texto en las cajas.

Comprobamos que el inicio no es mayor al final.

* Google: Podríamos comprobar si el usuario ha escrito números o no.

<https://ironpdf.com/es/blog/net-help/csharp-tryparse/>

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

**CODIGO FORMULARIO**

**private** **void** **btnNumerosImpares\_Click**(**object** sender, EventArgs e)

{

**int** inicio **=** **int.Parse**(**this.**txtInicio**.**Text);

**int** fin **=** **int.Parse**(**this.**txtFin**.**Text);

**if** (inicio **>** fin)

    {

        MessageBox**.Show**("El inicio debe ser inferior al final.");

    }

**else**

    {

        //QUEREMOS LOS NUMEROS IMPARES

        //VAMOS A PREGUNTAR SI EL NUMERO DE INICIO ES PAR

**if** (inicio **%** 2 **==** 0)

        {

            //EL INICIO ES PAR

            //LE SUMAMOS UNO PARA QUE SEA IMPAR

            inicio **+=** 1;

        }

        //YA SABEMOS QUE ES IMPAR, PUES REALIZAMOS UN BUCLE DESDE EL INICIO

        //Y CON UN INCREMENTO DE 2 EN 2.

**string** resultado **=** "";

**for** (**int** i **=** inicio; i **<=** fin; i **+=** 2)

        {

            resultado **+=** i **+** ", ";

        }

**this.**txtResultado**.**Text **=** resultado;

    }

}

## CLASE CHAR

Es un tipo de dato que nos va a permitir almacenar un solo carácter.

Se representa mediante la comilla simple.

char letra = 'k';

Sus métodos son de tipo **static**, es decir, que no utilizan el objeto (letra) sino que se llaman directamente desde la clase.

La mayoría de sus métodos son herramientas para saber el tipo de carácter que tenemos almacenado.

* char.IsLetter(carácter): Devuelve si el carácter es una letra
* char.IsNumber(carácter): Devuelve si el carácter es de tipo numérico
* IsDigit(carácter): Devuelve si el carácter es un digito de 1-9
* IsLetterOrDigit(): Indica si es una letra o un dígito
* IsWhiteSpace(): Carácter en espacio
* IsSymbol(): Indica si el carácter es un símbolo o no
* IsPunctuation(): Indica si el carácter es un símbolo de puntuación.
* IsUpper(): Indica si el carácter es en mayúscula
* IsLower(): Indica si el carácter es en minúscula.
* ToLower(): Convierte el carácter a minúscula
* ToUpper(): Convierte el carácter a mayúscula

Vamos a realizar un formulario en el que recorreremos todos los símbolos ASCII del teclado. (255)

Lo que haremos será preguntar por el tipo de carácter recorrido y lo separamos en cajas.

Creamos un nuevo formulario llamado **Form11ClaseChar**

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

**CODIGO FORMULARIO**

**private** **void** **btnIniciar\_Click**(**object** sender, EventArgs e)

{

    //RECORREMOS TODOS LOS CARACTERES ASCII DE 0 - 255

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

    {

        //CONVERTIMOS CADA POSICION i DEL BUCLE

        //EN 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 que nos permite representar textos dentro de un programa.

En realidad, un string es un conjunto, un conjunto de char/letras.

**Norma**: Todo en Net comienza en cero, es decir, cada carácter de un texto tiene una posición. La primera siempre será cero.

string cadena = "VISUAL STUDIO";

0 🡪 V

1 🡪 I

2 🡪 S

…

Por supuesto, un string diferencia mayúsculas de minúsculas.

Para acceder a la posición de un carácter, se utiliza una propiedad que se llama **indizada**. Simplemente es una propiedad que no se escribe y se accede a ella directamente mediante los corchetes []

cadena[0] 🡪 V

cadena[1] 🡪 I

cadena[2] 🡪 S

…

METODOS Y 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

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

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, Word

Descripción generada automáticamente con confianza media

* **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, Word

Descripción generada automáticamente con confianza media

* **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.

Queremos validar un mail. Creamos un formulario llamado **Form12ValidarEmail**

Mediante los métodos de string, debemos indicar si un mail es válido o no.

* Que exista una @
* Que exista un punto
* Que no exista una @ al inicio
* Que no exista una @ al final
* Que exista un punto después de la @
* Que solamente exista una @
* Dominio de 2 a 4 caracteres (es, com, org)

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

Descripción generada automáticamente

Interfaz de usuario gráfica, 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.**lblResultado**.**Text **=** "No existe @ en el mail";

    }**else** **if** (email**.IndexOf**(".") **==** **-**1)

    {

**this.**lblResultado**.**Text **=** "Debe existir un punto";

    }**else** **if** (email**.StartsWith**("@") **==** **true**)

    {

**this.**lblResultado**.**Text **=** "@ al inicio del mail";

    }**else** **if** (email**.EndsWith**("@") **==** **true**)

    {

**this.**lblResultado**.**Text **=** "@ al final del email";

    }**else** **if** (email**.IndexOf**("@") **!=** email**.LastIndexOf**("@"))

    {

**this.**lblResultado**.**Text **=** "Existe más de una @ en el mail";

    }

**else** **if** (email**.LastIndexOf**(".") **<** email**.IndexOf**("@"))

    {

**this.**lblResultado**.**Text **=** "Debemos tener un punto despues de la @";

    }

**else**

    {

        //CAPTURAMOS LA POSICION DEL ULTIMO PUNTO

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

        //DEBEMOS QUEDARNOS CON EL DOMINIO, PERO SIN EL PUNTO

        //email@dominio.com

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

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

        {

**this.**lblResultado**.**Text **=** "Mail correcto";

        }

**else**

        {

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

        }

    }

}

Vamos a realizar un formulario para sumar todos los caracteres numéricos que escribamos en una caja de texto.

1234 🡪 La suma es **10**

Creamos un nuevo formulario llamado **Form13SumaNumerosString**

Esta conversión, lo que hace es recuperar el código ASCII de un carácter

Imagen que contiene Texto

Descripción generada automáticamente



Para convertir un char a un int, debemos convertir el char a STRING y luego aplicar **Parse**

Texto

Descripción generada automáticamente

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

**CODIGO FORMULARIO**

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

{

**string** textoNumeros **=** **this.**txtNumeros**.**Text;

    //1234

    //DECLARAMOS LA SUMA

    //TODA VARIABLE DEBE SER INICIALIZADA/INSTANCIADA

**int** suma **=** 0;

    //RECORREMOS LA LONGITUD DEL TEXTO

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

    {

        //CAPTURAMOS CADA CARACTER DEL TEXTO EN SU POSICION i

**char** caracter **=** textoNumeros[i];

        //CONVERTIMOS CADA CARACTER A NUMERO (conversion explicita)

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

        //INCREMENTAMOS LA SUMA CON CADA NUMERO

        suma **+=** numero;

    }

**this.**lblResultado**.**Text **=** "La suma de " **+** textoNumeros

**+** " es " **+** suma**.ToString**();

}

Vamos a visualizar clases que no tenemos directamente dentro de nuestro entorno gráfico.

Cada conjunto de clases está dentro de lo que se denomina un **namespace**

Un espacio de nombres es un acceso directo para poder utilizar determinadas clases que son de NET, pero no están en nuestra clase directamente.

Por ejemplo, tenemos clases que nos permiten acceder a ficheros. Las clases de ficheros están dentro del **namespace** System.IO.

Si queremos utilizar las clases de ficheros, tendremos que hacer un **using** de **System.IO**

Cualquier formulario, por defecto, tiene unos **namespace**, pero podemos agregar más para traer más clases que necesitemos.

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Si necesitamos trabajar con ficheros en un formulario, necesitamos el **using** **System.IO**

Vamos a realizar una aplicación que se encargará de medir el tiempo que tarda un proceso en realizarse.

Con las clase string, vamos a invertir un texto. Normalmente, el texto no tarda nada en hacer ese proceso, pero que sucede si nos pasamos de 65.000 caracteres que es cuando le empieza a costar a string hacer las cosas.

Una vez que tengamos la lógica de invertir texto con String.

Vamos a realizar lo mismo con una clase especializada en grandes cadenas de texto. Dicha clase se llama **StringBuilder**

Para medir los tiempos del proceso vamos a utilizar una clase llamada **Stopwatch** que está en el **namespace** **System.Diagnostics**

Utilizaremos también un control nuevo llamado **RichTextBox**. Es como una caja, pero permite tener más caracteres que una caja de texto.

Creamos un nuevo formulario llamado **Form14StringBuilder**

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

Descripción generada automáticamente

**CODIGO FORMULARIO**

**private** **void** **btnInvertirString\_Click**(**object** sender, EventArgs e)

    {

        //NECESITAMOS UN CRONOMETRO PARA MEDIR EL TIEMPO

        Stopwatch krono **=** **new** Stopwatch();

**string** cadena **=** **this.**txtTexto**.**Text;

**int** longitud **=** cadena**.**Length;

        //INICIAMOS EL CRONOMETRO

        krono**.Start**();

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

        {

            //HOLA

            //RECUPERAMOS LA ULTIMA LETRA DEL TEXTO

**char** letra **=** cadena[longitud **-** 1];

            //ELIMINAMOS LA ULTIMA LETRA DEL TEXTO

            //HOL

            cadena **=** cadena**.Remove**(longitud **-** 1, 1);

            //INSERTAMOS EN LA CADENA EN LA POSICION DEL BUCLE i

            //AHOL

            cadena **=** cadena**.Insert**(i, letra**.ToString**());

        }

        //DETENEMOS EL CRONOMETRO

        krono**.Stop**();

        //TIENE UN PAR DE PROPIEDADES DE TIPO TIME QUE NOS DICEN

        //EL TIEMPO QUE HA TARDADO EN SEGUNDOS...

**this.**lblTiempo**.**Text **=** "Segundos: " **+** krono**.**Elapsed**.**Seconds

**+** ", Milisegundos: " **+** krono**.**ElapsedMilliseconds;

**this.**txtTexto**.**Text **=** cadena;

    }

**private** **void** **btnInvertirStringBuilder\_Click**(**object** sender, EventArgs e)

    {

        Stopwatch krono **=** **new** Stopwatch();

        //DEBEMOS INSTANCIAR UN OBJETO DE LA CLASE STRINGBUILDER

        StringBuilder cadena **=** **new** StringBuilder();

        //AÑADIMOS AL STRINGBUILDER LA CADENA

        cadena**.Append**(**this.**txtTexto**.**Text);

**int** longitud **=** cadena**.**Length;

        krono**.Start**();

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

        {

            //RECUPERAMOS LA ULTIMA LETRA

**char** letra **=** cadena[longitud **-** 1];

            //ELIMINAMOS EL ULTIMO CARACTER

            cadena **=** cadena**.Remove**(longitud **-** 1, 1);

            //INSERTAMOS EL CARACTER EN LA POSICION DE i

            cadena **=** cadena**.Insert**(i, letra);

        }

        //DETENEMOS EL CRONOMETRO

        krono**.Stop**();

**this.**lblTiempo**.**Text **=** "Segundos: " **+** krono**.**Elapsed**.**Seconds

**+** ", Milisegundos: " **+** krono**.**ElapsedMilliseconds;

**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 **Form15Isbn**

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.

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

Descripción generada automáticamente

**CODIGO FORMULARIO**

**private** **void** **btnComprobarIsbn\_Click**(**object** sender, EventArgs e)

{

    string isbn **=** **this.**txtIsbn**.**Text;

**if** (isbn**.**Length **!=** 10)

    {

        MessageBox**.Show**("El número ISBN debe tener 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";

        }

**else**

        {

**this.**lblResultado**.**Text **=** "El número introducido no es ISBN correcto";

        }

    }

}

## CONJUNTOS DE OBJETOS

Un string no deja de ser un conjunto de letras con una posición para cada letra.

Net utiliza conjuntos para todos los elementos del programa. Dichos conjuntos siempre estarán establecidos por un índice.

A Net, no le importa si es un conjunto de letras, de botones, de cajas de texto, de personas o de cualquier clase que hablemos.

Todos los conjuntos se manejan igual, es decir, accedemos al conjunto y tratamos cada elemento de forma individual.

Tenemos dos grandes grupos cuando hablamos de conjuntos en Net.

* **Arrays:** Es un conjunto de objetos definidos por un índice. Es un conjunto **estático**, es decir, no puede crecer ni decrecer, según creamos el conjunto se mantiene con los mismos elementos.

Por ejemplo, si creamos un Array de 5 elementos, siempre tendrá espacio en la memoria para 5 huecos, aunque dichos huecos estén vacíos. Esos elementos, aunque no los utilicemos, consumen memoria. Además, para la programación, al poder tener elementos vacíos, podemos tener espacios nulos.

* **Colecciones:** Es un conjunto de objetos definidos por un índice. Es un conjunto dinámico. A medida que vayamos añadiendo elementos o borrando elementos, el conjunto libera memoria y quita posiciones.

El tamaño de la colección siempre nos dirá el número de elementos real que contiene.

Todas las colecciones siempre van a tener los mismos métodos, aunque algunas colecciones podrían tener más. Esto es un concepto de POO llamado abstracción.

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

**Propiedades y métodos de una colección**

* Count: Número de elementos de una colección
* Add(elemento): Agrega un nuevo elemento a la colección
* Remove(elemento): Elimina un elemento de la colección enviando el propio objeto de la colección.
* RemoveAt(índice): Elimina un elemento de la colección enviando el índice/posición del elemento en la colección
* Clear(): Borra todos los elementos de la colección.
* IndexOf(elemento): Busca un elemento en la colección y nos devuelve su posición. Si no lo encuentra, nos devuelve -1
* Contains(elemento): Devuelve true/false por si un elemento existe en la colección.

Vamos a comenzar con colecciones gráficas. Tenemos un control llamado **ListBox** que contiene en su interior una serie de elementos.

Este ListBox tiene una propiedad, llamada **Items** que es la colección de elementos de su interior.

Propiedades:

* **SelectedIndex**: Nos devuelve el índice del elemento seleccionado
* **SelectedItem**:Nos devuelve el elemento seleccionado
* **SelectionMode**: Permite cambiar el tipo de selección de Simple a múltiple.
* **SelectedItems**: Es una colección con el conjunto de elementos seleccionados
* **SelectedIndices:** Es una colección con el conjunto de índices seleccionados.

Creamos un nuevo formulario llamado **Form16ColeccionGraficaListBox**

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

**CODIGO FORMULARIO**

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

{

**string** elemento **=** **this.**txtNuevoElemento**.**Text;

**this.**lstElementos**.**Items**.Add**(elemento);

}

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

{

    //PARA ELIMINAR UN ELEMENTO DE UNA COLECCION, TENEMOS 2 OPCIONES

    //1) BORRAR POR EL OBJETO.  SI TENEMOS REPETIDOS, SE COMPORTA RARO

    //2) BORRAR POR EL INDICE/POSICION

    //PRIMERO VAMOS A BORRAR POR EL OBJETO DE LA COLECCION

    //RECUPERAMOS EL ELEMENTO SELECCIONADO DE LA COLECCION

    //string elementoSeleccionado = this.lstElementos.SelectedItem.ToString();

    ////ELIMINAMOS EL OBJETO DE LA COLECCION (Remove(elemento))

    //this.lstElementos.Items.Remove(elementoSeleccionado);

    //RECUPERAMOS EL INDICE DEL ELEMENTO SELECCIONADO

**int** indiceSeleccionado **=** **this.**lstElementos**.**SelectedIndex;

    //ELIMINAMOS POR INDICE (RemoveAt(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)

{

    //EL INDICE PUEDE CAMBIAR A -1 PORQUE ESTAMOS ELIMINANDO ELEMENTOS

    //DEBERIAMOS SIEMPRE PREGUNTAR SI TENEMOS ELEMENTOS SELECCIONADOS

    //EN ESTE EVENTO

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

    {

        //TENEMOS UN ELEMENTO SELECCIONADO

**this.**lblIndexSeleccionado**.**Text **=** "Indice: "

**+** **this.**lstElementos**.**SelectedIndex;

**this.**lblItemSeleccionado**.**Text **=** "Item: "

**+** **this.**lstElementos**.**SelectedItem;

    }

}

### BUCLES DE REFERENCIA

Cuando trabajamos con conjuntos, existe un bucle que nos permite recorrer dichos conjuntos con una variable de referencia, es decir, una variable que vaya apuntando a cada elemento del conjunto.

Esto es solamente una herramienta, podemos utilizarla también con un bucle contador for.

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Esta línea indica una variable de referencia, es decir, hace referencia a cada elemento de la colección.



Tenemos un bucle llamado **foreach** que nos permite declarar una variable de referencia y recorrer los datos de la colección en una misma instrucción.

Captura de pantalla de un celular

Descripción generada automáticamente

Vamos a realizar un formulario para sumar números.

Lo que haremos será almacenar números en un Listbox y, al pulsar un botón, haremos la suma de todos los números, los pares y también la de los impares haciendo mediante un bucle de referencia.

Creamos un nuevo formulario llamado **Form17SumarNumerosListBox**

Interfaz de usuario gráfica, Aplicación, Word

Descripción generada automáticamente

**CODIGO FORMULARIO**

**private** **void** **btnResumen\_Click**(**object** sender, EventArgs e)

{

**int** sumaPares **=** 0;

**int** sumaImpares **=** 0;

**int** sumaTotal **=** 0;

    //RECORREMOS CON UN BUCLE FOREACH TODOS LOS NUMEROS

    //int QUE CONTIENE LA COLECCION GRAFICA

**foreach** (**int** numero **in** **this.**lstNumeros**.**Items)

    {

        sumaTotal **+=** numero;

**if** (numero **%** 2 **==** 0)

        {

            sumaPares **+=** numero;

        }

**else**

        {

            sumaImpares **+=** numero;

        }

    }

**this.**txtSumaImpares**.**Text **=** sumaImpares**.ToString**();

**this.**txtSumaPares**.**Text **=** sumaPares**.ToString**();

**this.**txtSumaTotal**.**Text **=** sumaTotal**.ToString**();

}

## METODOS DE CLASE

Todas las clases, tienen métodos y propiedades, por ejemplo, el método **Add()** pertenece a una colección y, la propiedad **BackColor**, pertenece a un Control.

Un método es un conjunto de acciones sobre una clase determinada. Tenemos métodos ya creados y podemos crear métodos nuestros en cualquier clase, sea una clase gráfica o no.

Un método puede ser diferenciado en dos/tres categorías.

* 1. Métodos de acción: **void**

Estos métodos son un conjunto de acciones y nada más. Realizan su acción y simplemente se invocan.

Ejemplo de un método de acción **void**:

void LimpiarCajas() {

this.txt1.Text = “”;

this.txt2.Text = “”;

this.txt3.Text = “”;

}

Texto

Descripción generada automáticamente

Podemos realizar la llamada al método desde varios lugares o acciones. Lo bueno que tienen los métodos es que podemos llamarlos desde múltiples sitios.

La llamada a un método void se realiza así:

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 lugar dónde han sido invocados. Siempre deben devolver un valor al terminar las intrucciones.

Sintaxis:

TipoDatoDevolver NombreMetodo() {

//ACCIONES

**return ValorTipoDato;**

}

Todos los métodos pueden recibir parámetros dentro de sus paréntesis.

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Para realizar la llamada al método, debemos almacenar lo que devuelve el método en alguna variable:

Texto

Descripción generada automáticamente con confianza baja

* 1. Métodos de Evento. Los métodos de evento son un conjunto de acciones que se ejecutan al realizar “algo” sobre un Control, es decir, un Click como hemos estado haciendo todo el tiempo del curso.



Los métodos de evento reciben siempre dos variables:

* **object sender**: Es el objeto que ha realizado la llamada al método.
* **EventArgs e:** La variable de argumentos del método. Dependiendo del evento, podría traer información.

**Truco:** Si la variable es **EventArgs**, no trae nada, solamente investigamos si es diferente a EventArgs.



En el momento de tener métodos de evento, podemos asociar varios controles a un mismo método.

Es decir, múltiples botones que realicen la misma acción al ser pulsados.

A esto se le llama métodos compartidos.

Para compartir un evento con un control, se hace con la siguiente sintaxis:

**Control.Accion += NombreEvento;**

Creamos un nuevo formulario llamado **Form18EventosMetodos**

Cuando realizamos doble click sobre un control, lo que hacemos es recuperar su evento por defecto. Pero un control, tiene muchos más eventos, por ejemplo, cuando pasamos por encima el ratón o cuando pulsamos teclas sobre él.

Interfaz de usuario gráfica

Descripción generada automáticamente con confianza baja

Si necesitamos recuperar más eventos de un control, se hace sobre la ventana de **Propiedades**. Dentro de dicha ventana, tenemos un icono de un **rayo** y podemos recuperar cualquier evento al pulsar sobre él en la ventana de propiedades.

Tabla

Descripción generada automáticamente

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

**CODIGO FORMULARIO**

**public** Form18EventosMetodos()

{

**InitializeComponent**();

    //ASOCIAMOS EL CLICK DE CADA BOTON A EL METODO

    //DE EVENTO BotonPulsado

**this.**button1**.**Click **+=** **BotonPulsado**;

**this.**button2**.**Click **+=** **BotonPulsado**;

**this.**button3**.**Click **+=** **BotonPulsado**;

}

//VAMOS A CREAR UN METODO PROPIO DE EVENTO PARA QUE LOS

//TRES BOTONES LO LEAN AL SER PULSADOS

**void** **BotonPulsado**(**object** sender, EventArgs e)

{

    //Y SI NECESITAMOS SABER QUE BOTON HEMOS PULSADO?

    //QUIERO DIFERENCIAR EL BOTON PULSADO

    //LA SOLUCION ESTA EN sender

    //sender ES EL CONTROL QUE HA REALIZADO LA LLAMADA

    //EN ESTE EJEMPLO, TODOS SON BOTONES

    //CAPTURAMOS EL BOTON PULSADO

    Button boton **=** (Button)sender;

    boton**.**BackColor **=** Color**.**Yellow;

    //MessageBox.Show("Pulsado");

}

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

{

}

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

{

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

}

Todo depende del evento y de la acción que necesitamos realizar

Pongamos el siguiente ejemplo:

Tenemos dos cajas de texto y solamente queremos filtrar un contenido sobre ellas.

Tenemos dos opciones:

* 1. Cuando el usuario haga un Click, recorremos cada carácter con string y evaluamos.
  2. Si solo queremos letras, le dejamos escribir solo letras. Si solo queremos números, le dejamos escribir solo números.

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Para esa acción/evento que necesitamos ahora mismo, lo haremos con un evento de Key, es decir, cuando el usuario esté escribiendo sobre las cajas.

Según escriba el usuario, lo que haremos será evaluar el contenido escrito.

En el evento TextChanged podría hacerlo, debería capturar lo que tenemos en la caja actual y evaluarlo.

Imagen que contiene Texto

Descripción generada automáticamente

En lugar de hacerlo en ese evento, podemos hacerlo directamente cuando el usuario pulse una tecla.

Tabla

Descripción generada automáticamente

Texto

Descripción generada automáticamente

Texto

Descripción generada automáticamente

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

Cada botón tendrá un número escrito.

Al pulsar un botón, lo que haremos será recuperar su número y Sumarlo en un Label, es decir, cada vez que pulsemos un botón, iremos sumando.

Creamos un nuevo formulario llamado **Form19SumarBotonesNumeros**

Captura de pantalla de un celular

Descripción generada automáticamente

**CODIGO FORMULARIO**

**public** **partial** **class** Form19SumarBotonesNumeros : Form

{

**public** Form19SumarBotonesNumeros()

    {

**InitializeComponent**();

**this.**button1**.**Click **+=** **SumarNumeros**;

**this.**button2**.**Click **+=** **SumarNumeros**;

**this.**button3**.**Click **+=** **SumarNumeros**;

**this.**button4**.**Click **+=** **SumarNumeros**;

    }

    //CREAMOS UN METODO DE EVENTO

**void** **SumarNumeros**(**object** sender, EventArgs e)

    {

        //RECUPERAMOS EL TEXTO DEL LABEL PARA LA SUMA

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

        //CAPTURAMOS EL BOTON QUE HA PULSADO EL USUARIO (sender)

        Button boton **=** (Button)sender;

        //RECUPERAMOS EL NUMERO DEL BOTON PULSADO

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

        //REALIZAMOS LA SUMA

        suma **+=** numero;

        //DIBUJAMOS EN EL LABEL LA SUMA

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

    }

}

Vamos a realizar una versión 2 sobre el mismo ejemplo.

Imaginemos que tenemos 100 botones en el formulario, tendríamos que estar, dentro del constructor, copiando y pegando para asociar todos los controles Button.

Texto

Descripción generada automáticamente

En lugar de estar haciendo eso, existe una colección llamada **Controls** que nos permite recorrer todos los controles que contenga un formulario de forma dinámica.

La colección Controls contiene en su interior objetos de la clase **Control**

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

Un objeto de la clase control es cualquier objeto visual de un formulario: Labels, botones, TextBox…

Vamos a cambiar ahora mismo el color de fondo de todos los controles que contiene el formulario.

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Para nosotros, en nuestro lógica actual, solamente nos interesan los botones.

Si ejecutamos el bucle, va a recorrer todos los controles. Debemos preguntar qué tipo de control estamos recorriendo en la llamada.

Para preguntar por un tipo de Clase, se utiliza el operador **is**

if (Objeto is Clase) { … }

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

De forma que, solamente nos quedaremos con los botones y agregaremos dichos botones a un método de evento compartido (SumarNumeros)

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Lo siguiente que vamos a realizar es otra versión más.

Cuando el formulario arranque, lo que haremos será poner números aleatorios dentro de cada botón.

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

Dicha clase tiene tres métodos:

* 1. NextBytes(): Rellena un Array de Bytes con números aleatorios.
  2. NextDouble(): Devuelve un número double entre 0 y 1 sin llegar a ser 0 ni 1 nunca. Devuelve 0.99 o 0.11 o 0.33
  3. Next(mínimo, máximo): Devuelve un número entero entre el número mínimo y el máximo.

Modificamos el código anterior y generamos números aleatorios:

Imagen de la pantalla de un celular con letras

Descripción generada automáticamente con confianza baja

**CODIGO FORMULARIO**

**public** **partial** **class** Form19SumarBotonesNumeros : Form

{

**public** Form19SumarBotonesNumeros()

    {

**InitializeComponent**();

        //DECLARAMOS UN OBJETO RANDOM

        Random random **=** **new** Random();

        //RECORREMOS TODOS LOS CONTROLES DEL FORMULARIO MEDIANTE

        //LA COLECCION CONTROLS

**foreach** (Control c **in** this**.**Controls)

        {

**if** (c **is** Button)

            {

                //GENERAMOS UN NUMERO ALEATORIO POR CADA BOTON

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

                c**.**Text **=** aleatorio**.ToString**();

                //ASOCIAMOS DINAMICAMENTE EL BOTON AL METODO DE EVENTO

                c**.**Click **+=** **SumarNumeros**;

            }

        }

    }

    //CREAMOS UN METODO DE EVENTO

**void** **SumarNumeros**(**object** sender, EventArgs e)

    {

        //RECUPERAMOS EL TEXTO DEL LABEL PARA LA SUMA

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

        //CAPTURAMOS EL BOTON QUE HA PULSADO EL USUARIO (sender)

        Button boton **=** (Button)sender;

        //RECUPERAMOS EL NUMERO DEL BOTON PULSADO

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

        //REALIZAMOS LA SUMA

        suma **+=** numero;

        //DIBUJAMOS EN EL LABEL LA SUMA

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

    }

}

**PRACTICAS JUEVES**

**TABLA MULTIPLICAR CON COLECCIÓN Y BOTONES**

* Tendremos múltiples botones en el diseño. Dichos botones no es necesario ni que los nombremos ni que tengan números en su interior.
* Al iniciar la aplicación, generamos números aleatorios para cada botón.
* Al pulsar cada botón, lo que haremos será mostrar su tabla de multiplicar dentro de un Control ListBox.

Interfaz de usuario gráfica

Descripción generada automáticamente

**NUMERO NARCISISTA**

Crear un formulario llamado **FormNarcisista** y resolver el siguiente problema matemático.

*Un número Narcisista es aquel que es igual a la suma de cada uno de sus dígitos elevados a la "n" potencia, estableciendo como potencia la longitud del número.*

Por ejemplo: El número **153** es narcisista

La longitud del número 153 es **3**, utilizamos este número como Potencia

Si separamos cada uno de sus caracteres y los elevamos a la potencia

* 1 elevado a 3: **1**
* 5 elevado a 3: **125**
* 3 elevado a 3: **27**

La suma de 1 + 125 + 27: **153**

Realizar una aplicación para comprobar la teoría. Para realizar la potencia de un número en Net, utilizamos **Math.Pow(numero, potencia)** y nos devuelve un **double**

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

Descripción generada automáticamente

**SOLUCION**

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

{

//RECUPERAMOS EL NUMERO DE LA CAJA DE TEXTO

//DEBEMOS RECUPERARLO COMO STRING

string numeroTexto = this.txtNumero.Text;

//Se dice que un numero narcisista es aquel que es igual a

//la suma de cada uno de sus dígitos elevados a la "n" potencia

int longitud = numeroTexto.Length;

int suma = 0;

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

char letraNumero = numeroTexto[i];

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

double potencia = Math.Pow(numero, longitud);

suma += (int)potencia;

}

if (suma.ToString() == numeroTexto)

{

this.lblResultado.Text = "El número "

+ numeroTexto + " es narcisista";

}

else

{

this.lblResultado.Text = "Este número no es narcisista";

}

}

**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 operación:

**( nº DNI - (PARTE ENTERA(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 |

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

***CODIGO FORMULARIO***

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

{

string dni = this.txtDni.Text;

//RECUPERAMOS LA LETRA DEL DNI

char letraDni = dni[dni.Length - 1];

//SI LO PENSAMOS CON LETRA...

string soloNumeros =

dni.Substring(0, dni.Length - 1);

int dniNumber = int.Parse(soloNumeros);

int resultado =

(dniNumber - ((dniNumber / 23) \* 23));

string secuenciaLetras = "TRWAGMYFPDXBNJZSQVHLCKET";

char letraValidacion = secuenciaLetras[resultado];

if (letraDni == letraValidacion)

{

this.lblResultado.Text = "DNI correcto!!!";

}

else

{

this.lblResultado.Text = "Incorrecto, su letra es "

+ letraValidacion;

}

}

**VALIDAR CODIGO EAN**

El codigo EAN (European Article Number) es un sistema de código de barras para asignar un número único a cada producto. Los códigos más comunes tienen 8 o 13 dígitos, especialmente 13 (sistemas conocidos como EAN8 y EAN13). En ellos van codificados el pais de origen del producto, la empresa y el propio producto. El último de los dígitos es un dígito de control para evitar errores de transcripción.

El algoritmo para comprobar que un código EAN8 o EAN13 ha sido transcrito correctamente es extremadamente sencillo.

Podemos describirlo algorítmicamente de esta manera:

Comprobar que el código tiene 8 o 13 dígitos. De no ser así, no es correcto.

1. Sumar los dígitos de lugares pares por un lado y los de los impares por otro, pero sin incluir el último dígito.
2. Si el código es EAN13, multiplicar la suma de los pares por 3.
3. Si el código es EAN8, es la suma de los impares la que se multiplica por 3.
4. Sumar el resultado de los pares y el de los impares y hallar el resto de la división por 10.
5. Realizar la operación 10 menos ese resto y ese es el dígito de control.
6. Si como resultado sale 10, entenderemos que el dígito de control es 0.
7. Comprobar que el dígito de control que hemos calculado y el último dígito del código EAN coinciden

Por ejemplo, para validar el código EAN8 "12345678"  (Obviamente es inventado)

1. ImageSeparar el dígito de control. Nos quedamos con "1234567" y "8"
2. Sumar pares:   sumapares=2+4+6=12
3. Sumar impares: sumaimpares=1+3+5+7=16
4. Como es EAN8, multiplicamos los impares por 3.
5. sumaimpares=16\*3=48
6. Sumar el resultado de pares e impares:  12+48=60
7. Hallar el resto de la division por 10:  60 mod 10 = 0
8. Hacer 10-resto:  10-0=10
9. Si el resultado del paso 8 es 10, el dígito de control es 0.
10. Comparar el dígito de control que hemos calculado con el que tenía el código: Nos sale 0 y el código tenía un 8. Es incorrecto.

* **Realizar la aplicación para validar un código EAN de 8 caracteres.**

### PROGRAMACION ORIENTADA A OBJETOS

Todos los lenguajes son iguales en el momento de trabajar a POO. Este concepto nos permite crear nuestras clases y modelarlas a nuestro gusto.

Es útil sobre todo cuando estamos leyendo de algún lugar (bbdd, servicios) y no tenemos ningún soporte para guardar lo que estamos leyendo.

Por ejemplo, si estamos leyendo nombres de una base de datos, pues tenemos la clase string.

Si estamos leyendo edades de una base de datos, tenemos la clase int.

Pero, si estamos leyendo los dos a la vez, como una Persona (por ejemplo) no tenemos ningún tipo de dato que sea string e int a la vez, tendríamos que crearlo nosotros.

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

1. **Abstracción**: Todas las clases tienen una misma “familia”, es decir, tienen una raíz y unos métodos comunes. Podemos abstraernos y utilizar una misma familia que tengan métodos comunes.
2. **Encapsulación**:Este concepto tiene que ver con la visibilidad de los elementos entre clases. No visualizaremos lo que no deseamos que sea visible para el uso de nuestra clase.
3. **Polimorfismo**:Un objeto puede tener múltiples formas para un método.
4. **Herencia**: Un objeto hereda de otro objeto sus propiedades y métodos y los podría mejorar e implementar.

Las clases pueden ser creadas dentro del mismo proyecto, pero eso no quiere decir que podamos utilizarlas en otros proyectos.

El concepto de crear clases implica poder reutilizarlas independientemente al proyecto en el que estemos trabajando.

Por ejemplo, si tenemos un proyecto de **Windows** y creamos una clase **Persona**, dicha clase persona no podría utilizarla en un proyecto de **Android**.

La forma de trabajar es tener un proyecto de clases NO gráficas y utilizarlas en cualquier otro proyecto.

Este tipo de proyectos se llaman **Class Library (Librería de clases)**.

**MODIFICADORES DE ACCESO**

Este concepto tiene que ver con la encapsulación, es decir, la visibilidad de los elementos dentro de una clase.

Tenemos una serie de palabras clave que indican el ámbito de visibilidad de una variable dentro de una clase.

1. public: Ámbito de acceso total, es decir, visibilidad total para la variable.
2. private: Este ámbito hace que la variable declarada en una clase, solamente sea visible para dicha clase.
3. internal: Ámbito solamente para proyectos. Las variables declaradas como internal solamente serán visibles entre las clases del mismo proyecto, no para otros proyectos.
4. protected: Es un ámbito para la herencia. Las variables solamente serán visibles para las clases que hereden directamente.

Vamos a comenzar a crear un nuevo proyecto sobre la solución.

Una solución, permite tener múltiples proyectos abiertos a la vez.

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Cuando creamos un nuevo proyecto, se crea un proyecto con un **namespace** para acceder a las clases de dicho proyecto.

El namespace será el nombre del proyecto.

Para utilizar las clases de un proyecto A en un proyecto B, debemos hacer dos pasos:

* 1. Agregar la librería del proyecto de clases a utilizar.
  2. Utilizar el namespace en cada clase

Comenzamos creando un nuevo proyecto de tipo **Class Library** llamado **ProyectoClases**

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

Descripción generada automáticamente

A partir de ahora, hablaré sobre el Proyecto de clases.

Vamos a crear una clase llamada **Persona**

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Vamos a utilizar esta clase Persona dentro del proyecto de Formularios (Windows) y sacar nuestras conclusiones.

Para agregar una librería de un proyecto a otro proyecto, debemos hacer **Referencia** **de proyecto**

Sobre el proyecto de Windows, botón derecho y agregamos la referencia al proyecto de clases.

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Para nuestras pruebas, vamos a crear un nuevo formulario llamado **Form20TestClases**

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

Descripción generada automáticamente

Creamos una persona en un botón

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Cuando diseñamos clases, somos NOSOTROS quienes decidimos qué se puede hacer y que NO sobre nuestra clase.

Todo esto son conceptos de usabilidad.

En nuestro cerebro, una persona no debería tener una edad negativa.



Cuando trabajamos con clases, se utilizan propiedades. Una propiedad tiene GET y SET para poder trabajar con sus valores.

* GET se utiliza cuando alguien recupera el valor de la propiedad
* SET se utiliza cuando alguien establece el valor de la propiedad

Mediante estas dos palabras, tenemos un código dónde podremos validar lo que deseamos hacer o no sobre nuestras clases y propiedades.

A esa propiedad con get y set se le llamada **Propiedad extendida** y se utiliza cuando deseamos comprobar un valor de la propiedad.

Se necesita un campo para “controlar” los valores de la propiedad. Cuando el dato que nos han dado nos guste, utilizamos dicho Campo.

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Vamos a realizar el ejemplo, comprobando el valor de la edad.

Texto

Descripción generada automáticamente con confianza media

Podemos ser “suaves” o podemos lanzar una excepción. Una excepción detiene el programa porque alguien ha hecho algo grave.

Para lanzar una excepción se utiliza: **throw new Exception(“mensaje error”)**

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

**PROPIEDADES AUTOIMPLEMENTADAS**

Siempre que hablamos de clases, hablamos de propiedades.

Tenemos dos formas de crear propiedades. Implementada o Extendida.

Si no deseamos comprobar nada, utilizamos **Implementada**

Sintaxis:

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

**PROPIEDADES ENUMERADAS**

Una propiedad enumerada nos ofrece una serie de valores para decidir entre los que tengamos disponibles. El programador no puede inventarse un valor, tendrá que elegir alguno de los disponibles.

Una enumeración está compuesta por una serie de valores.

Dichos valores se declaran en una clase, pero dentro del **namespace**

public enum NombreEnumeracion { VALOR1, VALOR2, VALOR3…

}

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Posteriormente, simplemente creamos una propiedad con el tipo de Enumeración.

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

**METODOS DE CLASE**

Un método es un conjunto de acciones de una clase.

Podemos tener métodos de acción solamente (**void**) o métodos que devuelvan un valor (**return)**

Los métodos pueden recibir parámetros entre sus paréntesis.

Pongamos que necesitamos un método para devolver el nombre completo de una Persona (nombre y apellidos).

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

**POLIMORFISMO**

Este concepto tiene que ver con los métodos.

Por supuesto, podemos tener todos los métodos que deseemos en una clase.

¿Qué sucede si tenemos métodos que, más o menos, realicen la misma acción?

Ejemplo, hemos creado un método para mostrar el nombre completo de Nombre y Apellidos.

¿Y si necesito recuperar los Apellidos y el Nombre?

* GetNombreCompleto(): Nombre y Apellidos
* GetNombreCompletoInverso(): Apellidos y Nombre
* GetNombreCompletoMayusculas(): Nombre y Apellidos Mayúsculas
* GetNombreCompletoInversoMayusculas(): Apellidos y nombre mayúsculas.

Ahora mismo, estamos creando varios pedales de freno para un Coche.

Lo suyo es, un mismo nombre de método y que el usuario decida las posibilidades.

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

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

**CONSTRUCTORES DE CLASE**

Un constructor en realidad es también un método.

Es el primer método que se ejecuta en cualquier clase.

Sirve para iniciar todos los elementos que conforman una clase, para modelar el inicio de las propiedades de la clase.

Cuando un programador está llamando al constructor, lo hace mediante la palabra **new**. En ese momento, está leyendo el constructor aunque no lo hayamos escrito.

Interfaz de usuario gráfica, Aplicación, Teams

Descripción generada automáticamente

Un constructor debe llamarse como la clase.

Un constructor podría tener sobrecarga/polimorfismo.

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Llamamos a un constructor u otro…



### ACCESO A FICHEROS

El namespace para acceder a ficheros es **System.IO**

Tenemos un montón de clases: DirectoryInfo, FileInfo…

Por supuesto, tenemos clases para acceder a ficheros y su contenido.

TextWriter, TextReader: Solo texto

StreamReader, StreamWriter: Texto, bytes, PDF

¿Cuándo estamos leyendo un fichero, sabemos el tiempo que va a tardar el equipo en leer dicho fichero?

Para leer ficheros, necesitamos realizarlo de forma asíncrona, es decir, cuando leemos, sabemos cuando comenzamos, pero no cuando finaliza y debemos generar un código para poder realizar estas acciones correctamente.

Tenemos dos palabras clave para utilizar acciones asíncronas:

* **async y await:** Son dos instrucciones que debemos utilizar a la vez.

Async: Se utiliza en la definición del método.

Await: Se utiliza en la llamada a los métodos asíncronos.

void LeerFichero() {

string data = “”;

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

data = fichero.ReadFile();

this.txtContenido.Text = data;

}

**async** void LeerFichero() {

string data = “”;

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

data = **await** fichero.ReadFile();

this.txtContenido.Text = data;

}

### USING EN BLOQUE

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

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

using (OBJETO) {

//ACCESO AL OBJETO

}

Pongamos el siguiente ejemplo:

public class AccesoDatos {

public AccesoDatos(string conexión){

//AQUÍ NOS CONECTAMOS

}

public string LeerDatosBBDD(){

//LEEMOS ALGO DE LA BASE DE DATOS

}

}

Cuando vayamos a utilizar la clase de acceso a datos, deberíamos conectarnos con la base de datos y, posteriormente, leer los datos.

AccesoDatos datos = new AccesoDatos(“server=SQL Express….”);

string respuesta = datos.LeerDatosBBDD();

La línea de color ROJO, todavía no sabemos si ha terminado de conectar o NO. El objeto podría estar todavía conectando en su creación.

Con los bloques **using** nos aseguramos que el objeto que estamos instanciando/creando ha sido creado correctamente antes de ser utilizado.

using (AccesoDatos datos = new AccesoDatos(“server=SQL Express…”)) {

string respuesta = datos.LeerDatosBBDD();

}

Trabajamos sobre el proyecto de Formularios (Windows).

Sobre el proyecto, creamos un nuevo form llamado **Form21Files**

Texto

Descripción generada automáticamente

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

Descripción generada automáticamente

Interfaz de usuario gráfica, Aplicación, Word

Descripción generada automáticamente

**CONTENIDO FICHERO**

**public** **partial** **class** Form21Files : Form

{

**string** rutaFichero;

**public** Form21Files()

    {

**InitializeComponent**();

        //TENEMOS DOS FORMAS DE ESCRIBIR UNA RUTA DENTRO DEL PROGRAMA

        //LA RUTA A MI FICHERO ESTARA EN Documents. C:\Users\Serra\Documents

        //EN C# LA CONTRABARRA INDICA UN CARACTER ESPECIAL DEL LENGUAJE

        //POR EJEMPLO, UN ENTER \n O UN TABULADOR \t

        //1) UTILIZAMOS DOBLE CONTRABARRA PARA LAS RUTAS

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

        //2) UTILIZAMOS @ ANTES DEL STRING PARA QUE SEA LITERAL

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

    }

    //DEBEMOS UTILIZAR OPERACIONES ASINCRONAS

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

    {

        //VAMOS A UTILIZAR LA RUTA QUE TENEMOS A NIVEL DE CLASE

        FileInfo file **=** **new** FileInfo(**this.**rutaFichero);

        //CON LA CLASE TEXTREADER INDICAMOS QUE VAMOS A LEER UN FICHERO

        //DE TEXTO PLANO

        //AL CREAR EL OBJETO, DEBEMOS ASEGURARNOS QUE LO HA CREADO

        //ANTES DE UTILIZARLO

**using** (TextReader reader **=** file**.OpenText**())

        {

            //AQUI EL OBJETO ESTARA 100% DISPONIBLE

            //SI TENEMOS METODOS ASINCRONOS, DEBEMOS UTILIZARLOS

            //LOS METODOS ASYNC FINALIZAN CON DICHA PALABRA

            //LA LLAMADA SE REALIZA CON await

**string** contenido **=** **await** reader**.ReadToEndAsync**();

            //CERRAMOS EL FICHERO (SIEMPRE)

            reader**.Close**();

**this.**txtContenido**.**Text **=** contenido;

        }

    }

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

    {

        FileInfo file **=** **new** FileInfo(**this.**rutaFichero);

**using** (TextWriter writer **=** file**.CreateText**())

        {

            //RECUPERAMOS EL CONTENIDO ESCRITO EN LA CAJA

**string** contenido **=** **this.**txtContenido**.**Text;

            //ESCRIBIMOS EL CONTENIDO EN EL FICHERO

**await** writer**.WriteAsync**(contenido);

            //SIEMPRE QUE TRABAJEMOS CON FICHEROS, TENEMOS

            //QUE APLICAR UN METODO LLAMADO Flush PARA LIBERAR LA MEMORIA

**await** writer**.FlushAsync**();

            //CERRAMOS EL FICHERO

            writer**.Close**();

**this.**txtContenido**.**Text **=** "";

            MessageBox**.Show**("Texto guardado");

        }

    }

}

A continuación, sobre la misma Aplicación, vamos a hacer que las rutas no sean estáticas, sino que serán dinámicas utilizando un Control nuevo de los formularios.

Tenemos una clase llamada **OpenFileDialog** para abrir ficheros en Forms.

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Tabla

Descripción generada automáticamente con confianza baja

Contiene un método llamado **ShowDialog()** para mostrar el control.

Tiene una propiedad llamada **FileName** que nos devuelve la ruta al fichero seleccionado.

Al igual que tenemos un control llamado **OpenFileDialog**, tenemos otro llamado **SaveFileDialog** que es igual, pero para guardar ficheros.

Imagen que contiene Tabla

Descripción generada automáticamente

Modificamos el código que tenemos en formulario para leer y guardar ficheros.

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

{

    //ABRIMOS EL CUADRO DE DIALOGO DE FICHEROS

**this.**openFileDialog1**.ShowDialog**();

    //CAPTURAMOS LA RUTA DEL FICHERO

**string** path **=** **this.**openFileDialog1**.**FileName;

    FileInfo file **=** **new** FileInfo(path);

    //CON LA CLASE TEXTREADER INDICAMOS QUE VAMOS A LEER UN FICHERO

    //DE TEXTO PLANO

    //AL CREAR EL OBJETO, DEBEMOS ASEGURARNOS QUE LO HA CREADO

    //ANTES DE UTILIZARLO

**using** (TextReader reader **=** file**.OpenText**())

    {

        //AQUI EL OBJETO ESTARA 100% DISPONIBLE

        //SI TENEMOS METODOS ASINCRONOS, DEBEMOS UTILIZARLOS

        //LOS METODOS ASYNC FINALIZAN CON DICHA PALABRA

        //LA LLAMADA SE REALIZA CON await

**string** contenido **=** **await** reader**.ReadToEndAsync**();

        //CERRAMOS EL FICHERO (SIEMPRE)

        reader**.Close**();

**this.**txtContenido**.**Text **=** contenido;

    }

}

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

{

**this.**saveFileDialog1**.ShowDialog**();

**string** path **=** **this.**saveFileDialog1**.**FileName;

    FileInfo file **=** **new** FileInfo(path);

**using** (TextWriter writer **=** file**.CreateText**())

    {

        //RECUPERAMOS EL CONTENIDO ESCRITO EN LA CAJA

**string** contenido **=** **this.**txtContenido**.**Text;

        //ESCRIBIMOS EL CONTENIDO EN EL FICHERO

**await** writer**.WriteAsync**(contenido);

        //SIEMPRE QUE TRABAJEMOS CON FICHEROS, TENEMOS

        //QUE APLICAR UN METODO LLAMADO Flush PARA LIBERAR LA MEMORIA

**await** writer**.FlushAsync**();

        //CERRAMOS EL FICHERO

        writer**.Close**();

**this.**txtContenido**.**Text **=** "";

        MessageBox**.Show**("Texto guardado");

    }

}

### COLECCIONES NO GRAFICAS

Hemos visto hasta ahora colecciones utilizando un control llamado **ListBox.**

Por supuesto, podemos tener colecciones en memoria sin necesidad de dibujar los elementos en una lista.

Por ejemplo, para leer los datos de un Fichero o de una base de datos y, posteriormente dibujarlos.

Las conjuntos, es decir, las colecciones, nos permiten tener datos almacenados de clases que ya existen dentro de Net (string) o de clases que nosotros hayamos creamos como, por ejemplo, Persona.

Podríamos almacenar un conjunto de personas si lo necesitamos.

La colección para almacenar datos en memoria se llama **List<T>**

Todos los objetos son del mismo tipo **T**, donde T puede ser, string, int, Persona o lo que veamos.

Como toda colección, tiene métodos para añadir (Add), eliminar (Remove), borrar todo (Clear)

Vamos a realizar un simple ejemplo en el que almacenaremos nombres en una colección NO gráfica y los dibujaremos cuando el usuario lo desee mediante un botón.

Creamos un nuevo Form llamado **Form22ColeccionesNoGraficas**

Interfaz de usuario gráfica, Aplicación, Word

Descripción generada automáticamente

**CODIGO FORMULARIO**

**public** **partial** **class** Form22ColeccionesNoGraficas : Form

{

    List<**string**> coleccionNombres;

**public** Form22ColeccionesNoGraficas()

    {

**InitializeComponent**();

        //EN EL CONSTRUCTOR ES DONDE CREAMOS LOS OBJETOS DE LA CLASE

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

    }

**private** **void** **btnGuardarNombre\_Click**(**object** sender, EventArgs e)

    {

**string** nombre **=** **this.**txtNuevoNombre**.**Text;

        //ALMACENAMOS EL NOMBRE DENTRO DE LA COLECCION NO GRAFICA

**this.**coleccionNombres**.Add**(nombre);

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

**this.**txtNuevoNombre**.**Text **=** "";

    }

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

    {

**this.**lstNombres**.**Items**.Clear**();

**foreach** (**string** nombre **in** **this.**coleccionNombres)

        {

**this.**lstNombres**.**Items**.Add**(nombre);

        }

    }

}

El siguiente ejemplo que vamos a realizar será prácticamente igual.

La diferencia estará en que almacenaremos **Personas** en lugar de simples string.

Cada vez que pulsemos sobre el botón de Guardar, almacenamos una Persona.

Cuando seleccionemos sobre el Listbox, mostraremos los datos de la Persona.

Vamos a utilizar la clase **Persona** que acabamos de crear en el proyecto de Clases.

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

Descripción generada automáticamente

Creamos un nuevo formulario llamado **Form23ColeccionPersonas**

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

Descripción generada automáticamente

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

**CODIGO FORMULARIO**

**public** **partial** **class** Form23ColeccionPersonas : Form

{

    List<Persona> coleccionPersonas;

**public** Form23ColeccionPersonas()

    {

**InitializeComponent**();

**this.**coleccionPersonas **=** **new** List<Persona>();

    }

**private** **void** **btnGuardarPersona\_Click**(**object** sender, EventArgs e)

    {

        //CADA VEZ QUE PULSEMOS EL BOTON, CREAMOS UNA NUEVA PERSONA

        Persona persona **=** **new** Persona();

        persona**.**Nombre **=** **this.**txtNombre**.**Text;

        persona**.**Apellidos **=** **this.**txtApellidos**.**Text;

        persona**.**Edad **=** **int.Parse**(**this.**txtEdad**.**Text);

        //ALMACENAMOS LA PERSONA EN LA COLECCION

**this.**coleccionPersonas**.Add**(persona);

**this.**lblMensaje**.**Text **=** "Número personas: " **+** **this.**coleccionPersonas**.**Count;

    }

**private** **void** **btnMostrarPersonas\_Click**(**object** sender, EventArgs e)

    {

**this.**lstPersonas**.**Items**.Clear**();

**foreach** (Persona persona **in** **this.**coleccionPersonas)

        {

**this.**lstPersonas**.**Items**.Add**(persona**.**Nombre);

        }

    }

**private** **void** **lstPersonas\_SelectedIndexChanged**(**object** sender, EventArgs e)

    {

**if** (**this.**lstPersonas**.**SelectedIndex **!=** **-**1)

        {

            //ENTRE TODAS LAS PERSONAS, NOS INTERESA EL INDICE

            //DE LA SELECCIONADA

**int** indice **=** **this.**lstPersonas**.**SelectedIndex;

            //RECUPERAMOS LA PERSONA DE LA COLECCION

            Persona persona **=** **this.**coleccionPersonas[indice];

**this.**txtNombre**.**Text **=** persona**.**Nombre;

**this.**txtApellidos**.**Text **=** persona**.**Apellidos;

**this.**txtEdad**.**Text **=** persona**.**Edad**.ToString**();

        }

    }

}

El siguiente ejemplo, creamos un formulario llamado **Form24ColeccionCoches**

En lugar de almacenar Personas, debemos almacenar Coches.

Un Coche estará compuesto por **Marca, Modelo y Velocidad**.

No es necesario comprobar nada, simplemente con propiedades **Implementadas** nos bastará para crear la clase.

La clase Coche debemos crearla en el **ProyectoClases**

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

Sobre el proyecto de clases, creamos una nueva clase llamada **Coche**

**COCHE**

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

En el formulario, debemos hacer el **using** sobre **ProyectoClases**

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

Descripción generada automáticamente

**CODIGO FORMULARIO**

**public** **partial** **class** Form24ColeccionCoches : Form

{

    List<Coche> coleccionCoches;

**public** Form24ColeccionCoches()

    {

**InitializeComponent**();

**this.**coleccionCoches **=** **new** List<Coche>();

    }

**private** **void** **btnGuardarCoche\_Click**(**object** sender, EventArgs e)

    {

        Coche car **=** **new** Coche();

        car**.**Marca **=** **this.**txtMarca**.**Text;

        car**.**Modelo **=** **this.**txtModelo**.**Text;

        car**.**Velocidad **=** **int.Parse**(**this.**txtVelocidad**.**Text);

**this.**coleccionCoches**.Add**(car);

**this.**lblMensaje**.**Text **=** "Coches almacenados: " **+** **this.**coleccionCoches**.**Count;

**this.**txtVelocidad**.**Text **=** "";

**this.**txtMarca**.**Text **=** "";

**this.**txtModelo**.**Text **=** "";

    }

**private** **void** **btnMostrarCoches\_Click**(**object** sender, EventArgs e)

    {

**this.**lstCoches**.**Items**.Clear**();

**foreach** (Coche coche **in** **this.**coleccionCoches)

        {

**this.**lstCoches**.**Items**.Add**(coche**.**Marca **+** " " **+** coche**.**Modelo);

        }

    }

**private** **void** **lstCoches\_SelectedIndexChanged**(**object** sender, EventArgs e)

    {

**if** (**this.**lstCoches**.**SelectedIndex **!=** **-**1)

        {

**int** index **=** **this.**lstCoches**.**SelectedIndex;

            Coche car **=** **this.**coleccionCoches[index];

**this.**txtMarca**.**Text **=** car**.**Marca;

**this.**txtModelo**.**Text **=** car**.**Modelo;

**this.**txtVelocidad**.**Text **=** car**.**Velocidad**.ToString**();

        }

    }

}

## SERIALIZACION CLASES Y OBJETOS

El concepto de serialización implica almacenar un objeto en un punto y poder recuperar dicho objeto en otro punto diferente con la misma forma.

Aunque no lo veamos, este concepto está día a día en nuestro entorno de trabajo.

Nosotros, por el simple hecho de tener un móvil, ya estamos aplicando dicho concepto.

Cuando vemos la información de nuestro móvil, no importa la marca, simplemente vemos la información. Dicha información tiene un formato que se reconoce en todos los dispositivos y que son capaces de interpretar.

A eso se le llama deserialización.

Imaginemos el ejemplo que acabamos de realizar (Coches).

Queremos almacenar la información en un fichero.

Tendríamos que guardar el Coche de “alguna forma” para recuperarlo posteriormente.

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Esto representa un Coche y podríamos leer el fichero, separar con métodos de string las @ y poner las propiedades en el objeto.

Lo que sucede es que este formato no es un standard, yo he elegido @, pero vosotros podéis elegir perfectamente cualquier carácter.

Es muy difícil que podamos convertir cualquier “texto” en un objeto.

Existen formatos creados para internet para la comunicación de información y de objetos.

Dicha información puede ser escrita en un formato llamado JSON o en otro formato llamado XML.

Coche con formato JSON

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Coche con formato XML

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Vamos a visualizar este concepto utilizando serialización XML con ficheros.

Almacenaremos una clase llamada **Mascota** con **Nombre, Raza** y **Edad**

Utilizaremos un **namespace** llamado **System.Xml.Serialization**

Tendremos un objeto llamado **XmlSerializer** que es el encargado de almacenar la clase con formato XML dentro de un fichero.

**Nota:** No podemos almacenar cualquier clase, tiene que ser una clase propia nuestra, por ejemplo, no sirve almacenar la clase **string**

Comenzamos creando un nuevo formulario para almacenar una sola Mascota.

Texto

Descripción generada automáticamente

Creamos un nuevo form llamado **Form25SerializarSimple**

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

Sobre el proyecto de clases, creamos una nueva clase llamada **Mascota**

**MASCOTA**

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

En el formulario, realizamos los **namespace** para la ejecución de la aplicación.

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

**CODIGO FORMULARIO**

**public** **partial** **class** Form25SerializarSimple : Form

{

    //NECESITAMOS UN OBJETO PARA REALIZAR LA SERIALIZACION

    XmlSerializer serializer;

**public** Form25SerializarSimple()

    {

**InitializeComponent**();

        //CADA OBJETO SERIALIZER ESTA ESPECIALIZADO EN UNA CLASE DETERMINADA

        //DEBEMOS INDICAR LA CLASE QUE ALMACENAREMOS EN SU INTERIOR

        //PARA INDICAR LA CLASE QUE VAMOS A ALMACENAR SE UTILIZA LA SINTAXIS

        // typeof(CLASE)

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

    }

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

    {

        //CREAMOS UN NUEVO OBJETO MASCOTA

        Mascota mascota **=** **new** Mascota();

        mascota**.**Nombre **=** **this.**txtNombre**.**Text;

        mascota**.**Raza **=** **this.**txtRaza**.**Text;

        mascota**.**Anyos **=** **int.Parse**(**this.**txtEdad**.**Text);

        //DEBEMOS ALMACENAR LA INFORMACION EN UN FICHERO FISICO CON

        //FORMATO XML.  PARA ELLO, SE UTILIZA LA CLASE StreamWriter

        //SI NO LE DECIMOS RUTA, LO ALMACENA DONDE TENGAMOS EL PROYECTO

**using** (StreamWriter writer **=** **new** StreamWriter("mascota.xml"))

        {

            //DEBEMOS SERIALIZAR EL OBJETO MASCOTA CON UN METODO

            //LLAMADO Serialize

**this.**serializer**.Serialize**(writer, mascota);

            //COMO ESTAMOS ESCRIBIENDO EN UN FICHERO, UTILIZAMOS Flush

**await** writer**.FlushAsync**();

            writer**.Close**();

        }

**this.**txtNombre**.**Text **=** "";

**this.**txtRaza**.**Text **=** "";

**this.**txtEdad**.**Text **=** "";

    }

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

    {

        //PARA LEER EL FICHERO, UTILIZAMOS StreamReader

**using** (StreamReader reader **=** **new** StreamReader("mascota.xml"))

        {

            //DENTRO DEL FICHERO, TENEMOS UN XML QUE REPRESENTA LA MASCOTA

            //MEDIANTE EL METODO Deserialize, EL PROPIO LENGUAJE YA SABE INTERPRETAR

            //Y RECONOCER EL OBJETO

            Mascota mascota **=** (Mascota)**this.**serializer**.Deserialize**(reader);

            reader**.Close**();

**this.**txtRaza**.**Text **=** mascota**.**Raza;

**this.**txtNombre**.**Text **=** mascota**.**Nombre;

**this.**txtEdad**.**Text **=** mascota**.**Anyos**.ToString**();

        }

    }

}

Vamos a realizar la misma práctica, pero con un conjunto de objetos Mascota (List)

En realidad será todo igual, solamente que almacenaremos un **typeof(List<Mascota>)**

Creamos un nuevo form llamado **Form26SerializarColeccion**

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

**CODIGO FORMULARIO**

**public** **partial** **class** Form26SerializarColeccion : Form

{

    List<Mascota> coleccionMascotas;

    XmlSerializer serializer;

**public** Form26SerializarColeccion()

    {

**InitializeComponent**();

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

**this.**coleccionMascotas **=** **new** List<Mascota>();

    }

    //CREAMOS UN METODO PARA DIBUJAR LAS MASCOTAS EN LA LISTA

**private** **void** **DibujarMascotas**()

    {

**this.**lstMascotas**.**Items**.Clear**();

**foreach** (Mascota mascota **in** **this.**coleccionMascotas)

        {

**this.**lstMascotas**.**Items**.Add**(mascota**.**Nombre);

        }

**this.**txtNombre**.**Text **=** "";

**this.**txtRaza**.**Text **=** "";

**this.**txtEdad**.**Text **=** "";

    }

**private** **void** **btnNuevaMascota\_Click**(**object** sender, EventArgs e)

    {

        Mascota mascota **=** **new** Mascota();

        mascota**.**Nombre **=** **this.**txtNombre**.**Text;

        mascota**.**Raza **=** **this.**txtRaza**.**Text;

        mascota**.**Anyos **=** **int.Parse**(**this.**txtEdad**.**Text);

**this.**coleccionMascotas**.Add**(mascota);

**this.DibujarMascotas**();

    }

**private** **void** **lstMascotas\_SelectedIndexChanged**(**object** sender, EventArgs e)

    {

**if** (**this.**lstMascotas**.**SelectedIndex **!=** **-**1)

        {

**int** index **=** **this.**lstMascotas**.**SelectedIndex;

            Mascota mascota **=** **this.**coleccionMascotas[index];

**this.**txtNombre**.**Text **=** mascota**.**Nombre;

**this.**txtRaza**.**Text **=** mascota**.**Raza;

**this.**txtEdad**.**Text **=** mascota**.**Anyos**.ToString**();

        }

    }

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

    {

**using** (StreamWriter writer **=** **new** StreamWriter("listamascotas.xml"))

        {

**this.**serializer**.Serialize**(writer, **this.**coleccionMascotas);

**await** writer**.FlushAsync**();

            writer**.Close**();

        }

**this.**coleccionMascotas**.Clear**();

**this.**lstMascotas**.**Items**.Clear**();

    }

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

    {

**using** (StreamReader reader **=** **new** StreamReader("listamascotas.xml"))

        {

**this.**coleccionMascotas **=** (List<Mascota>)

**this.**serializer**.Deserialize**(reader);

            reader**.Close**();

**this.DibujarMascotas**();

        }

    }

}