Clase 15 – Excepciones.

Gestión de errores:

Es la técnica que permite **interceptar** con éxito errores en **tiempo de ejecución**, esperados y no esperados. En C# la gestión de errores se maneja por medio de **excepciones**. Cuando se produce un error, **se lanza la excepción**. El programa debe construirse usando **diferentes técnicas** de gestión de errores para atrapar las excepciones y administrarlas de manera conveniente.

Las excepciones detienen el flujo actual del programa. Si no se hace nada, este deja de funcionar. El programador debe habilitar su programa para que resuelva problemas sin bloquearse.

Todas las excepciones derivan de la clase **Excepción**. -> Parte del Runtime del lenguaje común CLR.

Ventajas:
* Los mensajes de error no **están representados por valores enteros o enumeraciones**. Se utilizan clases concretas.
* Cada clase de excepción puede estar en su propio archivo, sin estar vinculada con las demás clases.
* Se generan mensajes de error significativos.
* Cada clase de excepción es descriptiva y representa un error concreto de forma clara y evidente.
* Cada clase de excepción tiene también información específica.

Bloque Try-Catch:

Son la solución que ofrece la orientación a objetos a los problemas de tratamiento de errores. La idea consiste en separar físicamente las instrucciones básicas para el tratamiento de errores. El código que **podría lanzar excepciones** se coloca en un bloque **try**. Y el código de **tratamiento** de excepciones se coloca en el **catch** aparte.

TRY: Contiene una expresión que puede generar la excepción. En caso que se produzca, el Runtime detiene la ejecución normal y empieza a buscar un bloque catch que pueda capturar la excepción pendiente.

Si no se encuentra un bloque catch apropiado, el Runtime desenreda la pila de llamadas en buscad e la función.

Llega hasta el final para encontrar un bloque catch, si incluso así no lo encuentra, cierra el programa.

Si encuentra un bloque catch, se considera que la excepción ha sido capturada y se reanuda la ejecución normal del programa.

MULTIPLES CATCH:

Un bloque de código en una **instancia try** puede contener muchas instrucciones. Y cada una **producir una o mas excepciones**. Es posible que haya muchos **bloques catch**, y que cada uno capture algo distinto. La captura de una excepción se basa únicamente en su tipo. El **Runtime** captura **automáticamente** la excepción de un tipo concreto de acuerdo al **bloque catch que corresponda**.

CATCH GENERICO:

Un bloque **catch** general (**Exception**) puede capturar cualquier excepción independientemente de su clase. Se utiliza cuando se puede producir la **falta de un controlador adecuado**. Un **bloque try** no puede tener más de un **bloque catch general**. Y este, en caso que exista, debe ser el **ultimo bloque** catch en el programa.

THROW:

Se emplea la instrucción **Throw** para lanzar una **excepción definida por el usuario**. Las excepciones esperan como parámetro una cadena con un mensaje significativo. Solo es posible lanzar un objeto si el tipo de ese objeto deriva directa o indirectamente de **System.Exception**. Se puede lanzar una instrucción Throw en un bloque catch.

FINALLY:
La clausula **Finally de C#** contiene un conjunto de instrucciones para ejecutar sea cual sea el flujo de control. Se ejecutará, aunque se abandone el bucle try como resultado de ejecución normal, o como resultado de una excepción. Es útil en 2 casos: **Para evitar la repetición de instrucciones / Para liberar recursos tras el lanzamiento de una excepción.**

Clase 16 – Test Unitarios:

Ciclo de vida de los sistemas:

DISEÑO → ARMADO → TEST → APRENDER

Test unitarios:

La idea de los **Test Unitarios** es escribir **casos de prueba** para cada función no trivial o método en el módulo, de forma que cada caso sea independiente del resto. Luego, con las **Pruebas de Integración**, se podrá asegurar el correcto funcionamiento del sistema o subsistema en cuestión.

Pruebas de Integración:

Son aquellas que se realizan en el **ámbito del desarrollo del software**, una vez que se han **aprobado las pruebas unitarias**, y lo que prueban es que todos los **elementos unitarios** que componen el software, **funcionen juntos** (probándolos en grupo).

Pruebas funcionales:

Es una prueba basada en la ejecución, revisión y retroalimentación de las funcionalidades previamente diseñadas para el software. Se hacen mediante el diseño de modelos de prueba que buscan evaluar cada una de las opciones con las que cuentan el paquete informático.

Clase Assert:

* Explícita para determinar si el método de prueba se supera o no.
* Cumple su tarea a través de métodos estáticos.
* Estos métodos analizan una condición True – False.

```
[TestMethod]
public void Withdraw_ValidAmount_ChangesBalance()
{
    // arrange
    double currentBalance = 10.0;
    double withdrawal = 1.0;
    double expected = 9.0;
    var account = new CheckingAccount("JohnDoe", currentBalance);
    // act
    account.Withdraw(withdrawal);
    double actual = account.Balance;
    // assert
    Assert.AreEqual(expected, actual);
}
```

Clase 17 – TIPOS GENÉRICOS:

GENERICS:

Es el mecanismo e implementación de **clases parametrizadas** introducido en la versión **2.0 del lenguaje C#. Una clase parametrizada** es exactamente igual a una de las clases habituales, solo qué, **su definición contiene algún elemento que depende de un parámetro que debe ser especificado en el momento de la declaración de un objeto de dicha clase**.

Esto puede resultar **extremadamente útil** a la hora de programar clases genéricas, **capaces de implementar un tipado fuerte sin la necesidad de conocer los tipos paras los que serán utilizadas.**

List es una clase parametrizada:

```
List<Parametro> l = new List<Parametro>();
```

Dictonary es otro ejemplo, con dos parámetros:

```
Dictionary<int, string> m = new Dictionary<int, string>();
```

```
public class Mensaje<T>
{
    private T miAtributo;
}

// ...
Mensaje<string> tipoTexto = new Mensaje<string>();
Mensaje<MiClase> tipoMio = new Mensaje<MiClase>();
```

RESTRICCIONES:

Una buena regla consiste en aplicar el mayor numero de restricciones posibles que siga permitiendo manejas los tipos que se deben utilizar.

where T : struct	El argumento de tipo debe ser un tipo de valor.
where T : class	El argumento de tipo debe ser un tipo de referencia.
where T : unmanaged	El argumento de tipo no debe ser un tipo de referencia y no debe contener ningún miembro de tipo de referencia en ningún nivel de anidamiento.
where T : new()	El argumento de tipo debe tener un constructor sin parámetros público. Cuando se usa conjuntamente con otras restricciones, la restricción new() debe especificarse en último lugar.
where T : <nombre de clase base>	El argumento de tipo debe ser o derivarse de la clase base especificada.
where T : <nombre de interfaz>	El argumento de tipo debe ser o implementar la interfaz especificada. Pueden especificarse varias restricciones de interfaz. La interfaz de restricciones también puede ser genérica.
where T : U	El argumento de tipo proporcionado por T debe ser o derivarse del argumento proporcionado para U.

- Algunas de estas restricciones son **mutuamente excluyentes**.
- Todos los tipos de valor deben tener un **constructor sin parámetros accesible**.
- La restricción **struct** implica la restricción new() y la restricción new () no se puede combinar con la restricción struct.
- La restricción **unmanaged** implica la restricción struct.
- La restricción **unmanaged** no se puede combinar con las restricciones struct o new().

Clase 18 – INTERFACES:

INTERFACES:

Es un **contrato que establece** una clase en la cual **asegura que implementará un conjunto de métodos**. Son una manera de describir **qué debería hacer una clase sin especificar cómo**. Descripción de uno o más método que posteriormente alguna clase **puede** implementar.

GENERALIDADES:

* C# **no permite** especificar **atributos** en las interfaces.
* Todos los métodos son **públicos** (No se permite especificarlo).
* Todos los métodos son como "**Abstractos**" – Ya que no cuentan con implementación (No se permite especificarlo)
* Se puede especificar **propiedades** (Sin implementación).
* Las clases pueden implementar **varias interfaces**.
*Las interfaces pueden "**simular**" algo parecido a la herencia múltiple.

DEFINIR UNA INTERFAZ:

```
[modificadores] interface INombreInterface
{
    //Miembros de la interface
}
```

* Se utiliza la palabra reservada **Interface**.

* Por convención, los nombres comienzan con la letra **I**, seguida del identificador.

Para que la clase implemente una interface, se emplea el **operador (:)**
Para implementar una interface a una clase derivada, **primero** hay que indicar la **clase base**, luego la interface.
Para sobrescribir los miembros de las interfaces, **NO se emplea la palabra override**, ya que no son ni virtual ni abstract.

```
[modificadores] class NombreClase : IMiInterface1, IMiInterface2
{ }
```

IMPLEMENTACION EXPLICITA:

Los miembros implementados explícitamente sirven para **ocultar la implementación de miembros de interfaces** a las clases que no lo implementan. También para **evitar ambigüedades, cuando** por ejemplo **una clase implementa dos interfaces** las cuales posee un **miembro con la misma firma**.

Las clases derivadas de clases que implementan **interfaces** de **manera explícita** no pueden sobrescribir los métodos definidos **explícitamente**.

Sintácticamente **la implementación de una interfaz de manera explícita e implícita es igual**, lo único que cambia es la **firma del miembro en la clase que implementa la interfaz**.

```
public interface IMiInterfaz
{
    string MiMetodo();
}

public class PruebaInterfazImplicita : IMiInterfaz
{
    string IMiInterfaz.MiMetodo()
    {
        return "Hola";
    }
}
```

```
public static void Main()
{
    PruebaInterfazImplicita obj = new PruebaInterfazImplicita();

    // Produce un error de compilación de que la clase
    //no contiene ese método
    //obj.MiMetodo();

    // Debemos castear el objeto
    string str = ((IMiInterfaz)obj).MiMetodo();
    Console.WriteLine(str);
    Console.ReadKey();
}
```

Clase 19 – ARCHIVOS DE TEXTO – PARTE (1):

STREMWRITTER:

La clase **StreamWritter** escribe caracteres en archivos de texto. Se encuentra en System.IO

StreamWriter (string path): Inicializa una nueva instancia de la clase StreamWriter, en un path específico. Si el archivo existe, se sobrescribirá, sino se creará.

StreamWriter (string path, bool append): Ídem anterior, si append es true, se agregarán datos al archivo existente. Caso contrario, se sobrescribirá el archivo.

StreamWriter (string path, bool append, Encoding e): Ídem anterior, dónde se le puede especificar el tipo de codificación que se utilizará al escribir en el archivo.

Write (string value):	Escribe una cadena en el archivo sin provocar salto de línea.
WriteLine(string value):	Escribe una cadena en un archivo provocando salto de línea.
Close():	Cierra el objeto StreamWriter.

```
using System.IO;
// ...
// Abro el archivo ubicado en una dirección de la máquina
StreamWriter sw = new StreamWriter("C:\\prueba.txt");

// Agrego una línea de texto
sw.WriteLine("Hola mundo!!");

// Agrego otra línea de texto
sw.WriteLine("Chau mundo!!");

// Cierro el archivo
sw.Close();
```

STREAMREADER:

La clase **StreamReader** lee desde un archivo de texto. Se encuentra en System.IO

StreamReader (string path): Inicializa una nueva instancia de la clase StreamReader. Se especifica de donde se leerán los datos.

StreamReader (string path, Encoding e): Ídem anterior, se especifica el tipo de codificación que se utilizará para leer el archivo.

Close():	Cierra el objeto StreamReader.
Read():	Lee un carácter del stream y avanza carácter a carácter. Retorna un entero.
ReadLine():	Lee una línea de caracteres del stream y lo retorna como un string.
ReadToEnd():	Lee todo el stream y lo retorna como una cadena de caracteres.

```
using System.IO;
// ...
// Abro el archivo ubicado en una dirección de la máquina
StreamReader sw = new StreamReader("C:\\prueba.txt");

// Leo una línea de texto
sr.ReadLine();

// Cierro el archivo
sw.Close();
```

EXCEPCIONES:

PathTooLongException:	La ruta supera la longitud máxima definida por el sistema
NotSupportedException:	Un nombre de archivo o de directorio de la ruta de acceso contiene un formato no válido
SecurityException:	El usuario no tiene los permisos necesarios para ver la ruta de acceso

OBJETOS UTILES:

File.Exists(string path): Es true si tiene los permisos necesarios y path contiene el nombre de un archivo existente; de lo contrario, es false. También devuelve **false si path es null**, una ruta de acceso no válida o una cadena de longitud cero. Si no tiene permisos suficientes para leer el archivo especificado, no se produce ninguna excepción y el método devuelve false, independientemente de la existencia del path.

File.Copy(string, string): Copia un archivo existente en un archivo nuevo. No se permite sobrescribir un archivo.

File.Delete(string path): Elimina el archivo especificado.

DIRECTORY:

Directory.Delete(string path): Elimina el directorio especificado, siempre y cuando esté vacio.

Directory.Delete(string, boolean): Elimina el directorio especificado y, si está indicado, los subdirectorios y archivos que contiene.

Directory.Exists(string): Determina si la ruta de acceso dada hace referencia a un directorio existente en el disco.

GetFiles(String): Devuelve los nombres de archivo (con sus rutas de acceso) del directorio especificado.

SPECIAL FOLDERS:

Por medio del método de clase **GetFolderPath** de **Environment** podemos obtener la dirección de una carpeta:
Environment.GetFolderPath

A través del enumerado Environment.SpecialFolder podemos acceder a las carpetas del sistema sin conocer su ruta completa:

Environment.GetFolderPath(Environment.SpecialFolder.Desktop)	retorna el path al **escritorio**
Environment.GetFolderPath(Environment.SpecialFolder. MyDocuments)	carpeta de Mis **Documentos**
Environment.GetFolderPath(Environment.SpecialFolder. ProgramFiles)	directorio de **archivos de programa**.

Clase 19 – ARCHIVOS DE TEXTO – PARTE (2):

SERIALIZACION:

¿Qué es? → Es el proceso de convertir un objeto en memoria en una secuencia lineal de bytes.

¿Para qué sirve? → Para pasarlo a otro proceso - Para pasarlo a otra máquina - Para grabarlo en disco - Para grabarlo en una base de datos.

FORMATTERS: Controlan el formato de la serialización.

Serialización a XML: Por defecto incluye sólo las propiedades y atributos públicos.

Serialización Binaria: Por defecto incluyen todos los atributos y propiedades, ya sean públicas o privadas.

¿Y después?
Se reconstruye el objeto mediante **Deserialización** - proceso inverso.
Puede ser en el mismo proceso o no, en la misma máquina o no.

SERIALIZACION XML:

La **serialización XML** sólo serializa los atributos **públicos** y los **valores de propiedad** de un objeto en una secuencia XML.

La serialización XML no convierte los métodos, indexadores, atributos privados ni propiedades de sólo lectura (salvo colecciones de sólo lectura).

La clase central de la serialización XML es **XmlSerializer** y sus métodos más importantes son **Serialize** y **Deserialize**.

La secuencia XML que genera **XmlSerializer** cumple con la recomendación 1.0 del W3C (www.w3.org). Los tipos de datos generados cumplen las especificaciones enumeradas en el documento titulado "**XML Schema Part 2: Datatypes**".

Al crear una aplicación que utiliza la clase **XmlSerializer**, debe tener en cuenta los siguientes elementos y sus implicaciones:

La clase **XmlSerializer** crea archivos C# (.cs) y los compila en archivos .dll en el directorio especificado por la variable de entorno TEMP; la serialización se produce con esos archivos DLL.

Una clase debe tener un constructor por defecto para que **XmlSerializer** pueda serializarla.

Sólo se pueden **serializar** los **atributos** y **propiedades públicas**.

Los **métodos no se pueden serializar**.

XmlSerializer (System.Type type) Inicializa una nueva instancia de la clase XmlSerializer, serializar objetos del tipo especificado.

Serialize (System.IO.Stream stream, Object o) Serializa el objeto, y escribe en un documento Xml usando el Stream especificado.

Deserialize (System.IO.Stream stream) Deserializa el documento Xml contenido por el Stream especificado.

XmlTextWriter Provee una manera de generar archivos con contenido de datos XML.

XmlTextWriter (string filename, System.Text.Encoding encoding) Crea una instancia de XmlTextWriter.
El filename indica en que archivo se escribirá. Con encoding se indicará cual será la codificación.

XmlTextReader Provee una manera de leer archivos con contenido de datos XML

XmlTextReader (string url): Crea una instancia de XmlTextReader.
El url indica en que archivo están los datos XML.

RESUMEN:

Se debe colocar un **constructor** por **defecto** en las clases a serializar.

Sólo se **guardaran** los **atributos** o **propiedades públicas**.

Si hay relación de herencia, se deberá colocar **[XmlInclude(typeof(Clase))]** en la clase base e indicando cada clase heredada.

Espacio de nombres: **System.Xml.Serialization**

SERIALIZACION BINARIA:

```
[Serializable]
class MiClase
{
}
```

Para poder hacer una serialización binaria se debe agregar el marcador [**Serializable**]

BINARYFORMATTER:

Serializa y **Deserializa** objetos en formato binario.
Se encuentra en el espacio de nombres **System.Runtime.Serialization.Formatters.Binary**
Puede **serializar atributos públicos** y **privados**.
Una clase debe tener un **constructor por defecto** para que **BinaryFormatter** pueda serializarla.
Los métodos más importantes de la clase **BinaryFormatter** son**: Serialize** / **Deserialize**

BINARYFORMATTER (METODOS):

BinaryFormatter(): Inicializa una nueva instancia de la clase BinaryFormatter.

Serialize(System.IO.FileStream seralizationStream, Object graph): Serializa el objeto especificado y escribe en un archivo binario usando el serializationStream especificado.

Deserialize(System.IO.FileStream serializationStream): Deserializa el archivo binario contenido por el serializationStream especificado.

FILESTREAM:

Genera un objeto para leer, escribir, abrir y cerrar archivos.

Métodos:

FileStream (string path, System.IO.FileMode mode): Inicializa una instancia de FileStream, indicando ubicación y el modo en que se creará o abrirá el archivo.

Read (byte[] array, int offset, int count): Lee un bloque de bytes y escribe los datos en el buffer dado.

Seek (long offset, System.IO.SeekOrigin origin): Establece la posición del stream al valor dado.

Write (byte[] array, int offset, int count): Escribe un bloque de bytes en el stream.

Clase 21 – BASE DE DATOS SQL

La cadena de conexión (**Connection String**) es **donde se especificarán los datos (usuario, servidor, etc.)** de una conexión a una fuente de datos. Para SQL Server, a fin de ejemplo, utilizaremos cadenas similares a esta:

String connectionStr = "Data Source=[Instancia Del Servidor]; Initial Catalog=[Nombre de la Base de Datos];Integrated Security=True";

EJEMPLO DE CONEXIÓN:

```
using System.Data.SqlClient;
// ...
SqlConnection conexion;
conexion = new SqlConnection(connectionStr);
```

COMMAND:

Representa un procedimiento almacenado o una instrucción de **Transact-SQL** que se ejecuta en una base de datos de **SQL Server**. Un comando puede ser de diferentes tipos (Procedimiento Almacenado, etc.), por ahora solo utilizaremos del tipo texto.
El comando **deberá estar asociado a una conexión**, en la cual ejecutará sus acciones.

EJEMPLO DE COMANDO:

```
SqlCommand comando;

comando = new SqlCommand();
comando.CommandType = System.Data.CommandType.Text;

comando.Connection = conexion;
```

EJEMPLO INSERT – UPDATE – DELETE

```
String consulta;
consulta = "UPDATE Personas SET nombre = 'Fer' WHERE id = 1";
//consulta = "INSERT INTO Personas (nombre)
VALUES('Pedro')";
//consulta = "DELETE FROM Personas WHERE id = 1";

comando.CommandText = consulta;
conexion.Open();
comando.ExecuteNonQuery();
```

Clase 22 – THREADS

Hilos de ejecución: En un sistema, un **hilo de ejecución**, **hebra** o **subproceso**, es una secuencia de **tareas encadenadas muy pequeñas** que pueden ser ejecutadas por **el sistema operativo**. Es simplemente una tarea que puede ser ejecutada el mismo tiempo que otra tarea.

Proceso: Se conoce como proceso, al conjunto de hilos de ejecución que **comparten los mismos recursos**.

El proceso sigue en ejecución mientras al menos uno de sus hilos de ejecución siga activo. En el momento que todos estos hilos finalizan, el proceso no existe más y todos sus recursos son liberados.

HILO PRINICPAL ---------- FIN DEL CICLO DE VIDA

HILO DECUNDARIO ---------- FIN DEL CICLO DE VIDA

EJEMPLO BASICO:

```
// Agrego la biblioteca
using System.Threading;

// ...

// Creo el hilo
Thread t = new Thread(UnMetodo);
// Inicio el Hilo
t.Start();
```

HILOS PARAMETRIZADOS:

El método utilizado puede tener parámetros. Para esto, deberemos utilizar **ParameterizedThreadStart** al instanciar el nuevo hilo. El parámetro se pasará mediante el método **Start** de dicho hilo.

```
// Creo el hilo
Thread t = new Thread(new
ParameterizedThreadStart(Metodo));
// Inicio el Hilo, indicando el parámetro
t.Start(1000);
```

```
private void Metodo(object o)
{
      Console.WriteLine((int)o);
}
```

HILOS Y CONTROLES VISUALES:

Si deseamos modificar un control visual de un formulario (**TextBox**, **ComboBox**, **Label**, **etc**.) desde un hilo diferente al principal ("dueño" de estos controles) deberemos invocar a dicho hilo. Para esto le consultaremos al control si necesita ser invocado el hilo principal (**InvokeRequired**). Luego invocaremos dicho hilo (**BeginInvoke**) mediante un delegado.

```
private void Metodo(object o)
{
    if (this.label.InvokeRequired)
    {
        this.label.BeginInvoke((MethodInvoker)delegate()
         {
            this.label.Text = (int)o;
         }
        );
    }
    else
    {
        this.label.Text = (int)o;
    }
}
```

Dicha invocación puede necesitar parámetros. Para resolver este caso, utilizaremos un **array de Object**. Al realizar el **Invoke** (sincrónico, espera que un Thread finalice para ejecutar otro) o **BeginInvoke** (asincrónico) se pasará el **delegado** y dicho **array**.

```
delegate void Callback();
public void OtroMetodo(string texto) {
    if (this.textBox.InvokeRequired) {
        Callback d = new Callback(this.OtroMetodo);
        object[] objs = new object[] { texto };
        this.Invoke(d, objs);
    } else {
        this.textBox.Text = texto;
    }
}
```

Clase 23 – EVENTOS

Un **evento** es el modo que tiene una clase en particular de **proporcionar notificaciones** a sus clientes cuando ocurre algo en particular dentro del objeto. El uso más habitual para los eventos lo vemos en las **interfaces gráficas** (evento **Click** de un botón, evento **Load** de un Form, etc.).

Los eventos proporcionan un **medio apropiado para que los objetos puedan señalizar cambios de estado** que pueden resultar útiles para los clientes de ese objeto.

Un evento **es un mensaje enviado por un objeto para indicar que se ha producido una acción invocada programáticamente** o por un usuario. Cada evento tiene un emisor que **produce** el evento y un receptor que lo **captura**.

Utilizando eventos, los componentes de la interfaz avisan a la lógica de negocios que el usuario ha ejecutado alguna acción sobre los componentes de la misma (por ejemplo: **presionar el botón del Mouse o presionar una tecla**).

- El objeto que produce (**desencadena**) el evento se denomina ***emisor*** del evento.
- El procedimiento que **captura** el evento se denomina ***receptor*** o *manejador* del evento.
- En cualquier caso, el emisor no sabe qué objeto o método responderá a los eventos que produzca. Es necesario tener un componente que enlace el emisor del evento con el receptor del evento.

El Framework .NET utiliza un tipo de delegado para trabajar como un puntero a función entre el emisor y el receptor del evento. En la mayoría de casos, el Framework .NET crea el delegado y se ocupa de gestionar los detalles por nosotros. Sin embargo, **es posible crear delegados para los casos en que se desee que un evento utilice diferentes controladores de eventos en diferentes circunstancias**.

Los eventos se declaran mediante **delegados**. Un **delegado** es un tipo que representa referencias a métodos con una lista de parámetros determinada y un tipo de valor devuelto.

* Un objeto delegado encapsula un método de modo que se pueda llamar de forma anónima. Cuando ocurre el evento, se llama a los delegados que proporcionan los clientes para el evento.

Los delegados son como los punteros de función de C++, pero tienen seguridad de tipos. Permiten pasar los métodos como parámetros. Pueden encadenarse entre sí; por ejemplo, se puede llamar a varios métodos en un solo evento. Un Evento puede tener múltiples manejadores y viceversa.

```
public delegate void MiDelegado(object sender, EventArgs e);

public event MiDelegado ElEvento;
```

MANEJADORES

Para asociar un evento a un manejador de eventos en tiempo de ejecución, hay que 'agregarlo' al evento del emisor.
objEmisor.evento += MetodoManejador;

Para quitar un evento de un manejador de eventos en tiempo de ejecución, hay que utilizar la instrucción -=.
objEmisor.evento -= MetodoManejador;

```
button1.Click += MetodoManejador;
// ...
private void MetodoManejador(Object sender, EventArgs e)
{
    MessageBox.Show("Uso el manejador de eventos");
}
```

Clase 25 – METODOS DE EXTENSION:

Permiten "**agregar**" métodos a los tipos existentes sin crear un nuevo tipo derivado, recompilar o modificar de otra manera el tipo original.

Son una **clase especial de método estático**, pero se les llama como si fueran **métodos de instancia en el tipo extendido**.

En el caso del código de cliente, **no existe ninguna diferencia aparente entre llamar a un método de extensión y llamar a los métodos realmente definidos en un tipo**.

Se definen como métodos estáticos, pero se les llama usando la sintaxis de método de instancia.

```
namespace MetodoDeExtension
{
    public static class MiExtension
    {
        public static int Metodo(this [tipo] [nombre])
        {
            return 0;
        }
    }
}
```

IMPLEMENTACION:

El primer parámetro especifica en qué tipo funciona el método, y el parámetro está precedido del modificador **this**.

Los métodos de extensión únicamente se encuentran dentro del ámbito cuando el espacio de nombres se importa explícitamente en el código fuente con una directiva **using**.

Son válidos tanto para clases como para interfaces.