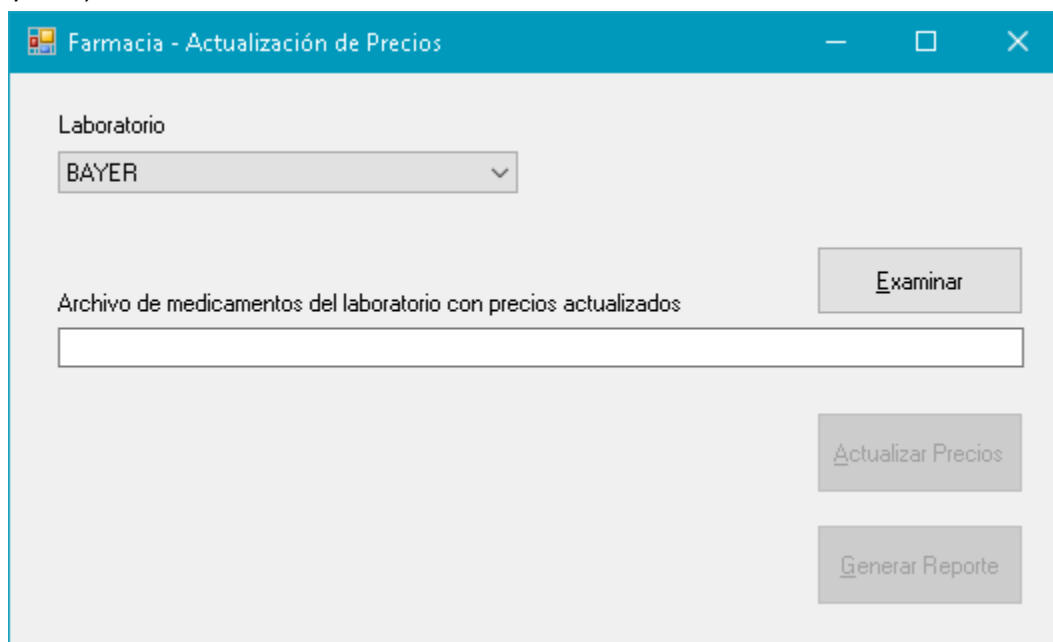


Situación profesional 1: Farmacia “La Salud”

Farmacia “La Salud”

Usted está haciendo una pasantía en la farmacia “La Salud”. El profesional farmacéutico que la gestiona, sabiendo que usted está estudiando informática en el Colegio Universitario IES, le ha solicitado el desarrollo de una aplicación para actualizar los precios de sus medicamentos a partir de archivos de texto (txt) enviados por los laboratorios que los producen.

Los archivos con las actualizaciones de los precios se tienen que elegir de cualquier medio y Ubicación, por ejemplo, de una carpeta del disco duro, de una unidad extraíble, de una lectora de CD/DVD, etc.



The screenshot shows a Windows-style application window titled "Farmacia - Actualización de Precios". Inside the window, there is a "Laboratorio" label above a dropdown menu currently showing "BAYER". Below this, there is a label "Archivo de medicamentos del laboratorio con precios actualizados" above a text input field. To the right of the input field is a button labeled "Examinar". Below the input field and the "Examinar" button, there are two more buttons: "Actualizar Precios" and "Generar Reporte".

En la interfaz, el usuario selecciona de una lista desplegable el nombre del laboratorio cuyos medicamentos desea actualizar. Seguidamente con un botón de comando deberá buscar y seleccionar de la ubicación adecuada el archivo con los datos de los medicamentos. Para comenzar el proceso de actualización deberá pulsar el botón “Actualizar”.

Cuando se concluye el proceso de actualización de precios deberá mostrarse un mensaje indicativo, si se detecta algún error también deberá informarse por pantalla.

La aplicación debe controlar que no se actualicen datos de listas con fecha anterior a la que tiene cada medicamento.

También deberá controlar si no coincide el laboratorio y el archivo elegido cuando se actualizan los precios. El archivo de texto para la actualización contiene en cada línea el código del medicamento, el nombre del medicamento y el precio del medicamento.

El botón “Generar Reporte” deberá generar un archivo “.txt” con el nombre y ubicación que el usuario elija, el reporte mostrará el detalle de la tabla Medicamentos con los precios actualizados. El formato del reporte debe tener esta estructura:

Reporte de Medicamentos Actualizados

Código	Nombre	Lab	Precio	Fecha
1010002	GENIOL X 3 TABLETAS	101	29,50	10/01/2022
1010001	ASPIRINA X 6 TABLETAS	101	29,70	25/03/2022
1020003	PASTILLAS IBUPROFENO ADULTOS	102	52,00	01/04/2022
1020002	JARABE PARA LA TOS NIÑOS	102	85,00	01/04/2022
1010005	NOVALGINA X 4 TABLETAS	101	32,00	25/03/2022
1020005	JARABE PARA LA FIEBRE NIÑOS	102	90,00	19/02/2022

Fecha del reporte: 05/03/2023

La base de datos se llama Farmacia.mdb y contiene dos tablas denominadas laboratorios y medicamentos:

- La tabla “laboratorios” contiene un número para identificar al laboratorio y el nombre del laboratorio. La clave principal es el número de laboratorio.
- La tabla “medicamentos” contiene el código de medicamento del laboratorio, el nombre del medicamento, el número de laboratorio al que pertenece el medicamento, el precio del medicamento y la fecha de la última actualización de precios. La clave principal es el código del medicamento.

SP1/H1: Componentes de ADO .NET

Para resolver nuestra situación profesional necesitaremos tener acceso a la base de datos y a sus tablas para realizar lecturas y escrituras de sus datos, para lograr esto trabajaremos con las clases de ADO .NET pensadas para resolver de manera eficiente la interacción desde el código de C# con las bases de datos. Este tema se trató con profundidad y detalle en el texto de Laboratorio de Programación 2, por lo que sugerimos una revisión del mismo para despejar cualquier duda. Seguidamente presentamos, a modo de resumen, los puntos más importantes centrados en la resolución de la SP.

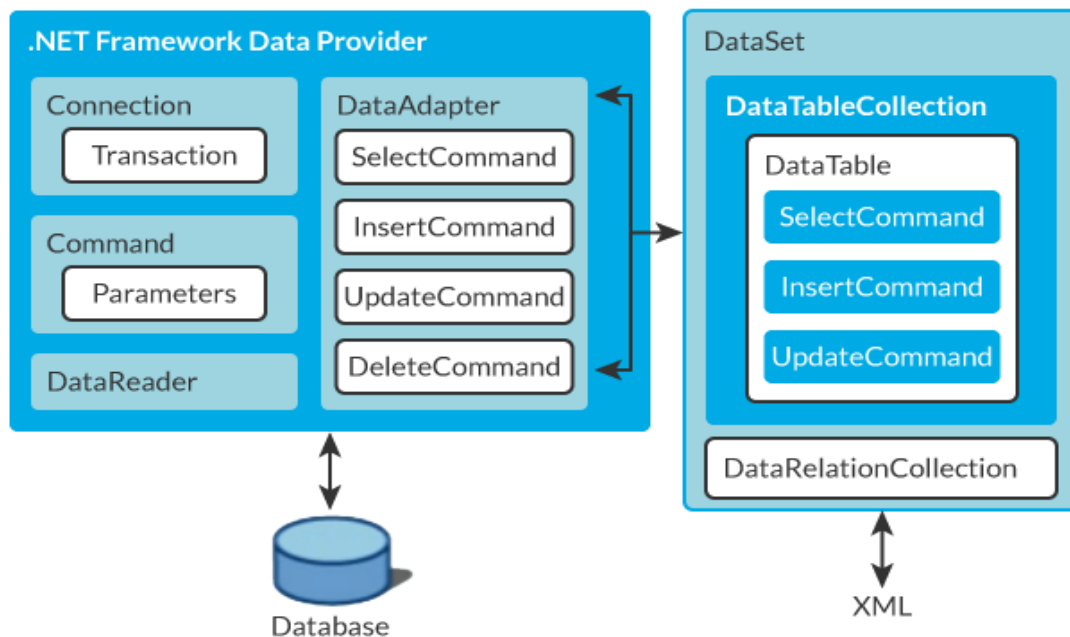
ADO .NET

ADO .NET es el modelo de objetos implementado en la plataforma .NET (ActiveX Data Objects), usado para el acceso a todo tipo de datos. ADO .NET está diseñado para trabajar con conjuntos de datos desconectados, lo que permite reducir el tráfico de datos en la red y lograr así un mejor rendimiento en las aplicaciones que lo utilizan. ADO .NET también permite trabajar en modo conectado.

La estructura de ADO .NET es la siguiente:

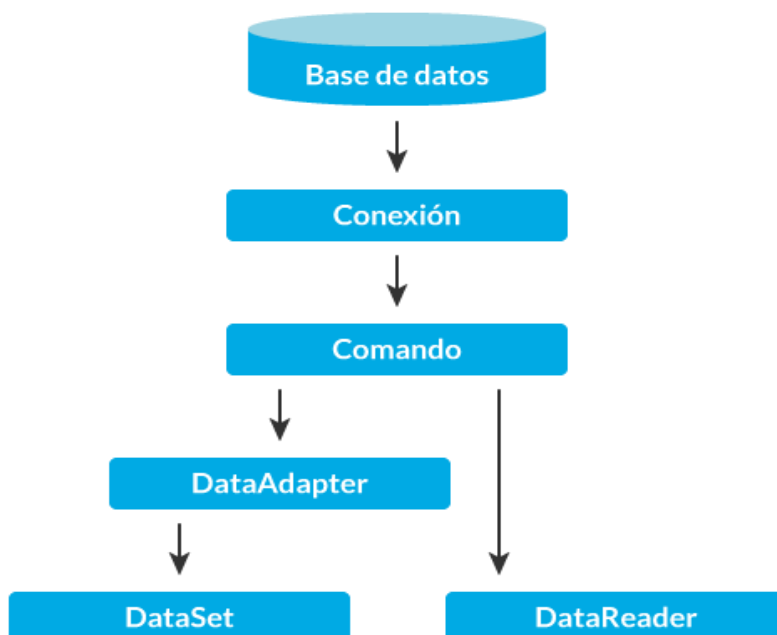
Diagrama de componentes de ADO .NET

Un conjunto de interfaces, clases, estructuras y enumeraciones que permiten el acceso a datos desde la plataforma .NET de Microsoft. En la imagen vemos sus principales herramientas y relaciones.



Resumiendo, ADO.NET nos ofrece las herramientas para acceder a una base de datos, encontraremos mucha información sobre sus características y usos, aquí exponemos algunas de las más comunes para luego dar lugar a la práctica.

Componentes principales de ADO .NET



Los proveedores de datos de .NET y el espacio de nombres System.Data proporcionan los objetos ADO.NET que utilizaremos en un escenario desconectado. ADO.NET proporciona un modelo de objetos común para proveedores de datos de .NET. La siguiente lista describe los principales objetos ADO.NET que utilizaremos en un escenario desconectado:

- *Connection*: establece y gestiona una conexión a una fuente de datos específica. Por ejemplo, la clase OleDbConnection se conecta a fuentes de datos OLE DB.
- *Command*: ejecuta un comando en una fuente de datos. Por ejemplo, la clase OleDbCommand puede ejecutar instrucciones SQL en una fuente de datos OLE DB.
- *DataSet*: diseñado para acceder a datos con independencia de la fuente de datos. En consecuencia, podemos utilizarlo con varias y diferentes fuentes de datos, con datos XML, o para gestionar datos locales a la aplicación. El objeto DataSet contiene una colección de uno o más objetos DataTable formados por filas y columnas de datos, además de clave principal, clave foránea, restricciones e información de la relación sobre los datos en los objetos DataTable.
- *DataReader*: proporciona un flujo de datos eficaz, sólo-reenvío y de sólo-lectura desde una fuente de datos.
- *DataAdapter*: utiliza los objetos Connection, Command y DataReader implícitamente para poblar un objeto DataSet, y para actualizar la fuente de datos central con los cambios efectuados en el DataSet. Por ejemplo, OleDbDataAdapter puede gestionar la interacción entre un DataSet y una base de datos Access.

En la práctica repasamos lo necesario para:

1. Realizar la conexión con la base de datos
2. Leer el contenido de una tabla de la base de datos
3. Modificar y agregar registros a una tabla de la base de datos

Conexión:

1. Importar los nombres de espacio de System.Data OleDb:

```
using System.Data.OleDb;
```

2. Establecer el valor de la cadena de conexión. Podemos obtener la sentencia de las cadenas de conexión a diferentes bases de datos en esta página: [Cadena de conexión](#).
3. Por ejemplo, para una base de datos de tipo Access cuyo nombre sea “database.mdb”, la cadena de conexión tendrá este contenido:

“Provider=Microsoft.Jet.OLEDB.4.0; Data Source=database.mdb”.

4. Crear la conexión y realizar su apertura; se usa un objeto de tipo **OleDbConnection**, su propiedad **ConnectionString** y el método **Open()**. Por ejemplo:

```
OleDbConnection cnn = new OleDbConnection();  
cnn.ConnectionString = "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=database.mdb";  
cnn.Open();
```

Lectura de una tabla

1. Ejemplo de lectura de la tabla ‘Cantantes’:



IdCantante	Nombre
1	CantanteA
2	CantanteB
3	CantanteC
*	0

Datos registrados en la tabla Cantantes

Lectura de la tabla “Cantantes” usando un DataSet:

```
OleDbConnection objcnn = new OleDbConnection();  
OleDbCommand objcmd = new OleDbCommand();  
OleDbDataAdapter objda;  
DataSet objds = new DataSet(); // objeto DataSet as usar  
// definir la cadena de conexión  
string strcnn = "Provider=Microsoft.Jet.OLEDB.4.0; Data Source=Cantantes.mdb";  
// asignar la cadena al objeto conexión  
objcnn.ConnectionString = strcnn;  
// abrir la conexión con la base de datos  
objcnn.Open();  
// establecer las propiedades al objeto comando  
objcmd.Connection = objcnn;  
objcmd.CommandType = CommandType.TableDirect;  
objcmd.CommandText = "Cantantes"; // nombre de la tabla a leer  
// crear el objeto DataAdapter pasando como parámetro el objeto comando que  
// queremos vincular  
objda = new OleDbDataAdapter(objcmd);  
// ejecutar la lectura de la tabla y almacenar su contenido en el dataAdapter  
objda.Fill(objds, "Cantantes");  
// controlar si hay registros disponibles  
if(objds.Tables["Cantantes"].Rows.Count > 0)  
{  
    string datos = ""; // variable auxiliar para almacenar los datos de la tabla  
    // recorrer los registros  
    foreach (DataRow dr in objds.Tables["Cantantes"].Rows)  
    {  
        // concatenar los campos de la tabla 'Cantantes': 'IdCantante' y 'Nombre'
```

```
        datos += dr["IdCantante"].ToString() + ", " + dr["Nombre"] + "\r\n";
    }
    MessageBox.Show(datos, "Tabla de Cantantes - DataSet");
}
// cerrar la conexión
objcnn.Close();
```

Algunos comentarios sobre el código anterior:

1. El método “**Fill**” se encarga de ‘rellenar’ el **DataSet** con todos los registros de la tabla definida en el objeto comando y al mismo tiempo se le asigna un nombre dentro del DataSet, en este caso el nombre coincide con el que tiene la tabla en la base de datos, esto facilita el control del contenido del DataSet. El proceso puede repetirse usando otros objetos comando y obtener en el DataSet varias tablas, cada una con su propio nombre y contenido. Para cada tabla se necesita un objeto comando y su correspondiente objeto **dataAdapter**.
2. Las tablas en el DataSet forman una colección cuyos elementos (las tablas) pueden identificarse con su nombre o con un índice que indique su posición dentro de la colección:
objds.Tables["Cantantes"] sería equivalente a: objds.Tables[0] si esa fuera la primera tabla agregada al DataSet.
3. A su vez la tabla posee una colección de objetos **DataRow** que puede ser recorrida con un ciclo foreach() y para acceder a los campos de la tabla se usan los nombres de los campos, así dr["IdCantante"] devuelve el valor del campo “IdCantante” en el registro actual.

Agregar nuevos registros a una tabla

Para agregar registros nuevos a la tabla debemos primero crear un objeto de tipo DataRow con la misma estructura de la tabla y luego asignar los valores a cada uno de sus campos, una vez que el registro está completo lo agregaremos a la tabla que tenemos en el DataSet. Finalmente deberemos sincronizar el cambio en el DataSet con la base de datos real.

Necesitamos entonces crear un nuevo **DataRow** y también un objeto de tipo **OleDbCommandBuilder** para que el DataAdapter sepa cómo debe actualizar la base de datos con los cambios realizados en el DataSet.

Retomamos nuestro ejemplo con la tabla de Cantantes para agregar un nuevo registro.

```
// obtenemos una referencia a la tabla de Cantantes
DataTable dt = objds.Tables["Cantantes"];
// creamos el nuevo DataRow con la estructura de campos de la tabla Cantantes
DataRow dr = dt.NewRow();
// asignamos los valores a todos los campos del DataRow
dr["IdCantante"] = 4;
dr["Nombre"] = "CantanteD";
// agregamos el DataRow a la tabla de Cantantes
dt.Rows.Add(dr);
// creamos el objeto OleDbCommandBuilder pasando como parámetro el DataAdapter
OleDbCommandBuilder cb = new OleDbCommandBuilder(objda);
// actualizamos la base con los cambios realizados
objda.Update(objds, "Cantantes");
```

El método **NewRow()** del objeto **DataTable** crea un nuevo **DataRow** con los campos de la tabla pero con su contenido vacío, es decir sin valores iniciales.

El objeto **OleDbCommandBuilder** es necesario para que el **DataAdapter** tenga la información sobre cómo ejecutar las actualizaciones sobre la base de datos.

Modificar valores en registros ya existentes

Para modificar los valores en algunos campos de un registro determinado debemos obtener una referencia al registro con un objeto **DataRow**, luego iniciar el modo de edición del **DataRow**, hacer los cambios de los valores en sus campos y finalizar el modo edición. Finalmente se actualizarán los cambios sobre la base de datos como en el caso de agregar un nuevo registro.

El modo edición se controla con los métodos “**BeginEdit()**” y “**EndEdit()**” del objeto **DataRow**, mientras esté iniciado el modo de edición se podrán hacer cambios en los valores del **DataRow**, de otra forma los cambios serán descartados.

Por ejemplo, podemos modificar el nombre del cantante con identificador igual a 1:

```
// obtenemos una referencia a la tabla de Cantantes
DataTable dt = objds.Tables["Cantantes"];

// recorrer los registros de la tabla
foreach (DataRow dr in dt.Rows)
{
    // buscar el cantante con ID = 1
    if((int)dr["IdCantante"] == 1)
    {
        // establecer el modo de edición del DataRow
        dr.BeginEdit();
        // asignamos el nuevo valor al nombre del cantante
        dr["Nombre"] = "NuevoCantanteA";
        // finalizamos el modo edición sobre del DataRow
        dr.EndEdit();
    }
}
```



```

        break; // salir del foreach
    }
}
// creamos el objeto OleDbCommandBuilder pasando como parámetro el DataAdapter
OleDbCommandBuilder cb = new OleDbCommandBuilder(objda);
// actualizamos la base con los cambios realizados
objda.Update(objds, "Cantantes");

```

Uso de la Clave Primaria (*Primary Key*) y el método ‘Find’

La clave primaria de una tabla es el conjunto de campos cuyos valores no pueden repetirse y que por lo tanto permiten identificar a cualquier registro como único en la tabla. Este concepto es propio de las bases de datos relacionales y nos permite gestionar el contenido de una tabla de forma más eficiente.

Podemos agregar la clave primaria a una tabla que ya tenemos cargada en el DataSet de esta forma:

```

// crear el arreglo para las columnas de la clave primaria
DataColumn[] objdc; // arreglo de tipo DataColumn para la clave primaria
objdc = new DataColumn[1]; // la cantidad de elementos del arreglo depende de los
                           // campos que definen la clave primaria en la tabla
// asignar las columnas de la tabla que forman la clave primaria
objdc[0] = objds.Tables[0].Columns["IdCantante"];
// establecer la propiedad PrimaryKey con las columnas correspondiente
objds.Tables[0].PrimaryKey = objdc;

```

Necesitamos un arreglo de elementos de tipo **DataColumn** cuya cantidad de posiciones sea igual a la cantidad de columnas que forman la clave primaria de la tabla, en la mayoría de los casos será una sola posición, pero hay tablas cuya clave primaria está compuesta por 2 o más campos también. Cada elemento del arreglo recibirá el valor de la columna correspondiente y finalmente el arreglo completo se asigna a la propiedad ‘**PrimaryKey**’ de la tabla que tenemos en el DataSet.

El uso de la clave primaria está muy relacionado con las búsquedas ya que en lugar de recorrer los registros uno por uno hasta encontrar el que buscamos podemos aplicar el método ‘**Find**’ pasando como parámetro el valor del campo clave y obtener como resultado el registro buscado, todo en un solo paso, por ejemplo, si repetimos el proceso para modificar un registro:

```

// obtenemos una referencia a la tabla de Cantantes
DataTable dt = objds.Tables["Cantantes"];
// creamos el arreglo para los valores que forman la clave a buscar
object[] valor = new object[1];
valor[0] = 4; // asignamos el valor de IdCantante que queremos buscar
// ejecutar la búsqueda sobre la tabla en el DataSet
DataRow dr = objds.Tables[0].Rows.Find(valor);

```

```
// controlar el resultado de la búsqueda
if(dr is null == false) // si no es nulo
{
    // establecer el modo de edición del DataRow
    dr.BeginEdit();
    // asignamos el nuevo valor al nombre del cantante
    dr["Nombre"] = "NuevoCantanteA";
    // finalizamos el modo edición sobre del DataRow
    dr.EndEdit();
    break; // salir del foreach
}
// creamos el objeto OleDbCommandBuilder pasando como parámetro el DataAdapter
OleDbCommandBuilder cb = new OleDbCommandBuilder(objda);
// actualizamos la base con los cambios realizados
objda.Update(objds, "Cantantes");
```

Si comparamos con el ejemplo anterior de modificar, vemos que ya no necesitamos implementar un recorrido de los registros de la tabla hasta encontrar el valor a borrar, ahora la búsqueda es directa.

Otro punto a tener muy en cuenta es el de controlar siempre el resultado de la búsqueda ya que el método **Find** devolverá nulo (**null**) cuando el valor buscado no exista en la tabla.

SP1/Autoevaluación 1

1. Indique la opción correcta:

El objeto OleDbConnection usa la propiedadConnectionString para especificar el tipo y nombre de la base de datos.

- Verdadero X
- Falso

2. Indique la opción correcta:

El objeto OleDbCommand es el que establece la conexión con la base de datos.

- Verdadero
- Falso X

3. Marque la/s opción/es correcta/s.:

Un DataSet representa un conjunto completo de datos y se puede usar de distintas formas:

- Crear por programa tablas y rellenarlas con datos. X
- Obtener las tablas de un origen de datos ya existente. X
- Crear el DataSet a partir de un archivo de texto (.txt)

4. Indique la opción correcta:

El método "Fill" se usa para rellenar el DataSet con los datos de una tabla.

- Verdadero X
- Falso

5. Marque la/s opción/es correcta/s:

Para acceder a una tabla de la colección de tablas en un DataSet se usa esta sintaxis:

- dataset.Tables[I] X
- dataset.Tables["NombreTabla"] X

SP1/H2: Las clases Streams

Como ya estudiamos en la materia Laboratorio de Programación 2, en C# un “**Stream**” representa una corriente o flujo de datos, flujos que son muy usados en el manejo de archivos ya que implementan una capa de código extra que facilita al programador el acceso a los datos del archivo. Recomendamos revisar este tema en el texto de la materia Laboratorio de Programación 2 para recordar sus características y aplicaciones. De todas formas, a continuación, exponemos los puntos más importantes que necesitamos conocer.

Para los procesos de escritura y lectura disponemos de tipos de stream especializados: “**StreamWriter**” para realizar las escrituras en un archivo y “**StreamReader**” para las lecturas de archivos.

Repasamos brevemente cómo trabajar con el StreamWriter que necesitaremos para generar los reportes de nuestra aplicación.

Debemos agregar a nuestro código el espacio de nombres para IO (Input – Output):

```
using System.IO;
```

StreamWriter

Como dijimos antes el StreamWriter nos permite grabar datos. Necesitamos crear un objeto de la clase StreamWriter usando como parámetro la ruta incluyendo el nombre del archivo que queremos escribir. También debemos tener en cuenta el modo de trabajo con el archivo, puede ser escritura simplemente y en ese caso el archivo será creado como un nuevo archivo vacío o podemos trabajar con el modo “append” para continuar grabando datos en un archivo que ya existe sin perder la información previamente grabada.

Ejemplos:

1. Abrir y crear un archivo para escribir en él:

```
StreamWriter sw = new StreamWriter(NombreArchivo, false);
```

El primer parámetro: NombreArchivo, contiene la ruta y nombre del archivo, por ejemplo, “C:\\Datos\\Reporte.txt”.

El segundo parámetro: “false” indica que el archivo será creado desde cero, sin importar que ya exista. Si no existe se crea, si ya existe se borra y se crea de nuevo.

2. Abrir un archivo que ya existe para agregar más datos (modo ‘append’):

```
StreamWriter sw = new StreamWriter(NombreArchivo, true);
```

Para que nos permita seguir grabando sobre un archivo que ya existe sin perder su contenido el segundo parámetro debe ser “true” (modo append). En este modo si el archivo no existe será creado, pero si ya existe solamente se abre sin borrar su contenido y nos permitirá seguir grabando más datos.

Después de crear el stream ya podemos grabar datos sobre el archivo, la clase StreamWriter posee varios métodos para ello:

- Write: graba un string en el archivo
- WriteLine: graba un string en el archivo y agrega un salto de línea al final.

Ejemplo: grabar el contenido de dos variables en una línea del archivo:

```
StreamWriter sw = new StreamWriter(NombreArchivo, true);  
int Codigo = 1000;  
string Materia = "Lenguaje Java";  
sw.WriteLine(Codigo.ToString() + "," + Materia);
```

Usamos el método “WriteLine” y como parámetro se concatena el valor de la variable “Numero” convertida a string con una coma y con el contenido de la variable “Nombre”. Todo se grabará en una única línea en el archivo y se termina con un salto de línea. Resultaría:

1000,Lenguaje Java

Finalmente se debe cerrar el stream, es decir cerrar la conexión con el archivo físico y luego liberar los recursos del stream, usamos el método “Close” para lo primero y “Dispose” para liberar los recursos:

```
sw.Close();  
sw.Dispose();
```

El método “Close” nos asegura que todos los datos enviados al stream son grabados en el archivo y el uso del método “Dispose” es particularmente importante para que no queden objetos sin usar en la memoria del sistema.

StreamReader

El **StreamReader** se usa para leer datos desde un archivo, el esquema de uso es muy similar al StreamWriter, creamos el objeto indicando por parámetro la ruta y nombre del archivo que deseamos leer, el archivo debe existir, de lo contrario se genera un error.

Métodos principales de **StreamReader**:

- **Read**: lee un carácter de la secuencia
- **ReadLine**: lee caracteres hasta el fin de línea
- **ReadBlock**: lee un bloque de caracteres, se especifica la cantidad a leer y el buffer destino.
- **ReadToEnd**: lee desde la posición actual hasta el fin de la secuencia.

Ejemplo de lectura con el método “ReadLine”:

```
String Linea;
StreamReader sr = new StreamReader(NombreArchivo);
while (sr.EndOfStream == false)
{
    Linea = sr.ReadLine();
    // mostrar la línea
    MessageBox.Show(Linea);
}
sr.Close();
sr.Dispose();
```

Se crea el StreamReader con el nombre del archivo, si no se incluye la ruta, el archivo se tratará de abrir en el mismo directorio de la aplicación. Seguidamente, se realiza un ciclo repetitivo con la condición de fin de stream usando la propiedad **EndOfStream** del streamReader. Mientras esta propiedad tenga el valor falso podemos continuar realizando lecturas, cuando “EndOfStream” es verdadero nos indica que la secuencia llegó al final.

Dentro del ciclo “while” realizamos la lectura de una línea completa del archivo y la almacenamos en la variable “Linea”, luego mostramos el contenido de la variable.

Al salir del ciclo “while” debemos cerrar la secuencia usando el método “Close” y finalmente liberar los recursos del stream con el método “Dispose”.

Proceso de los datos obtenidos de cada línea del archivo

Retomando el ejemplo visto para grabar datos con StreamWriter, supongamos que además de leer el archivo necesitamos tener cada valor en forma individual para realizar otro proceso, en este caso sería el código y el nombre de la materia grabados en cada línea. Una alternativa es usar el separador entre valores, el carácter coma (“,”) para descomponer la línea en varias partes y asignar cada parte a una variable. Dicha descomposición o partición se puede hacer con el método “**Split**” de la clase string, ejemplos:

```
string Linea = "1000,Lenguaje Java";  
string[] partes = Linea.Split(',');  
int codigo = int.Parse(partes[0]);  
string materia = partes[1];
```

Con el método Split con el carácter ‘,’ como parámetro, obtenemos un arreglo de tipo string con varios elementos, tantos como comas existan en el string más uno. En este ejemplo hay una sola coma, se obtiene un arreglo de dos elementos.

Cada elemento del arreglo se podrá entonces convertir al tipo de dato correcto y asignar a una variable. En nuestro ejemplo, el elemento de la posición cero se convierte a entero y se asigna a la variable “codigo”, el elemento de la posición 1 se asigna a la variable “materia” y no necesita conversión ya que es de tipo string.

También podemos obtener el mismo resultado sin usar la variable “partes”, simplemente agregamos a la llamada del método Split el índice del elemento que nos interesa acceder:

```
string Linea = "1000,Juan Castro";  
int numero = int.Parse(Linea.Split(',')[0]);  
string nombre = Linea.Split(',')[1];
```

Para evitar errores en la conversión de números con parte decimal se recomienda especificar el valor del parámetro para la configuración regional o “cultura” deseada: Español - Latinoamérica, o Inglés - EEUU, etc.

Ejemplo: suponemos un string que contiene dos valores decimales separados por coma:

```
string Linea = "123.45,234.56";  
string[] partes = Linea.Split(',');  
decimal numero1 = decimal.Parse(partes[0], CultureInfo.InvariantCulture);  
decimal numero2 = decimal.Parse(partes[1], CultureInfo.InvariantCulture);
```

El segundo parámetro del método **Parse** se denomina “**CultureInfo**” y representa la configuración regional aplicada para los formatos de fecha, idioma, y de números que varían de país o región. Como el punto se usa para separar la parte decimal de un número real en la

región de EEUU entonces debemos aplicar ese formato al método Parse, hay dos formas de hacerlo, una es con el valor “CultureInfo.InvariantCulture” que usa de modo predeterminado los formatos independientes de la cultura o configuración regional del sistema donde se ejecute la aplicación, es decir toma las configuraciones básicas del lenguaje C# sin importar la región configurada en el sistema.

Para poder trabajar con los valores de “CultureInfo” se debe agregar al código la cláusula “using”:

```
using System.Globalization;
```

Más detalles del uso de “CultureInfo”:

[CultureInfo Class \(System.Globalization\) | Microsoft Docs](#)

1. Indique la opción correcta:

La clase “StreamReader” posee propiedades y métodos para realizar lecturas de archivos.

- Verdadero X
- Falso

2. Indique la opción correcta:

Con la clase “StreamReader” solamente se pueden grabar datos en archivos que ya existen.

- Verdadero
- Falso X

3. Indique la opción correcta:

Al leer datos de un archivo la lectura se puede realizar de un único carácter, una línea completa o de un bloque de caracteres.

- Verdadero X
- Falso

4. Indique la opción correcta:

Para cerrar un archivo que ya no se necesita se usa el método “FileClose”.

- Verdadero
- Falso X

5. Indique la opción correcta:

Al realizar lecturas de un archivo se puede usar la propiedad “EndOfStream” para determinar si todos los datos han sido leídos.

- Verdadero X
- Falso

SP1/H3: Bloque Try – Catch, Excepciones

El manejo de las excepciones es un tema conocido cuyo tratamiento podemos implementar con la estructura try-catch, pero ahora veremos cómo implementar nuestras propias excepciones para agregar mejores prestaciones a nuestros desarrollos. Se trata de crear nuevas clases que hereden de la clase “**Exception**” y redefinir algunas propiedades o métodos para obtener la funcionalidad deseada.

Desde el punto de vista técnico del lenguaje: *“En .NET, una excepción es un objeto que hereda de la clase System.Exception. Una excepción se inicia desde un área del código en la que se ha producido un problema. La excepción se pasa hacia arriba en la pila hasta que la aplicación la controla o el programa finaliza.”*

Recordemos el esquema de uso de la estructura try-catch:

El formato general del bloque es este:

```
try
{
    // bloque de sentencias a ejecutar
    // que pueden provocar excepciones
}
catch (tipo1_Exception ex)
{
    // código para tratar la excepción de tipo1
}
catch (tipo2_Exception ex)
{
    // código para tratar la excepción de tipo2
}
finally
{
    // bloque de código que se ejecutará siempre
    // ocurran o no excepciones
}
```

En el bloque “**try**” se colocan todas las instrucciones que debemos ejecutar normalmente pero que de alguna forma pueden generar una excepción, por ejemplo, la lectura de un archivo o realizar una división de 2 variables, etc.

A continuación, colocamos los bloques “**catch**” que son los encargados de atrapar las excepciones cuando se presentan. Cada “catch” puede procesar un tipo específico de excepción,

por ejemplo: `DivideByZeroException`, `IndexOutOfRangeException`, `NullReferenceException`, `InvalidOperationException`, `FileNotFoundException`, etc. O también podemos dejar un único bloque “catch” con la clase base de todas las excepciones: “`Exception`” por lo que cualquier tipo de excepción que ocurra será atrapado por el bloque “catch”.

En tercer lugar, tenemos el bloque “**finally**” cuyo contenido será ejecutado siempre, sin importar si en el bloque “try” se dispararon o no excepciones. Este bloque es de uso opcional si existe al menos un bloque “catch”.

Lanzando Excepciones desde nuestro código

Estos bloques try-catch nos permiten capturar las excepciones y darles el tratamiento adecuado para cada caso, pero también es posible relanzar la excepción original o crear un nuevo tipo de excepción por medio la sentencia “**throw**” (lanzar).

Sintaxis de la sentencia throw:

```
throw e;
```

Donde ‘e’ es un objeto de la clase “`System.Exception`” o de alguna otra clase derivada de ella.

Ejemplo de uso:

```
int[] numbers = { 2, 4, 6, 8, 10, 12, 14, 16, 18, 20 };
public int GetNumber(int index)
{
    if (index < 0 || index >= numbers.Length)
    {
        throw new IndexOutOfRangeException();
    }
    return numbers[index];
}
```

El método “`GetNumber`” debe devolver un valor del arreglo “number” de acuerdo al índice que recibe por parámetro, pero si ese índice está fuera del rango permitido se dispara una excepción del tipo “`IndexOutOfRangeException`” (índice fuera de rango), se usa la sentencia ‘throw’ para lanzar la excepción que deberá ser controlada desde el bloque de código que llame al método “`GetNumber`”.

El uso de “throw” se puede ver en muchas funciones de validación de datos, disparando excepciones ante situaciones que detectan valores incorrectos o no permitidos por las reglas de

negocio de la aplicación. Las excepciones que se disparan pueden ser excepciones ya definidas en el lenguaje o excepciones nuevas, creadas por el programador para personalizar su tratamiento.

Creando excepciones propias

Para crear una excepción propia se debe definir una nueva clase que herede de la clase “Exception” y que implemente al menos 3 constructores, uno sin parámetros, otro con el parámetro para establecer el mensaje de la excepción y otro constructor que reciba el mensaje y la excepción interna (Inner Exception) que podría contener.

Ejemplo:

```
public class InvalidDepartmentException : Exception
{
    public InvalidDepartmentException() : base() { }
    public InvalidDepartmentException(string message) : base(message) { }
    public InvalidDepartmentException(string message, Exception inner) : base(message,
inner) { }
}
```

En el ejemplo se observa el uso de “**base()**”, es la forma por medio de la cual se ejecuta el código del constructor de la clase base, de la cual hereda la nueva clase, en este caso sería la clase “Exception”.

Esta nueva clase creada se podrá usar en los bloques catch o para lanzar excepciones con la sentencia throw.

Ejemplo:

```
public void SetDepartment(String name)
{
    if(name.Length == 0)
    {
        throw new InvalidDepartmentException(“El nombre no puede estar vacío”);
    }
    // continuar el proceso del nombre
}
```

En el código del ejemplo anterior se crea la nueva excepción invocando al constructor que recibe el mensaje por parámetro.

1. Indique la opción correcta:

Una excepción es un objeto que contiene información sobre el último error ocurrido.

- Verdadero X
- Falso

1. Indique la opción correcta:

La clase “Exception” es la base para crear nuevas excepciones.

- Verdadero X
- Falso

2. Indique la opción correcta:

La sentencia “throw” permite lanzar una excepción desde cualquier punto del código.

- Verdadero X
- Falso

3. Indique la opción correcta:

El bloque try-catch-finally no permite implementar un control de errores estructurado.

- Verdadero
- Falso X

4. Indique la opción correcta:

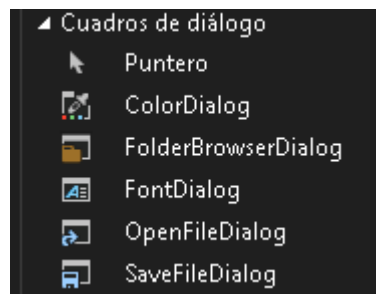
La sentencia “throw” permite lanzar excepciones sólo si estas son creadas por el programador.

- Verdadero
- Falso X

SP1/H4: Cuadro de diálogo común

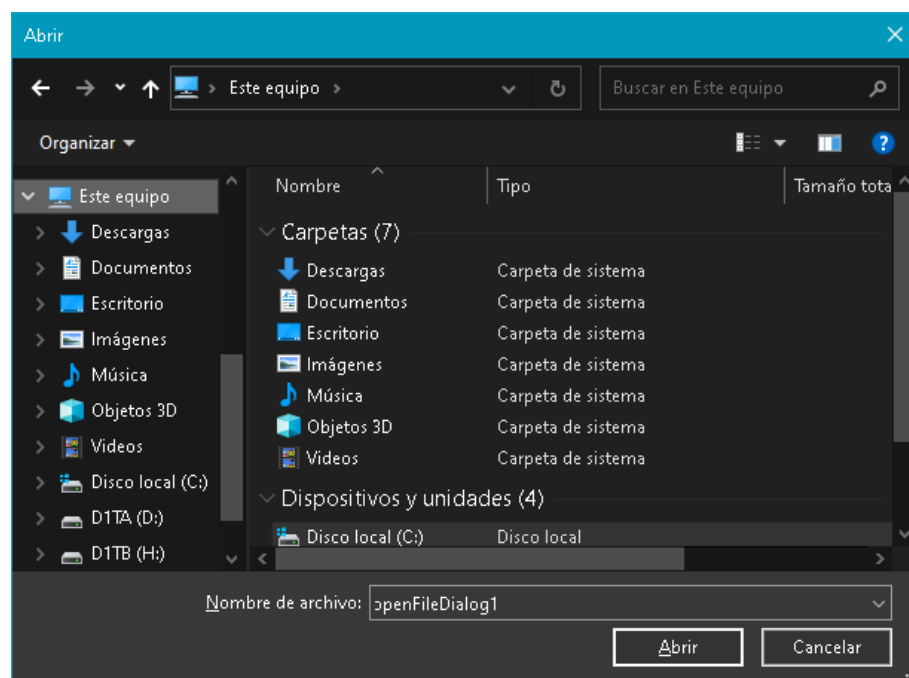
Trabajando con Cuadros de diálogo

Estos cuadros de diálogo nos van a permitir realizar actividades que son repetitivas sin necesidad de implementar su funcionalidad una y otra vez. Nos referimos, por ejemplo, a los cuadros de diálogo para la selección de un tipo de letra, para seleccionar un archivo para leer, o seleccionar una carpeta para guardar un archivo, etc. Estos componentes están disponibles en el cuadro de herramientas de *Visual Studio*:

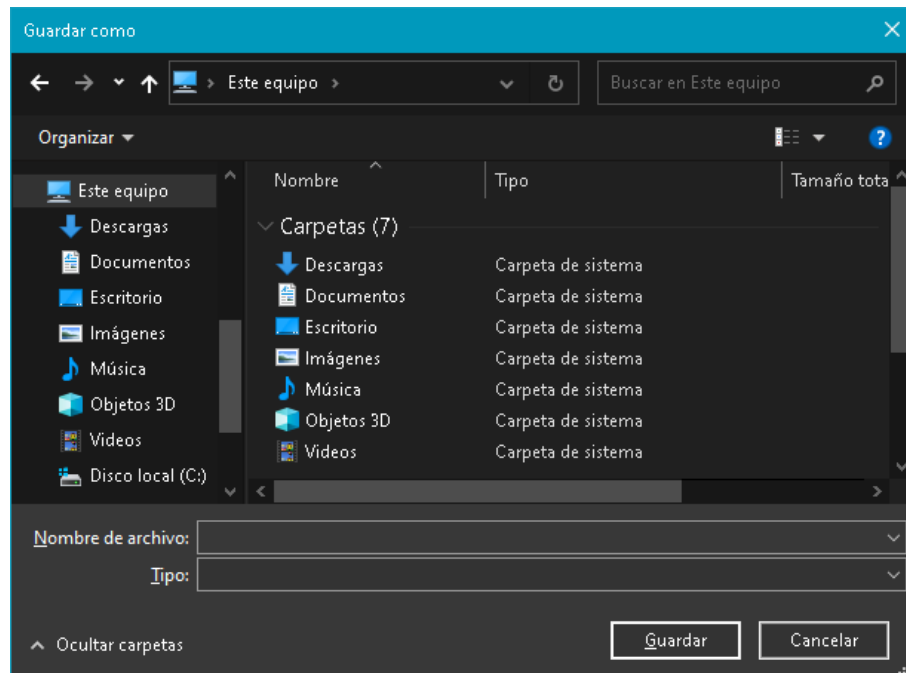


En este caso nos interesa trabajar con los diálogos para seleccionar un archivo y para definir el nombre y ubicación de un archivo a guardar.

- **OpenFileDialog**: permite seleccionar un archivo para abrirlo.



- **SaveFileDialog:** permite seleccionar una carpeta y especificar el nombre de un archivo para grabarlo.



El uso de todos estos diálogos es bastante simple, para lograr que el diálogo sea visualizado en la pantalla debemos ejecutar su método “**ShowDialog()**” y luego, cuando el usuario cierre el diálogo podremos acceder a sus propiedades para obtener los valores que fueron seleccionados. Debemos tener en cuenta también que en todos los diálogos tenemos disponible el botón “Cancelar” de manera que el usuario puede cerrar el diálogo con ese botón indicando que se deben descartar las posibles selecciones o modificaciones que hubiere realizado en el diálogo, por lo tanto, siempre debemos controlar de qué forma se cierra el diálogo. Eso se logra consultando el valor de la propiedad “**DialogResult**” que nos devuelve el método “**ShowDialog()**”.

El valor ‘**OK**’ corresponde al botón “Aceptar”, “Abrir” o “Guardar”, según el caso. Si el usuario cierra el diálogo con el botón “Cancelar” no tendremos en cuenta los valores que esperamos obtener del diálogo.

Revisaremos con un poco más de detalle el diálogo “**saveFileDialog**” más conocido como “Guardar cómo”. Como mencionamos antes cada diálogo tiene una serie de propiedades que pueden ser configuradas por el usuario en el momento de usar el diálogo, para “**saveFileDialog**” hay varias propiedades interesantes que tienen bastante uso:

- **Filter:** los filtros nos permiten especificar la extensión del archivo que vamos a usar, por ejemplo, *.txt, o *.doc, etc. El formato del texto que se asigna a la propiedad Filter

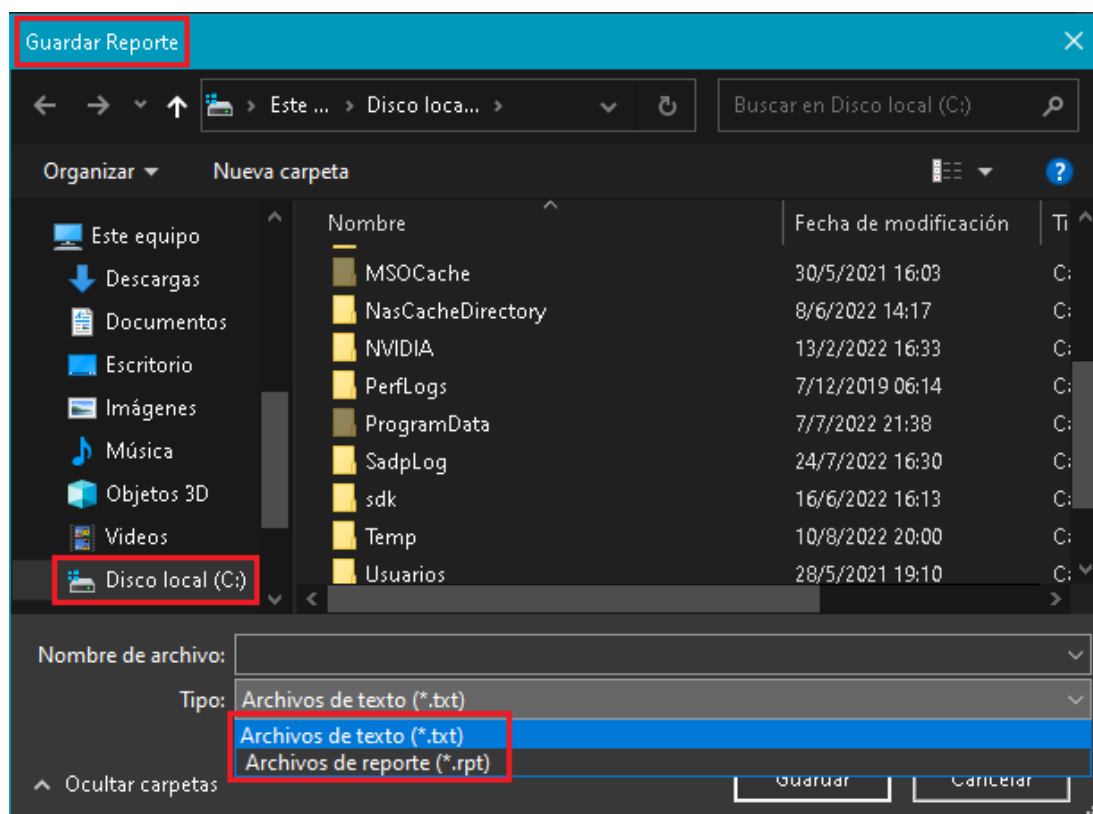
es: “descripción del filtro1 | máscara del filtro1 | descripción del filtro2 | máscara del filtro2” y así con todos los filtros que se quieran agregar, siempre se usa el carácter “|” como separador de cada parte. Ejemplo:

```
saveFileDialog.Filter = “Archivos de texto (*.txt) | *.txt | Archivos de Reporte (*.rpt) | *.rpt”;
```

- **FilterIndex:** indica qué filtro es el que está seleccionado, el primero tiene el índice cero.
- **Title:** especifica el título que mostrará el diálogo.
- **RestoreDirectory:** (booleano) si tiene el valor true el último directorio seleccionado será el que se muestre cuando el diálogo se abra nuevamente.
- **InitialDirectory:** indica cuál es el directorio inicial al mostrar el diálogo.

Todas estas propiedades se deben establecer antes de ejecutar el método “ShowDialog()” para que en el momento de visualizar el diálogo todos esos valores ya estén asignados.

En la siguiente imagen se puede ver el resultado de asignar “Guardar Reporte” a la propiedad “Title”, el valor “Archivos de texto (*.txt) | *.txt | Archivos de Reporte (*.rpt) | *.rpt” a la propiedad “Filter” y el valor “C:” a la propiedad “InitialDirectory”:



Finalmente, cuando el usuario escriba el nombre del archivo y presione el botón “Guardar” el diálogo se cerrará y podremos recuperar la ruta y el nombre del archivo desde la propiedad “**FileName**”, por ejemplo:

```
saveFileDialog1.Filter = "Archivos de texto (*.txt) |*.txt| Archivos de reporte (*.rpt)|*.rpt";
saveFileDialog1.FilterIndex = 0;
saveFileDialog1.Title = "Guardar Reporte";
saveFileDialog1.RestoreDirectory = true;
saveFileDialog1.InitialDirectory = "C:";
if(saveFileDialog1.ShowDialog() == DialogResult.OK )
{
    string ruta = saveFileDialog1.FileName;
    MessageBox.Show("Archivo de reporte: " + ruta);
}
```

Guardamos en la variable “ruta” el valor de la propiedad “FileName” con la ubicación y el nombre del archivo ingresado en el diálogo, podría ser, por ejemplo, “C:/Datos/Reporte.txt”, donde “C:/Datos/” es la ruta elegida y “Reporte.txt” es el nombre ingresado.

De forma similar podemos trabajar con el diálogo “OpenFileDialog” que posee las mismas propiedades para establecer el título, los filtros, el directorio inicial, etc.

1. Indique la opción correcta:

Con OpenFileDialog podemos establecer la ubicación y el nombre del archivo a grabar.

- Verdadero
- Falso X

2. Indique la opción correcta:

Con la propiedad "FileName" obtenemos el nombre del archivo seleccionado en un cuadro de tipo OpenFileDialog.

- Verdadero X
- Falso

3. Indique la opción correcta.

La propiedad ShowHelp=true nos permite ver el botón "Ayuda" además de los botones "Aceptar" y "Cancelar".

- Verdadero X
- Falso

4. Indique la opción correcta:

Indique la opción correcta. La propiedad Filter = "Archivos gráficos (*.jpg)|*.jpg" nos listará en la ventana todos los archivos de gráficos, sin importar su extensión.

- Verdadero
- Falso X

5. Indique la opción correcta:

Al cerrar el diálogo de tipo SaveFileDialog con el botón "Guardar" el valor devuelto es:

- DialogResult.OK X
- DialogResult.Cancel
- DialogResult.Save
- Ninguno de los anteriores

SP1/H5 Trace, EventLogs

El término “**Trace**” se refiere al rastreo o seguimiento que podemos hacer del funcionamiento de nuestra aplicación mientras se está ejecutando, “**EventLogs**” por su parte nos permite interactuar con los logs de eventos del sistema operativo Windows. Así podemos hacer un seguimiento de comportamiento de la aplicación, detectar errores y/o problemas de performance.

Para acceder a la clase “**Trace**” y sus métodos debemos agregar a nuestro código la directiva:

```
using System.Diagnostics;
```

También podemos configurar las propiedades “**autoflush**” y “**indentsize**” desde el archivo XML de configuración de la aplicación (App.config). “**autoflush**” define si los datos se escriben automáticamente en la salida definida de Trace o se espera hasta que se escriban en forma explícita, “**indentsize**” por su parte, indica la cantidad de espacios de tabulación que se agregarán a cada salida del trace. Esto se define así:

```
<configuration>
  <system.diagnostics>
    <trace autoflush="false" indentsize="3" />
  </system.diagnostics>
</configuration>
```

Y se agregan al archivo App.config:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <startup>
    <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.8" />
  </startup>
  <system.diagnostics>
    <trace autoflush="false" indentsize="3" />
  </system.diagnostics>
</configuration>
```

Para escribir información en la salida de Trace disponemos de varios métodos:

- **Write**
- **WriteLine**

- **WriteIf**
- **WriteLineIf**

Write escribe el texto pasado por parámetro.

WriteLine escribe el texto y agrega un salto de línea.

WriteIf escribe el texto sólo si se cumple la condición establecida como primer parámetro.

WriteLineIf igual que el anterior, pero agrega un salto de línea al final del texto.

Ejemplo:

```
using System;
using System.Diagnostics;

class Test
{
    static void Main()
    {
        Trace.AutoFlush = true;
        Trace.Indent();
        Trace.WriteLine("Iniciando Main");
        Trace.Unindent();
    }
}
```

Al ejecutar este ejemplo veremos en la consola de salida de Visual Studio el texto “Iniciando Main” separado del margen izquierdo con una tabulación.

Trace Listener

Un “**Listener**” representa un objeto que estará escuchando permanentemente los mensajes que envía el objeto Trace para procesarlos de acuerdo a su funcionalidad. Por defecto los mensajes escritos por Trace se dirigen a la consola de salida (DefaultTraceListener), pero es posible agregar otros Listeners:

- **FileLogTraceListener**
- **EventProviderTraceListener**
- **EventLogTraceListener**

- **TextWriterTraceListener**
- **IisTraceListener**
- **WebPageTraceListener**

Uno o más de estos Listeners se pueden agregar como salida del Trace:

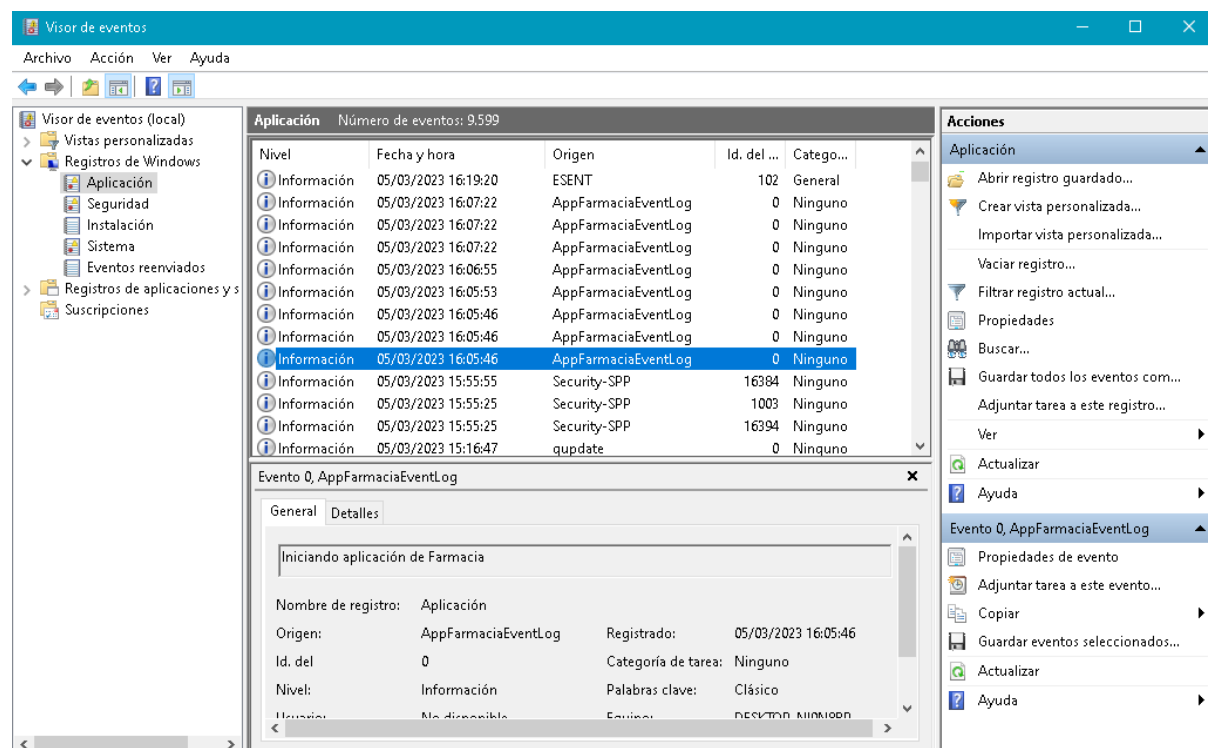
```
Trace.Listeners.Add(new EventLogTraceListener("Nombre_Aplicacion));
```

“Nombre_Aplicacion” se usará para identificar los mensajes escritos por Trace.

Trace y EventLog

Si se configura el listener “**EventLogTraceListener**” como salida de Trace entonces se podrán registrar todos los eventos que genere nuestra aplicación en el gestor de eventos de Windows para poder consultarlos posteriormente.

Los registros de todos los eventos de Windows se pueden visualizar con la aplicación “Visor de Eventos”:



Esto resulta particularmente interesante para mantener un registro de lo que ocurre durante la ejecución de una aplicación, revisar fechas y horas de cada evento y determinar posibles errores o problemas que presenta la aplicación. Por ejemplo, si un procedimiento demora mucho

tiempo en completarse podremos controlarlo fácilmente desde el visor de eventos y así buscar las posibles causas de la demora.

Permisos de uso de EventLog

El gestor de eventos de Windows requiere permisos de administrador para poder ser ejecutado sin errores, por lo que si la aplicación es ejecutada por un usuario no administrador se obtendrá un error indicando un problema de seguridad:



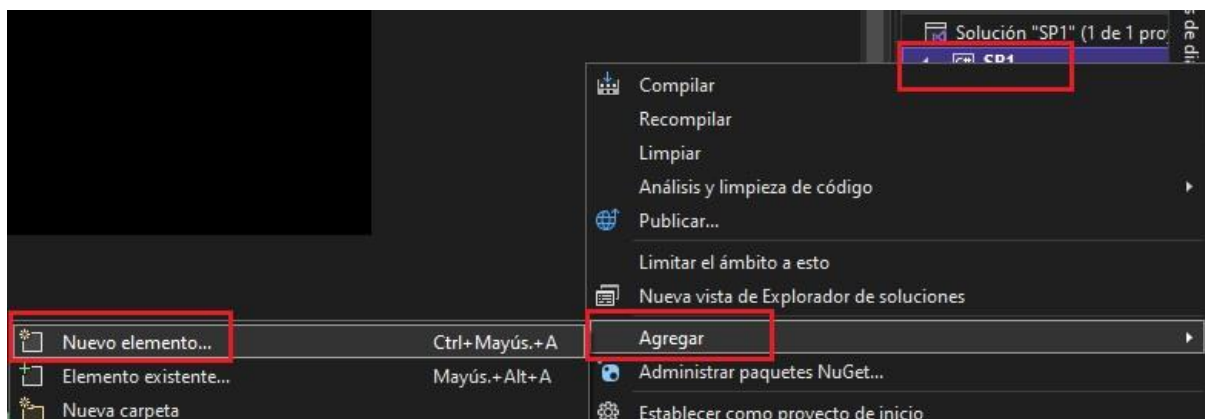
Sólo un usuario administrador puede hacer uso de EventLog

Para solucionar este problema hay que agregar al proyecto un archivo de manifiesto que permita ejecutar la aplicación a un usuario común con permisos de administrador. Los pasos a seguir son:

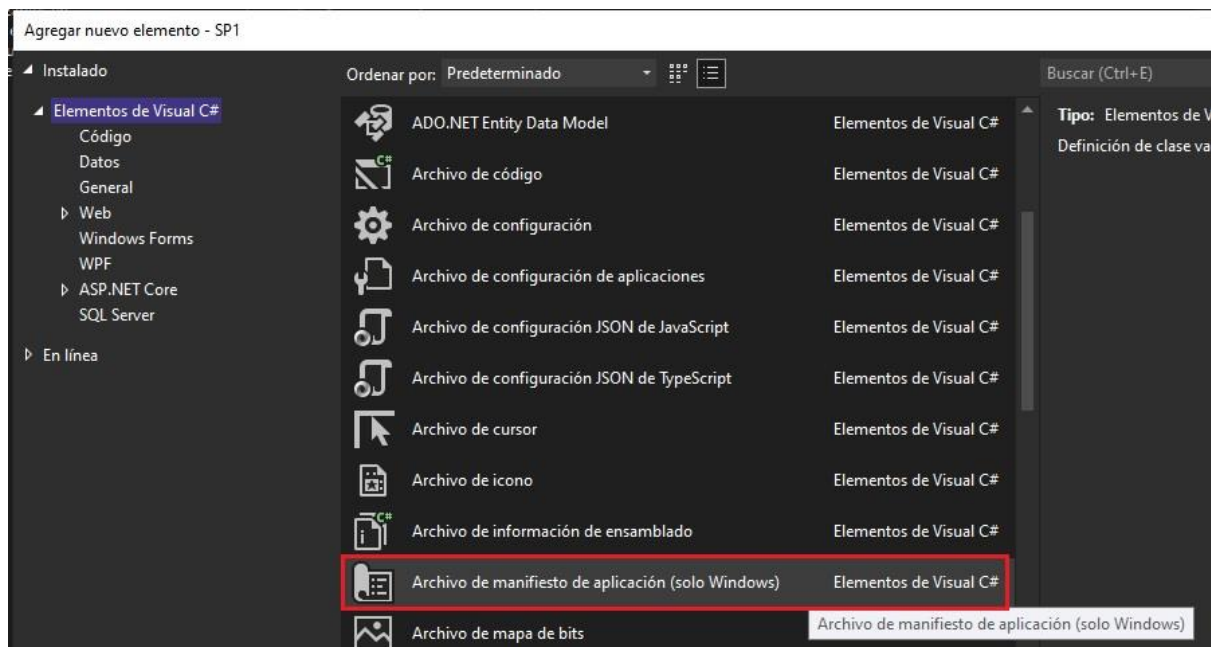
- 1- Agregar el archivo de manifiesto al proyecto.
- 2- Editar el archivo de manifiesto y modificar el permiso
- 3- Actualizar las propiedades del proyecto para que use el archivo de manifiesto creado

Agregar el archivo de Manifiesto al proyecto de Visual Studio:

Desde la ventana del Explorador de la Solución, sobre el nombre del proyecto, click derecho del mouse y seleccionamos “Agregar” – “Nuevo elemento”:

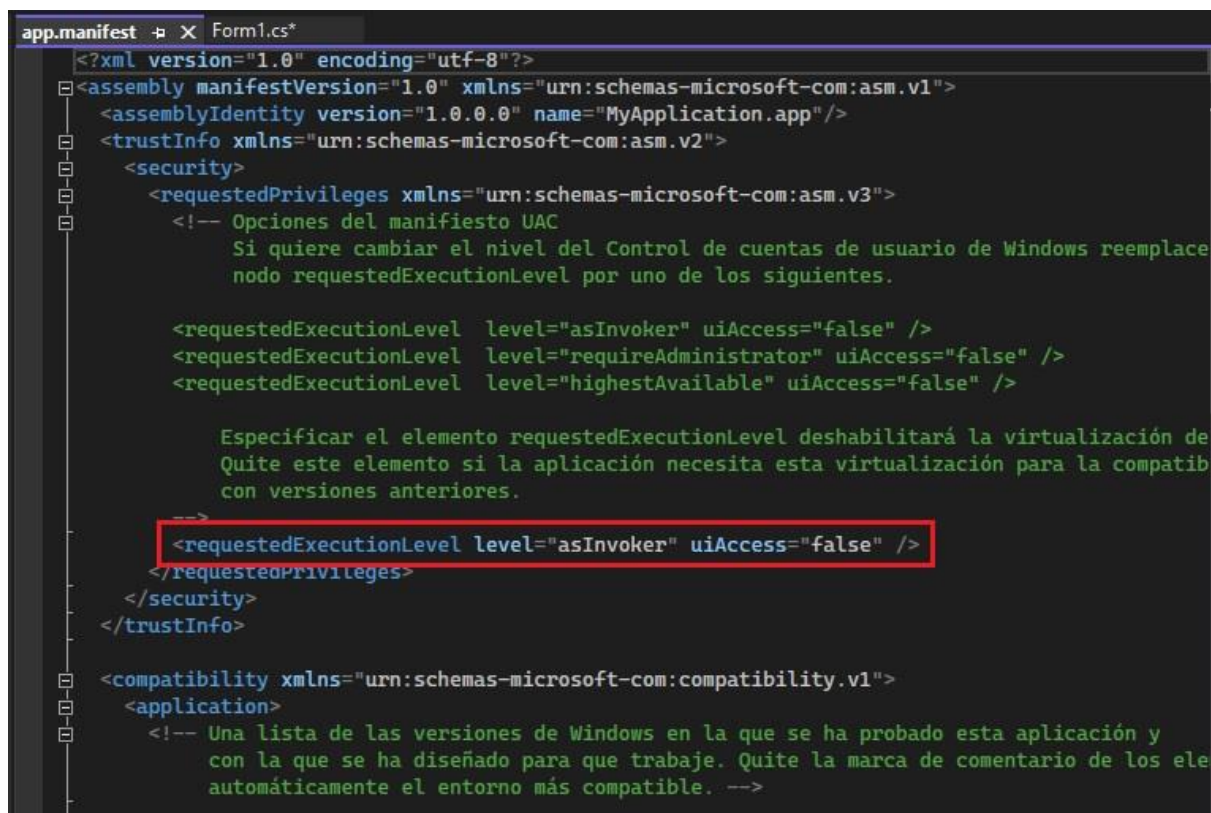


Luego, en la ventana que se abre buscamos la opción “Archivo de manifiesto de aplicación (sólo Windows)” y lo agregamos al proyecto:



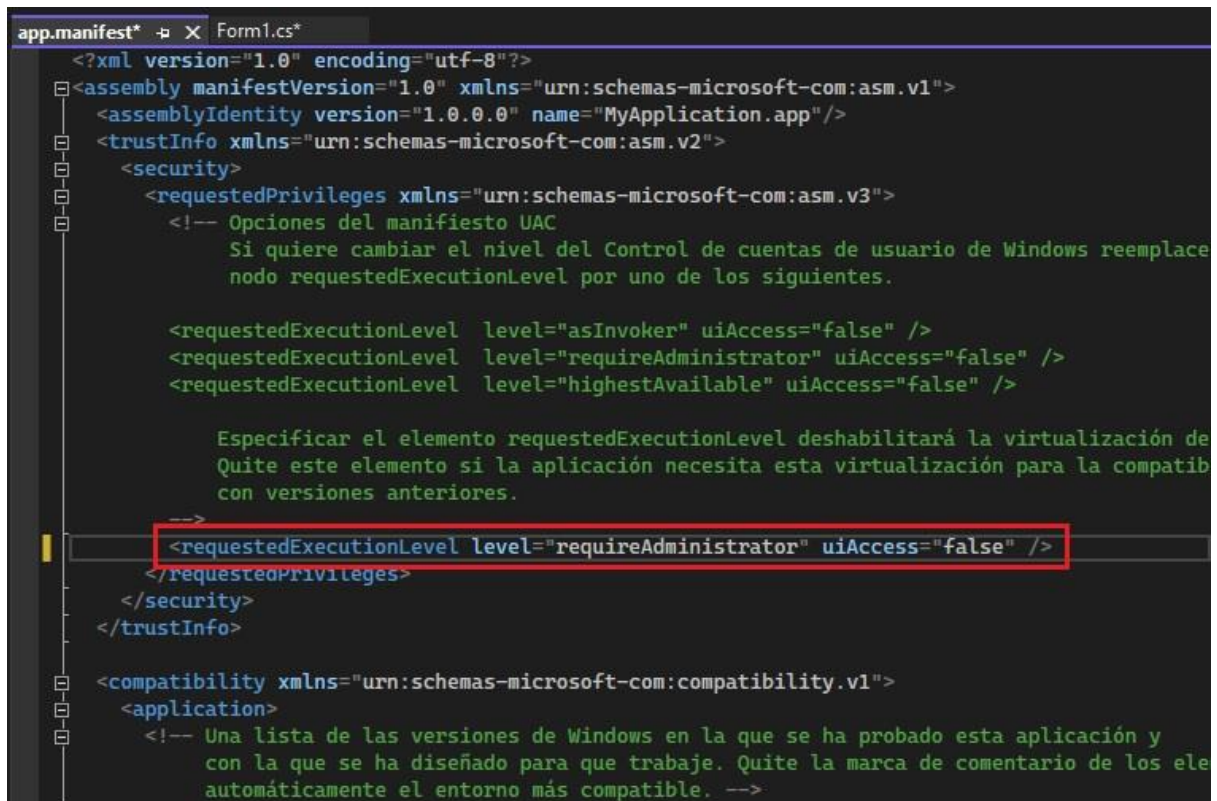
El archivo agregado lleva por nombre “app.manifest”.

Nos mostrará una ventana con su contenido (código xml):



La línea marcada es la que define el nivel de permisos con que se ejecuta la aplicación y el valor “asInvoker” es el que deberemos modificar.

Cambiamos “asInvoker” por “RequireAdmnistrator”, y resultará así:



```
<?xml version="1.0" encoding="utf-8"?>
<assembly manifestVersion="1.0" xmlns="urn:schemas-microsoft-com:asm.v1">
  <assemblyIdentity version="1.0.0.0" name="MyApplication.app"/>
  <trustInfo xmlns="urn:schemas-microsoft-com:asm.v2">
    <security>
      <requestedPrivileges xmlns="urn:schemas-microsoft-com:asm.v3">
        <!-- Opciones del manifiesto UAC
              Si quiere cambiar el nivel del Control de cuentas de usuario de Windows reemplace
              nodo requestedExecutionLevel por uno de los siguientes.

              <requestedExecutionLevel level="asInvoker" uiAccess="false" />
              <requestedExecutionLevel level="requireAdministrator" uiAccess="false" />
              <requestedExecutionLevel level="highestAvailable" uiAccess="false" />

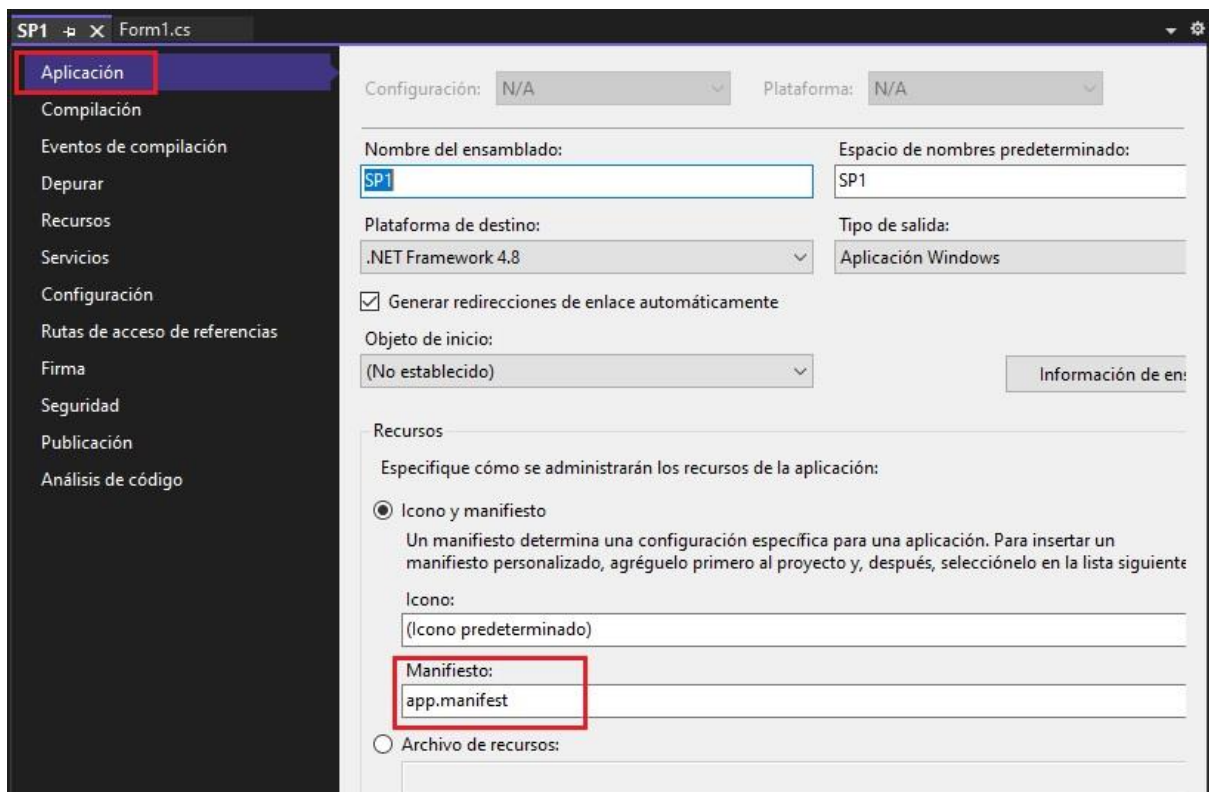
              Especificar el elemento requestedExecutionLevel deshabilitará la virtualización de
              Quite este elemento si la aplicación necesita esta virtualización para la compatib
              con versiones anteriores.
            -->
        <requestedExecutionLevel level="requireAdministrator" uiAccess="false" />
      </requestedPrivileges>
    </security>
  </trustInfo>

  <compatibility xmlns="urn:schemas-microsoft-com:compatibility.v1">
    <application>
      <!-- Una lista de las versiones de Windows en la que se ha probado esta aplicación y
            con la que se ha diseñado para que trabaje. Quite la marca de comentario de los ele
            automáticamente el entorno más compatible. -->
    </application>
  </compatibility>
</assembly>
```

Guardamos los cambios en el archivo y lo cerramos.

A continuación, debemos abrir la ventana de propiedades del proyecto, desde el menú principal de Visual Studio: opción “Proyecto”, luego “Propiedades de ...” y se mostrará esta ventana:

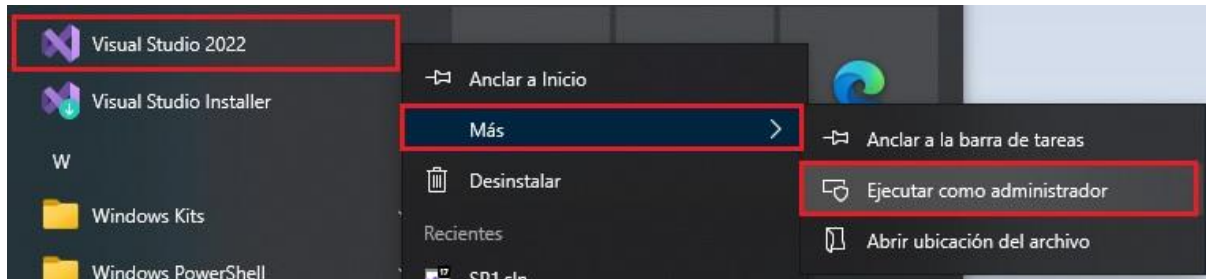
En la solapa “Aplicación” tenemos:



Debemos verificar que se muestre el archivo “app.manifest” en la casilla correspondiente al manifiesto, si no está asignado debemos agregarlo en forma manual.

Guardamos los cambios y cerramos la ventana.

Con esos cambios el usuario podrá ejecutar la aplicación sin errores. Para verificar esto desde el mismo Visual Studio, se deberá iniciar su ejecución como administrador:



Click con el botón derecho del mouse sobre “Visual Studio 2022”, luego elegimos “Más” y luego “Ejecutar como administrador”. Así Visual Studio tendrá los permisos necesarios para acceder al uso de EventLog.

Tipos de registros con Trace.

Además de escribir información simple con los métodos “Write”, la clase Trace posee métodos para definir el nivel del mensaje generado:

- **TraceInformation:** usado para información general de la aplicación
- **TraceWarning:** usado para generar mensajes de advertencia.
- **TraceError:** usado para mensajes de tipo error.

Ejemplo:

Aplicación Número de eventos: 9.621					
Nivel	Fecha y hora	Origen	Id. del ...	Catego...	
⚠ Advertencia	02/03/2023 19:04:46	Security-SPP	8233	Ninguno	
ℹ Información	02/03/2023 19:04:46	SecurityCenter	15	Ninguno	
⚠ Advertencia	02/03/2023 19:04:42	Security-SPP	8233	Ninguno	
ℹ Información	02/03/2023 19:04:38	Windows Error Reporting	1001	Ninguno	
ℹ Información	02/03/2023 19:04:37	Windows Error Reporting	1001	Ninguno	
ℹ Información	02/03/2023 19:04:37	Windows Error Reporting	1001	Ninguno	
⚠ Advertencia	02/03/2023 19:04:36	Security-SPP	8233	Ninguno	
❗ Error	02/03/2023 19:04:36	Application Error	1000	(100)	
❗ Error	02/03/2023 19:04:36	.NET Runtime	1026	Ninguno	
ℹ Información	02/03/2023 19:04:36	gupdate	0	Ninguno	
ℹ Información	02/03/2023 19:04:32	Winlogon	6000	Ninguno	
ℹ Información	02/03/2023 19:04:32	Windows Error Reporting	1001	Ninguno	
ℹ Información	02/03/2023 19:04:31	NvStreamSvc	2003	Ninguno	

En el recuadro se observan los distintos tipos de Nivel generados: Información, Advertencia y Error.

También podemos agregar un Listener para que la salida sea un archivo de texto:

```
Trace.Listeners.Add(new TextWriterTraceListener("Nombre_Archivo"));
```

Donde "Nombre_Archivo" puede incluir la ruta absoluta donde se ubicará el archivo creado, si sólo se especifica el nombre del archivo, éste se ubicará en la misma carpeta que contiene el archivo ejecutable (.exe) de la aplicación.

Para más detalles de las propiedades y métodos de la clase Trace puede consultar este enlace:

<https://learn.microsoft.com/en-us/dotnet/api/system.diagnostics.trace?view=netframework-4.8.1>

También puede revisar la información sobre la clase EventLog en este enlace:

<https://learn.microsoft.com/en-us/dotnet/api/system.diagnostics.eventlog?view=netframework-4.8.1>

1. Indique la opción correcta:

Para incluir la clase Trace en el código de la aplicación se debe agregar la cláusula “using system.Diagnostics”.

- Verdadero X
- Falso

2. Indique la opción correcta:

La clase Trace dispone de métodos para grabar y leer información, por ejemplo, Write() y Read().

- Verdadero
- Falso X

3. Indique la opción correcta:

En la clase Trace, el método WriteLine() permite escribir un texto y agrega un salto de línea al final.

- Verdadero X
- Falso

4. Indique la opción correcta:

El listener por defecto de Trace es la consola de salida.

- Verdadero X
- Falso

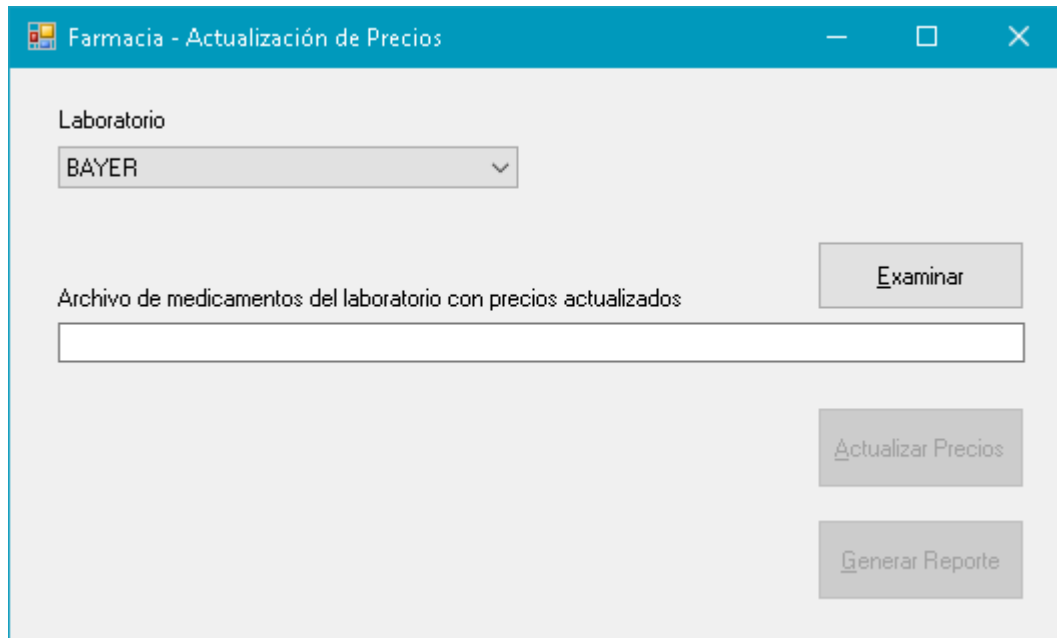
5. Indique la opción correcta:

Para visualizar los registros de Trace en el Visor de Eventos de Windows se debe usar el listener **EventProviderTraceListener**.

- Verdadero
- Falso X

SP1/Ejercicio resuelto

El diseño del formulario es realmente simple:



Tenemos un control comboBox, un textBox, Labels y botones de comando. Una vez completado el diseño del formulario pasamos a trabajar con el código de la aplicación. Trataremos de aplicar los conceptos de POO ya vistos anteriormente para crear las clases necesarias con sus correspondientes propiedades y métodos.

Como queremos tener excepciones personalizadas para el tratamiento de posibles errores en las clases que manejarán los datos de las tablas vamos a definir las clases “LaboratorioException” y “ActualizacionException”, cuyos contenidos serán muy simples, nos limitaremos a definir los 3 constructores recomendados en la documentación:

Así la clase “**LaboratorioException**” tendrá esté contenido:

```
LaboratorioException()  
LaboratorioException(string)  
LaboratorioException(string, System.Exception)
```

El código completo de la clase es este:

```
using System;  
using System.Collections.Generic;  
using System.Linq;
```

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace SP1
{
    public class LaboratorioException: Exception
    {
        //constructores
        public LaboratorioException() : base() {}
        public LaboratorioException(String message): base(message) {}
        public LaboratorioException(String message, Exception inner) : base(message,
inner) { }
    }
}

```

De forma similar, la clase “**ActualizacionException**” tendrá también los 3 constructores:

```

ActualizacionException()
ActualizacionException(string)
ActualizacionException(string, System.Exception)

```

Su código:

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace SP1
{
    public class ActualizacionException: Exception
    {
        public ActualizacionException() : base() { }
        public ActualizacionException(String message): base(message) { }
        public ActualizacionException(String message, Exception ex) : base(message,
ex) { }
    }
}

```

Clase “**CLaboratorio**”: permitirá conectarse a la base de datos y acceder a la tabla “Laboratorios”, se necesitan solamente operaciones de lectura sobre la tabla.

El contenido de la clase será:

```

CLaboratorio()
Dispose()
GetLaboratorios()
DS
Tabla

```

Tendrá el método constructor, los métodos Dispose() y GetLaboratorios(), además de dos elementos privados : el DataSet “DS” y un string “Tabla” para mantener el nombre de la tabla.

El código completo de “**CLaboratorio**” es:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Data;
using System.Data.OleDb;
using System.IO;

namespace SP1
{
    public class CLaboratorio
    {
        DataSet DS;
        String Tabla = "Laboratorios";

        public CLaboratorio()
        {
            try
            {
                OleDbConnection cnn = new OleDbConnection();
                cnn.ConnectionString = "Provider=Microsoft.Jet.OLEDB.4.0; Data
Source=Farmacia.mdb";
                cnn.Open();
                DS = new DataSet();
                OleDbCommand cmd = new OleDbCommand();
                cmd.Connection = cnn;
                cmd.CommandType = CommandType.TableDirect;
                cmd.CommandText = Tabla;
                OleDbDataAdapter DA = new OleDbDataAdapter(cmd);
                DA.Fill(DS, Tabla);
                DataColumn[] pk = new DataColumn[1];
                pk[0] = DS.Tables[Tabla].Columns["Laboratorio"];
                DS.Tables[Tabla].PrimaryKey = pk;
                OleDbCommandBuilder cb = new OleDbCommandBuilder(DA);
                cnn.Close();
            }
            catch (Exception ex)
            {
                String MsgErr = "CLaboratorio: " + ex.Message;
                throw new LaboratorioException(MsgErr);
            }
        }

        public DataTable GetLaboratorios()
        {
            if (DS != null && DS.Tables.Count == 1)
            {
                return DS.Tables["Laboratorios"];
            }
            return null;
        }

        public void Dispose()
        {
            DS.Dispose();
        }
    }
}
```

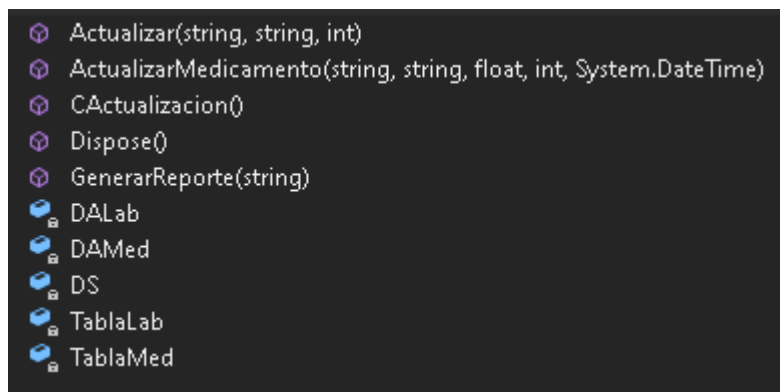
```
}  
}
```

En el **constructor** de la clase se realiza la conexión con la base de datos y se carga el contenido de la tabla “Laboratorios” en el objeto data set “DS”, también se define la propiedad de la clave primaria. En el bloque try-catch se capturan las excepciones y si algo falla se disparará una nueva excepción de tipo “LaboratorioException” con el comando “throw”. El texto que se genera en esta excepción lleva como prefijo el nombre de la clase “CLaboratorio” seguido del texto original de la excepción capturada. De esa forma será más fácil de reconocer en los logs a qué clase pertenecen las excepciones generadas.

El método “**GetLaboratorios**” devuelve el contenido completo de la tabla “Laboratorios” cargada previamente en el dataset “DS”.

El método “**Dispose**” se encarga de liberar los recursos del dataset “DS” una vez que la instancia de esta clase ya no se necesita más.

Continuamos el desarrollo del código con la clase “**CActualización**”, cuyo contenido queda definido así:



```
Actualizar(string, string, int)  
ActualizarMedicamento(string, string, float, int, System.DateTime)  
CActualizacion()  
Dispose()  
GenerarReporte(string)  
DALab  
DAMed  
DS  
TablaLab  
TablaMed
```

Esta clase será responsable, por medio de su **constructor**, de leer las tablas de Laboratorios y Medicamentos, definir las claves primarias y crear los objetos de tipo CommandBuilder para poder grabar los cambios necesarios.

El método “**Actualizar**” es el proceso principal de actualización que se encargará de leer los datos del archivo seleccionado en la interfaz, cada medicamento que forme parte del archivo se procesará con el método “**ActualizarMedicamento**”, el que se encargará de determinar si es un nuevo medicamento o uno ya existente.

El método “**GenerarReporte**” tiene por finalidad crear el archivo de texto con el reporte de todos los medicamentos existentes en la tabla de la base de datos.

Finalmente, el método “Dispose” libera los recursos empleados por los objetos DataSet y DataAdapter empleados en cada instancia de la clase.

El código completo de la clase resulta así:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Data;
using System.Data.OleDb;
using System.IO;
using System.Globalization;
using System.Diagnostics;

namespace SP1
{
    public class CActualizacion
    {
        DataSet DS;
        OleDbDataAdapter DALab;
        OleDbDataAdapter DAMed;
        String TablaLab = "Laboratorios";
        String TablaMed = "Medicamentos";

        public CActualizacion()
        {
            try{
                OleDbConnection cnn = new OleDbConnection();
                cnn.ConnectionString = "Provider=Microsoft.Jet.OLEDB.4.0; Data
Source=Farmacia.mdb";
                cnn.Open();
                DS = new DataSet();
                OleDbCommand cmd = new OleDbCommand();
                cmd.Connection = cnn;
                cmd.CommandType = CommandType.TableDirect;
                cmd.CommandText = TablaLab;
                DALab = new OleDbDataAdapter(cmd);
                DALab.Fill(DS, TablaLab);
                DataColumn[] pk = new DataColumn[1];
                pk[0] = DS.Tables[TablaLab].Columns["Laboratorio"];
                DS.Tables[TablaLab].PrimaryKey = pk;
                OleDbCommandBuilder cbLab = new OleDbCommandBuilder(DALab);
                //
                cmd = new OleDbCommand();
                cmd.Connection = cnn;
                cmd.CommandType = CommandType.TableDirect;
                cmd.CommandText = TablaMed;
                DAMed = new OleDbDataAdapter(cmd);
                DAMed.Fill(DS, TablaMed);
                DataColumn[] pkM = new DataColumn[1];
                pkM[0] = DS.Tables[TablaMed].Columns["Codigo"];
                DS.Tables[TablaMed].PrimaryKey = pkM;
            }
        }
    }
}
```

```

        OleDbCommandBuilder cbMed = new OleDbCommandBuilder(DAMed);
        cnn.Close();

    }
    catch (Exception ex)
    {
        String MsgErr = "CActualizacion: " + ex.Message ;
        throw new ActualizacionException(MsgErr);
    }
}

public bool Actualizar(String path, String NombreLaboratorio, int laboratorio)
{
    bool resultado = false;
    String linea = "";
    try
    {
        // abrir el archivo de medicamentos
        StreamReader sr = new StreamReader(path);
        String encabezado = sr.ReadLine();
        String Lab = encabezado.Split('(')[0];
        if(Lab.CompareTo(NombreLaboratorio) != 0)
        {
            throw new ActualizacionException("El archivo seleccionado no
corresponde al Laboratorio " + NombreLaboratorio);
        }
        String Fecha = encabezado.Split('(')[1].Split(' ')[0];

        while (sr.EndOfStream == false)
        {
            linea = sr.ReadLine();
            //
            String[] campos = linea.Split(',');
            if(campos.Length == 3)
            {
                String codigo = campos[0];
                String nombre = campos[1].Trim(' ');
                Single precio = Single.Parse(campos[2],
CultureInfo.InvariantCulture);
                //
                ActualizarMedicamento(codigo, nombre, precio, laboratorio,
DateTime.Parse(Fecha));
            }
        }
        sr.Close();
        sr.Dispose();
        resultado = true;
    }
    catch(ActualizacionException aex)
    {
        Trace.WriteLineIf(linea != "", "Ultima lectura: " + linea);
        throw new ActualizacionException("Error actualizando archivo de
Medicamentos: " + aex.Message);
    }
    return resultado;
}

public void ActualizarMedicamento(String codigo, String nombre, Single precio,
int laboratorio, DateTime fecha)
{
    try
    {

```

```

DataRow drM = DS.Tables[TablaMed].Rows.Find(codigo);
if (drM != null)
{
    // contolar la fecha
    if (fecha.CompareTo(drM["Fecha"]) >=0 )
    {
        // ya existe, se actualiza
        drM.BeginEdit();
        drM["Nombre"] = nombre;
        drM["Precio"] = precio;
        drM["Fecha"] = fecha;
        drM.EndEdit();
        DAMed.Update(DS, TablaMed);
        Trace.WriteLine("Medicamento actualizado " + nombre);
    }
    else
    {
        Trace.WriteLine("Medicamento con fecha anterior " + nombre);
    }
}
else
{
    // no existe, se agrega
    DataRow drNuevo = DS.Tables[TablaMed].NewRow();
    drNuevo["Codigo"] = long.Parse(codigo);
    drNuevo["Nombre"] = nombre;
    drNuevo["Precio"] = precio;
    drNuevo["Laboratorio"] = laboratorio;
    drNuevo["Fecha"] = fecha;
    DS.Tables["Medicamentos"].Rows.Add(drNuevo);
    DAMed.Update(DS, TablaMed);
    Trace.WriteLine("Medicamento agregado " + nombre);
}
}
catch(Exception ex)
{
    String MsgErr = "CActualizacion: " + ex.Message;
    throw new ActualizacionException(MsgErr);
}
}

public void GenerarReporte(String NombreArchivo)
{
    //
    try
    {
        StreamWriter sw = new StreamWriter(NombreArchivo);
        sw.WriteLine("Reporte de Medicamentos Actualizados");
        sw.WriteLine("-----");
        sw.WriteLine("    Código Nombre                                Lab
Precio
Fecha");
        foreach (DataRow dr in DS.Tables[TablaMed].Rows)
        {
            String linea = dr["Codigo"].ToString().PadLeft(10) + " ";
            linea += dr["Nombre"].ToString().PadRight(30);
            linea += dr["Laboratorio"].ToString().PadLeft(6);
            linea += String.Format("{0,12:F2}", dr["Precio"]);
            linea += string.Format("{0:d}", dr["Fecha"]).PadLeft(15);
            sw.WriteLine(linea);
        }
        sw.WriteLine("");
    }
}

```

```

        sw.WriteLine("Fecha del reporte: " +
DateTime.Now.ToString("dd/MM/yyyy"));
        sw.Close();
        sw.Dispose();
        Trace.WriteLine("Reporte generado");
    }
    catch(Exception ex)
    {
        String MsgErr = "CActualizacion: " + ex.Message;
        throw new ActualizacionException(MsgErr);
    }
}

public void Dispose()
{
    DALab.Dispose();
    DAMed.Dispose();
    DS.Dispose();
}
}
}

```

Algunas observaciones para tener en cuenta:

En el constructor, el manejo de las posibles excepciones se realiza lanzando una excepción de tipo “ActualizacionException”. Algo similar ocurre en los restantes métodos de la clase.

En los métodos “Actualizar”, “ActualizarMedicamento” y “GenerarReporte” se incluye el uso de la clase “**Trace**”, escribiendo mensajes informativos sobre la ejecución de cada uno de los métodos. Por ejemplo:

```
Trace.WriteLine("Medicamento actualizado " + nombre);
```

Para poder usar la clase “Trace” se incluye la directiva using:

```
using System.Diagnostics;
```

Y para poder hacer uso de las clases Stream se agrega la directiva:

```
using System.IO;
```

En el método “Actualizar”, al leer los datos del archivo de medicamentos, y obtener el valor del campo “precio”, hay que especificar el valor de “**CultureInfo**” para lograr una conversión correcta de los valores con parte decimal:

```
Single precio = Single.Parse(campos[2], CultureInfo.InvariantCulture);
```

En el método “GenerarReporte”, para conformar el contenido de la línea de texto que se debe grabar en el reporte, se trabaja con los métodos “PadLeft”, “PadRight” y “String.Format”. Los métodos “Pad...”, permiten establecer un ancho fijo para cada campo, alineando el valor a mostrar a la derecha o a la izquierda según se trate de campos de tipo numérico o de texto. El método “String.Format” se usa para asignar formatos específicos como, por ejemplo:

```
String.Format("{0,12:F2}", dr["Precio"]); // float con 2 decimales
```

O para formatos de fecha:

```
string.Format("{0:d}", dr["Fecha"]) // formato fecha: dd/mm/yyyy
```

Para más detalles de los formatos admitidos por el método “Format” puede consultar este enlace:

<https://learn.microsoft.com/en-us/dotnet/api/system.string.format?view=netframework-4.8.1>

Una vez finalizada la implementación de estas 4 clases resta hacer uso de las mismas desde el código del formulario, código que resultará bastante simple ya que toda la lógica de acceso a datos y manejo de archivos está implementada en estas clases.

Veamos entonces **el código del formulario**:

En primer lugar, debemos agregar la directiva using para poder trabajar con la clase Trace:

```
using System.Diagnostics; // permite usar Trace
```

Seguidamente revisamos el código del evento “**Load**”:

```
private void Form1_Load(object sender, EventArgs e)
{
    // controlar si la aplicación ya está registrada en EventLog
    if (!EventLog.SourceExists("AppFarmaciaEventLog"))
    {
        // si no está registrada, se agrega como nuevo Listener para Trace
        Trace.Listeners.Add(new EventLogTraceListener("AppFarmaciaEventLog"));
    }
    Trace.AutoFlush = true;
    Trace.Indent();
    Trace.WriteLine("Iniciando aplicación de Farmacia");
    try
    {
        CLaboratorio lab = new CLaboratorio();
        cmbLaboratorio.DisplayMember = "Nombre";
        cmbLaboratorio.ValueMember = "Laboratorio";
        cmbLaboratorio.DataSource = lab.GetLaboratorios();
        lab.Dispose();
        Trace.WriteLine("Datos de Laboratorios obtenidos " +
        cmbLaboratorio.Items.Count.ToString());
    }
    catch (LaboratorioException lex)
```

```

    {
        MessageBox.Show(lex.Message);
    }
}

```

Acá se agrega el listener “EventLogTraceListener” para obtener las salidas de Trace en el visor de eventos de Windows, luego se crea una instancia de la clase “CLaboratorio” y se ejecuta el método “GetLaboratorios” para vincular la tabla al control ComboBox de Laboratorio. La captura de excepciones se realiza con la clase propia “LaboratorioException”.

Continuamos con el código del botón “**Examinar**”:

```

private void btnExaminar_Click(object sender, EventArgs e)
{
    OpenFileDialog dlg = new OpenFileDialog();
    dlg.Title = "Archivo de Medicamentos";
    dlg.Filter = "Archivos txt|*.txt";
    dlg.FilterIndex = 0;
    dlg.InitialDirectory = Application.StartupPath;
    dlg.RestoreDirectory = true;
    if(dlg.ShowDialog() == DialogResult.OK)
    {
        txtArchivo.Text = dlg.FileName;
        btnActualizar.Enabled = true;
        btnReporte.Enabled = true;
        Trace.WriteLine("Se seleccionó el archivo " + dlg.FileName);
    }
    else
    {
        txtArchivo.Text = "";
        btnActualizar.Enabled = false;
        btnReporte.Enabled = false;
    }
}

```

Este botón de comando permite buscar y seleccionar el archivo de medicamentos que se usará para actualizar los medicamentos, se crea por código el objeto “OpenFileDialog”, se establecen sus principales propiedades y se visualiza ejecutando el método “ShowDialog”, si la respuesta es OK el nombre del archivo seleccionado se asigna al control TextBox y se habilitan los demás botones de comando. Con Trace se informa el nombre del archivo seleccionado.

Código del botón “**Actualizar**”:

```

private void btnActualizar_Click(object sender, EventArgs e)
{
    try
    {
        CActualizacion actualizacion = new CActualizacion();
    }
}

```

```

        bool res = actualizacion.Actualizar(txtArchivo.Text,
            cmbLaboratorio.Text,
            int.Parse(cmbLaboratorio.SelectedValue.ToString()));
        if (res)
        {
            MessageBox.Show("Actualización finalizada correctamente");
        }
        actualizacion.Dispose();
    }
    catch (ActualizacionException aex)
    {
        MessageBox.Show(aex.Message);
    }
}

```

En este procedimiento se crea una nueva instancia de la clase “CActualizacion”, luego se ejecuta el método “Actualizar” y se pasa por parámetro el nombre del archivo que contiene los medicamentos, el nombre del laboratorio seleccionado para la actualización, y el número del laboratorio seleccionado. El método “Actualizar” realizará su tarea y devolverá verdadero si finaliza correctamente, en caso contrario generará una excepción.

Código del botón “**GenerarReporte**”:

```

private void btnReporte_Click(object sender, EventArgs e)
{
    SaveFileDialog dlg = new SaveFileDialog();
    dlg.Title = "Guardar Reporte";
    dlg.Filter = "Archivos de Reporte (.txt)|*.txt";
    dlg.FilterIndex = 0;
    dlg.InitialDirectory = Application.StartupPath;
    if (dlg.ShowDialog() == DialogResult.OK)
    {
        try
        {
            CActualizacion actualizacion = new CActualizacion();
            actualizacion.GenerarReporte(dlg.FileName);
            MessageBox.Show("Reporte Generado. (" + dlg.FileName + ")");
            actualizacion.Dispose();
            Trace.WriteLine("Reporte generado: " + dlg.FileName);
        }
        catch (ActualizacionException aex)
        {
            MessageBox.Show(aex.Message);
        }
        finally
        {
            dlg.Dispose();
        }
    }
}

```

Para generar el reporte solicitado usamos un objeto de la clase “SaveFileDialog” con el que el usuario seleccionará la ubicación y el nombre del archivo que contendrá el reporte,

seguidamente se crea una instancia de la clase “CActualizacion” y se invoca el método “GenerarReporte” pasando por parámetro el nombre del archivo a crear.

Código del evento “SelectedIndexChanged” del control ComboBox:

```
private void cmbLaboratorio_SelectedIndexChanged(object sender, EventArgs e)
{
    Trace.WriteLine("Se seleccionó el laboratorio " + cmbLaboratorio.Text);
    txtArchivo.Text = "";
    btnActualizar.Enabled = false;
    btnReporte.Enabled = false;
}
```

En este evento, que se ejecuta cada vez que el usuario selecciona otro laboratorio de la lista desplegable, se actualiza el estado del TextBox y los botones de comando, también se hace uso de Trace para informar el laboratorio seleccionado.

El código completo del formulario queda de esta forma:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;
using System.Diagnostics; // permite usar Trace

namespace SP1
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void Form1_Load(object sender, EventArgs e)
        {
            // controlar si la aplicación ya está registrada en EventLog
            if (!EventLog.SourceExists("AppFarmaciaEventLog"))
            {
                // si no está registrada, se agrega como nuevo Listener para Trace
                Trace.Listeners.Add(new EventLogTraceListener("AppFarmaciaEventLog"));
            }
            Trace.AutoFlush = true;
            Trace.Indent();
            Trace.WriteLine("Iniciando aplicación de Farmacia");
            try
            {
                CLaboratorio lab = new CLaboratorio();
                cmbLaboratorio.DisplayMember = "Nombre";
                cmbLaboratorio.ValueMember = "Laboratorio";
                cmbLaboratorio.DataSource = lab.GetLaboratorios();
                lab.Dispose();
            }
            catch { }
        }
    }
}
```



```

        Trace.WriteLine("Datos de Laboratorios obtenidos " +
cmbLaboratorio.Items.Count.ToString());
    }
    catch(LaboratorioException lex)
    {
        MessageBox.Show(lex.Message);
    }
}

private void btnActualizar_Click(object sender, EventArgs e)
{
    try
    {
        CActualizacion actualizacion = new CActualizacion();
        bool res = actualizacion.Actualizar(txtArchivo.Text,
            cmbLaboratorio.Text,
            int.Parse(cmbLaboratorio.SelectedValue.ToString()));
        if (res)
        {
            MessageBox.Show("Actualización finalizada correctamente");
        }
        actualizacion.Dispose();
    }
    catch(ActualizacionException aex)
    {
        MessageBox.Show(aex.Message);
    }
}

private void btnExaminar_Click(object sender, EventArgs e)
{
    OpenFileDialog dlg = new OpenFileDialog();
    dlg.Title = "Archivo de Medicamentos";
    dlg.Filter = "Archivos txt|*.txt";
    dlg.FilterIndex = 0;
    dlg.InitialDirectory = Application.StartupPath;
    dlg.RestoreDirectory = true;
    if(dlg.ShowDialog() == DialogResult.OK)
    {
        txtArchivo.Text = dlg.FileName;
        btnActualizar.Enabled = true;
        btnReporte.Enabled = true;
        Trace.WriteLine("Se seleccionó el archivo " + dlg.FileName);
    }
    else
    {
        txtArchivo.Text = "";
        btnActualizar.Enabled = false;
        btnReporte.Enabled = false;
    }
}

private void btnReporte_Click(object sender, EventArgs e)
{
    SaveFileDialog dlg = new SaveFileDialog();
    dlg.Title = "Guardar Reporte";
    dlg.Filter = "Archivos de Reporte (.txt)|*.txt";
    dlg.FilterIndex = 0;
    dlg.InitialDirectory = Application.StartupPath;
    if(dlg.ShowDialog() == DialogResult.OK)
    {
        try

```

```

        {
            CActualizacion actualizacion = new CActualizacion();
            actualizacion.GenerarReporte(dlg.FileName);
            MessageBox.Show("Reporte Generado. (" + dlg.FileName + ")");
            actualizacion.Dispose();
            Trace.WriteLine("Reporte generado: " + dlg.FileName);
        }
        catch (ActualizacionException aex)
        {
            MessageBox.Show(aex.Message);
        }
        finally
        {
            dlg.Dispose();
        }
    }

    private void cmbLaboratorio_SelectedIndexChanged(object sender, EventArgs e)
    {
        Trace.WriteLine("Se seleccionó el laboratorio " + cmbLaboratorio.Text);
        txtArchivo.Text = "";
        btnActualizar.Enabled = false;
        btnReporte.Enabled = false;
    }
}

```

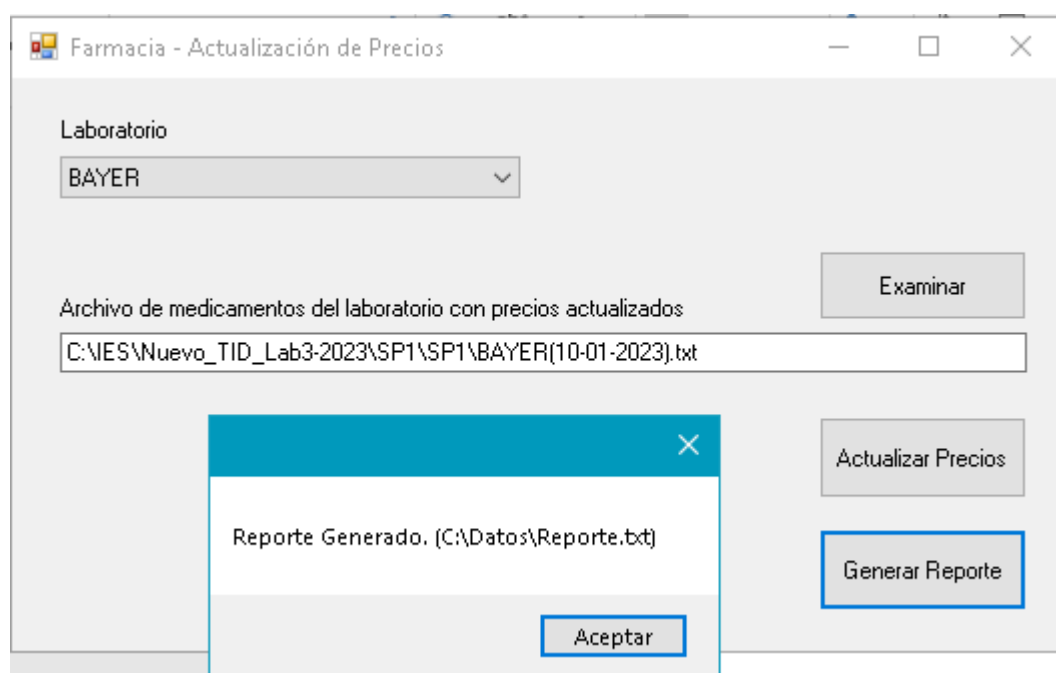
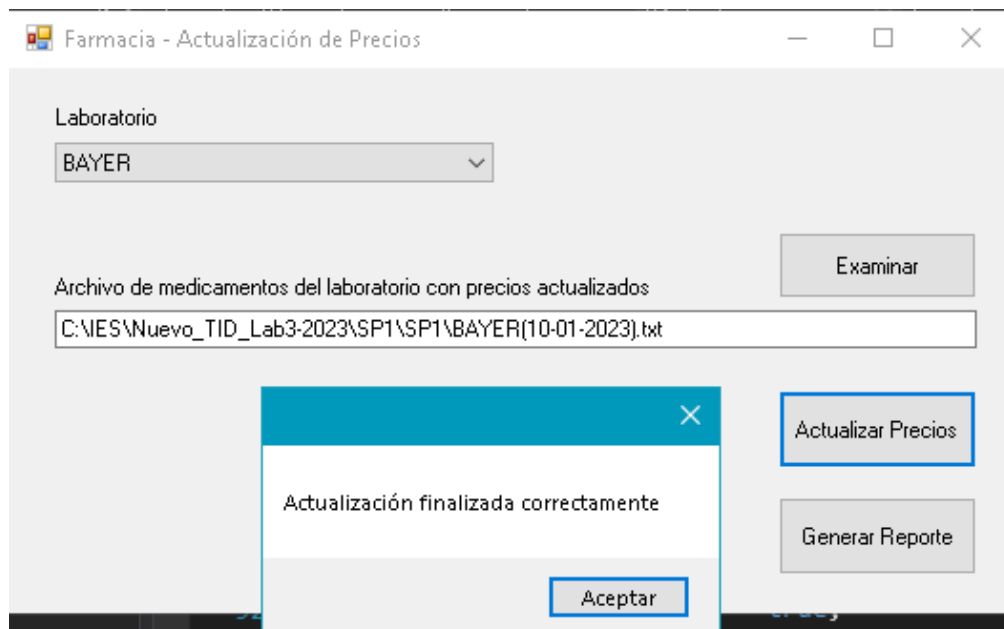
De esa forma se completa todo el desarrollo de la aplicación solicitada, podemos ejecutarla y verificar su funcionamiento y resultados.

Elegimos este archivo de medicamentos para actualizar la base de datos:

```

BAYER(10-01-2023).txt
1010002,"GENIOL X 3 TABLETAS",29.50
1010001,"ASPIRINA X 6 TABLETAS",25.00
1010004,"ASPIRINETA X 3 TABLETAS",27.25
1010010,"VITAMINA C 1G",250.00
1010003,"CAFIASPIRINA X 6 TABLETAS",21.50
1010009,"VITAMINA C 500mG",175.00

```



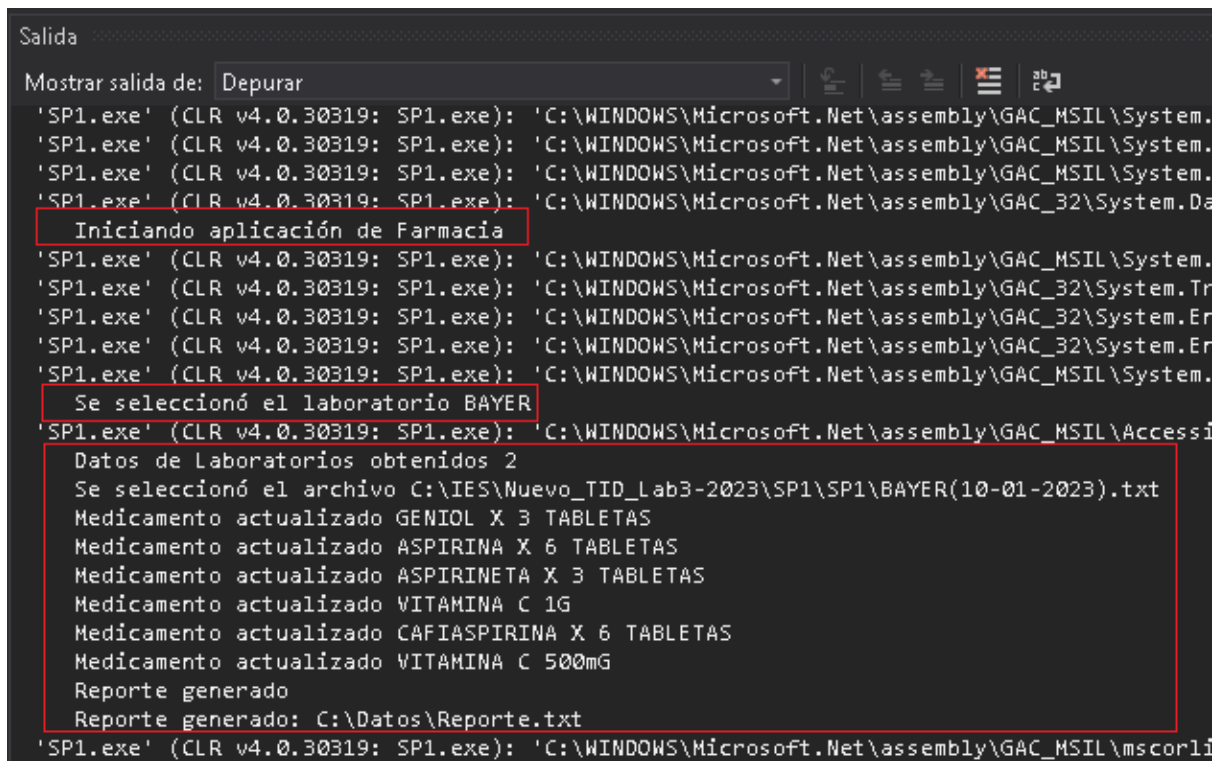
El archivo del reporte generado (Reporte.txt) contiene estos datos:

Reporte de Medicamentos Actualizados

```
-----
Código Nombre                               Lab    Precio    Fecha
1010002 GENIOL X 3 TABLETAS                101     29,50    10/01/2023
1010001 ASPIRINA X 6 TABLETAS              101     25,00    10/01/2023
1010004 ASPIRINETA X 3 TABLETAS            101     27,25    10/01/2023
1010010 VITAMINA C 1G                       101    250,00    10/01/2023
1010003 CAFIASPIRINA X 6 TABLETAS          101     21,50    10/01/2023
1010009 VITAMINA C 500mg                    101    175,00    10/01/2023
```

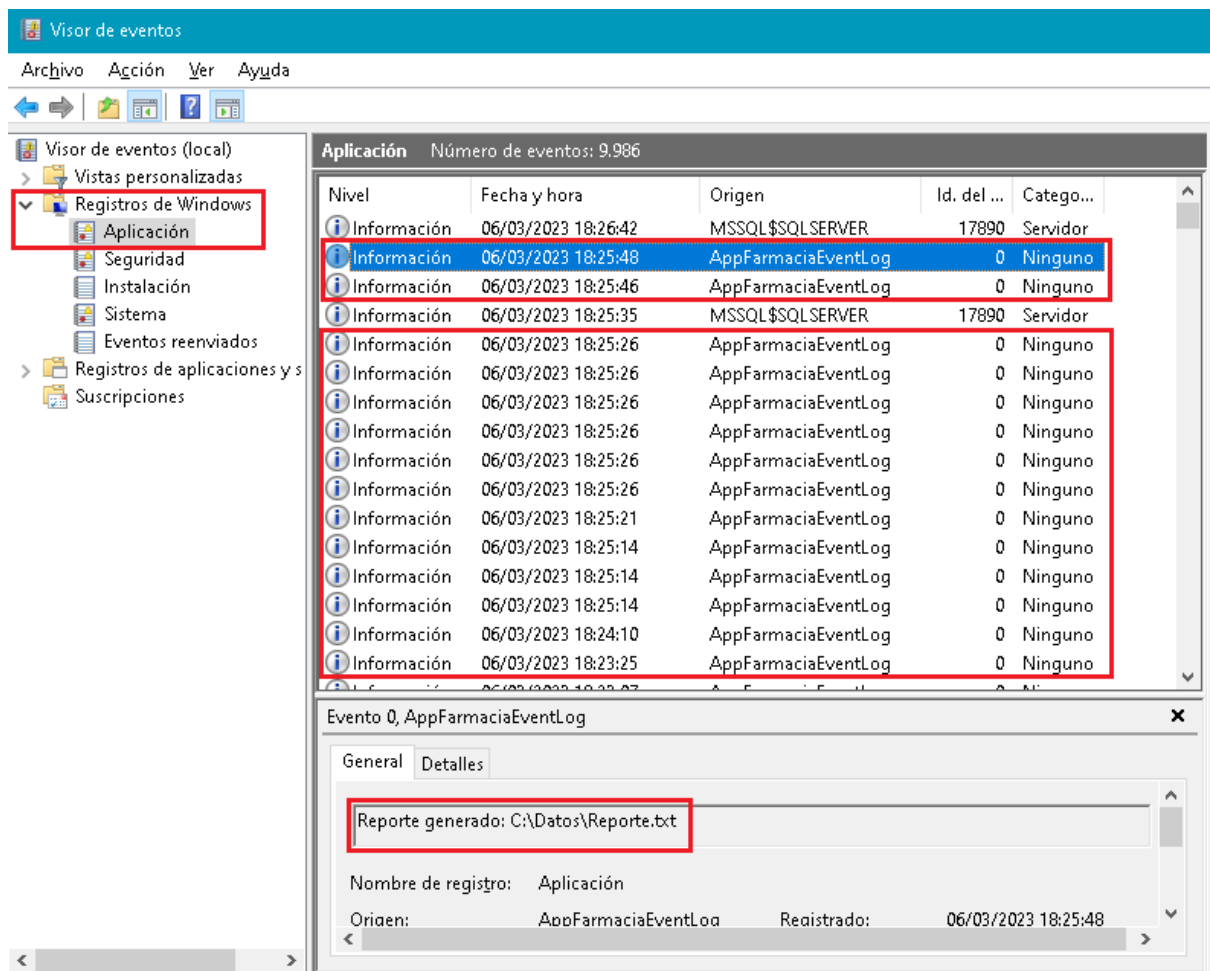
Fecha del reporte: 06/03/2023

La consola de salida muestra esta información generada por Trace:



```
'SP1.exe' (CLR v4.0.30319: SP1.exe): 'C:\WINDOWS\Microsoft.Net\assembly\GAC_MSIL\System.
'SP1.exe' (CLR v4.0.30319: SP1.exe): 'C:\WINDOWS\Microsoft.Net\assembly\GAC_MSIL\System.
'SP1.exe' (CLR v4.0.30319: SP1.exe): 'C:\WINDOWS\Microsoft.Net\assembly\GAC_MSIL\System.
'SP1.exe' (CLR v4.0.30319: SP1.exe): 'C:\WINDOWS\Microsoft.Net\assembly\GAC_32\System.Da
Iniciando aplicación de Farmacia
'SP1.exe' (CLR v4.0.30319: SP1.exe): 'C:\WINDOWS\Microsoft.Net\assembly\GAC_MSIL\System.
'SP1.exe' (CLR v4.0.30319: SP1.exe): 'C:\WINDOWS\Microsoft.Net\assembly\GAC_32\System.Tr
'SP1.exe' (CLR v4.0.30319: SP1.exe): 'C:\WINDOWS\Microsoft.Net\assembly\GAC_32\System.Er
'SP1.exe' (CLR v4.0.30319: SP1.exe): 'C:\WINDOWS\Microsoft.Net\assembly\GAC_32\System.Er
'SP1.exe' (CLR v4.0.30319: SP1.exe): 'C:\WINDOWS\Microsoft.Net\assembly\GAC_MSIL\System.
Se seleccionó el laboratorio BAYER
'SP1.exe' (CLR v4.0.30319: SP1.exe): 'C:\WINDOWS\Microsoft.Net\assembly\GAC_MSIL\Accessi
Datos de Laboratorios obtenidos 2
Se seleccionó el archivo C:\IES\Nuevo_TID_Lab3-2023\SP1\SP1\BAYER(10-01-2023).txt
Medicamento actualizado GENIOL X 3 TABLETAS
Medicamento actualizado ASPIRINA X 6 TABLETAS
Medicamento actualizado ASPIRINETA X 3 TABLETAS
Medicamento actualizado VITAMINA C 1G
Medicamento actualizado CAFIASPIRINA X 6 TABLETAS
Medicamento actualizado VITAMINA C 500mG
Reporte generado
Reporte generado: C:\Datos\Reporte.txt
'SP1.exe' (CLR v4.0.30319: SP1.exe): 'C:\WINDOWS\Microsoft.Net\assembly\GAC_MSIL\mscorlib
```

Y el Visor de Eventos de Windows muestra estos eventos:



Es la misma información que se observa en la consola de salida de Visual Studio, pero acá queda grabada de forma permanente cada vez que se ejecute la aplicación para que pueda ser consultada en el momento que sea necesario.

SP1/Ejercicio por resolver

Se solicita continuar el desarrollo de la aplicación realizada para la farmacia, el objetivo será mejorar sus prestaciones y agregar nuevas funcionalidades.

- Mejorar el proceso de actualización de cada medicamento controlando que el archivo de entrada tenga los valores correctos en todos sus campos: número de medicamento, nombre de medicamento y precio, cada línea deberá tener 3 valores y ser del tipo de dato correcto, en caso contrario se informará el error por medio de Trace, se descartará del proceso esa línea del archivo, pero se continuará procesando el resto de los medicamentos hasta finalizar el archivo.
- Usar los métodos TraceInformation, TraceWarning y TraceError para diferenciar el tipo de mensaje que se genera con Trace.
- Agregar un segundo formulario para la generación de los reportes, el formulario debe permitir filtrar el contenido del reporte por laboratorio, es decir que el usuario debe poder elegir el laboratorio por medio de una lista desplegable y generar el reporte de los medicamentos actualizados de ese laboratorio solamente, se deberá incluir también la opción de generar el reporte con los medicamentos de todos los laboratorios. Agregar al archivo generado una línea que contenga el nombre del laboratorio seleccionado o la leyenda “Todos los laboratorios” según la selección realizada por el usuario.

SP1/Evaluación de paso

1. Indique la opción correcta:

La clave principal de una tabla debe estar formada por un único campo de la tabla.

- Verdadero
- Falso X

2. Indique la opción correcta:

ADO .NET trabajando con entornos desconectados consume menos recursos que en entornos conectados.

- Verdadero X
- Falso

3. Indique la opción correcta:

La clave principal de una tabla debe estar formada por un único campo de la tabla.

- Verdadero
- Falso X

4. Indique la opción correcta:

Para crear un nuevo tipo de excepción se debe heredar de la clase base "Exception".

- Verdadero X
- Falso

5. Indique la opción correcta:

En la clase "OpenFileDialog" la propiedad que define la ubicación del archivo seleccionado es "FilePath".

- Verdadero
- Falso X

6. Indique la opción correcta:

Los registros creados con la clase Trace se pueden diferenciar de tipo usando los métodos "WriteInformation", "WriteWarning" y "WriteError".

- Verdadero
- Falso X

