EXCEPCIONES

La gestión de errores es la técnica que permite interceptar con éxito errores en tiempo de ejecución esperados y no esperados, en C# se llama excepciones. Cuando se produce un error se lanza una excepción.
Cuando un error es detectado las excepciones detienen el flujo actual del programa y deja de funcionar.

Ventaja:
Los mensajes de error no están representados por valores o enumeraciones, sino que en su lugar muestra mensajes concretos, utiliza clases concretas y cada una representa un error de forma clara y evidente.

Bloque Try – Catch

Estos bloque son la solución que ofrece C# a los problemas de tratamiento de errores, la idea consiste en separar físicamente el programa en el flujo normal del programa y el tratamiento de errores.
Las partes del código que podría lanzar excepciones se colocan en un bloque try, y el código para tratamiento de excepciones se pone en el bloque catch. Si se llegara a producir una excepción, el rutime detiene la ejecución normal y empieza a buscar un bloque catch que pueda capturar esa excepción pendiente.

Un bloque try puede causar muchas excepciones, por lo tanto puede que este tenga más que un bloque catch y cada uno captura un tipo específico de excepción, estas capturas solo que basan únicamente en su tipo. Aun así también se puede usar un catch genérico, que captura cualquier excepción, en este caso el control try no puede tener más de un catch genérico.

Trow

Cuando necesita lanzar una excepción, el runtime ejecuta una instrucción throw y lanza una excepción definida por el sistema, es posible utilizar la instrucción throw para lanzar excepciones propias. Se puede utilizar una instrucción throw en un bloque catch para volver a lanzar el mismo objeto excepción u otro nuevo.

Finally

La cláusula finally de C# contiene un conjunto de instrucciones que es necesario ejecutar sea cual sea el flujo de control. Las instrucciones del bloque finally se ejecutarán aunque el control abandone un bucle try o pase lo que pase, siempre se ejecutara.

ARCHIVOS DE TEXTO

Streams

La clase StreamWriter escribe caracteres en archivos de texto y la clase StreamReader lee desde un archivo de texto. Ambas clases se encuentran en el espacio de nombres System.IO.

StreamWriter

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.

StreamReader

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

StreamReader (string path, Encoding e): Ídem anterior, dónde se le 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.

Excepciones que podrían llegar a ocurrir

La ruta de acceso no es válida porque:
- Es una cadena de longitud cero, contiene sólo espacios en blanco, o contiene caracteres no válidos
- La ruta de acceso es Null, el archivo no existem, el nombre del archivo tieneunaato no valido.
- El archivo está en uso por otro proceso o hay un error de E/S.
- El usuario no tiene los permisos necesarios para ver la ruta de acceso.

Objetos útiles

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.

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

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

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.

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

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

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 Integrales

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

Pruebas Funcionales

Una prueba funcional es una prueba basada en la ejecución, revisión y retroalimentación de las funcionalidades previamente diseñadas para el software. Las pruebas funcionales se hacen mediante el diseño de modelos de prueba que buscan evaluar cada una de las opciones con las que cuenta el paquete informático.

Patrón AAA

El patrón AAA (Arrange, Act, Assert) es una forma habitual de escribir pruebas unitarias para un método en pruebas.

- La sección Arrange de un método de prueba unitaria inicializa objetos y establece el valor de los datos que se pasa al método en pruebas.
- La sección Act invoca al método en pruebas con los parámetros organizados.
- La sección Assert comprueba si la acción del método en pruebas se comporta de la forma prevista.

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.

TIPOS GENERICOS

Los genéricos se agregaron a la versión 2.0 del lenguaje C # y el tiempo de ejecución del lenguaje común (CLR). Los genéricos introducen en .NET Framework el concepto de parámetros de tipo, que hacen posible diseñar clases y métodos que difieren la especificación de uno o más tipos hasta que la clase o el método se declara y crea una instancia del código del cliente.

Restricciones

La cláusula where en una definición genérica especifica restricciones en los tipos que se usan como argumentos para los parámetros de tipo en un tipo genérico, método, delegado o función local. Las restricciones pueden especificar interfaces o clases base, o bien requerir que un tipo genérico sea una referencia, un valor o un tipo no administrado. Declaran las funcionalidades que debe poseer el argumento de tipo.

SERIALIZACION

Es el proceso de convertir un objeto en memoria en una secuencia lineal de bytes, esto sirve para pasarlo a otro proceso, otra máquina, grabarlo en disco o grabarlo en una base de datos.

Formatters

Controlan el formato de la serializacion. Existe la serializacion a XML, esta por defecto incluye solo las propiedades y atributos públicos, también está la serializacion binaria, esta por defecto incluyen todos los atributos y propiedades, ya sean públicos o privados.
Se reconstruyen los objetos mediante Deserializacion, puede ser en el mismo proceso o no, lo mismo que con la máquina.

Serializacion XML

Solo serializa los atributos públicos y los valores de propiedad de un objeto en una secuencia XML. Esta no convierte métodos, indexadores, atributos privados ni propiedades de solo lectura.

Al crea una aplicación que utiliza la clase XmlSerializer, debe tener en cuenta los siguientes elementos y sus implicaciones:
- Esta crea archivos .cs y los compila en archivos .dll, la serializacion se produce con esos archivos dll.
- Una clase debe tener un constructor por defecto para que XmlSerializer pueda serializarla.

XmlSerializer

XmlSerializer (System.Type type): Inicializa una nueva instancia de la clase XmlSerializer la cual puede serializar objetos del tipo especificado en el parámetro type.

Serialize (System.IO.Stream stream, Object o): Serializa el objeto especificado 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.

XmlTextReader

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

XmlTextReader (string url): Crea una instancia de XmlTextReader.

Serializacion Binaria

Para poder hacer una serializacion binaria se debe agregar el marcador [Serializable].

BinaryFormatter

Se encuentra en el espacio de nombres System.Runtime.Serialization.Formatters.Binary, este Serializa y Deserializa objetos en formato binario.
Una clase debe tener un constructor por defecto para que BinaryFormatter pueda serializarla.

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.

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.

INTERFACES

Una interfaz contiene definiciones para un grupo de funcionalidades relacionadas que una clase o estructura puede implementar.
Al utilizar interfaces, puede, por ejemplo, incluir el comportamiento de múltiples fuentes en una clase. Esa capacidad es importante en C # porque el lenguaje no admite la herencia múltiple de clases. Además, debe usar una interfaz si desea simular la herencia de las estructuras, porque en realidad no pueden heredar de otra estructura o clase.

Caracterisiticas

- 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).
- No se usa override.
- Se pueden especificar propiedades (sin implementación).
- Las clases pueden implementar varias interfaces.
- Las interfaces pueden “simular” algo parecido a la herencia múltiple.

Implementación explicita

Los miembros implementados explícitamente sirven para ocultar la implementación de miembros de interfaces a las clases que lo implementan. También sirve para evitar la ambigüedad cuando, por ejemplo, una clase implementa dos interfaces las cuales poseen 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.

BASE DE DATOS SQL

La cadena de conexión es donde se especificarán los datos de una conexión a una fuente de datos.

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 y deberá estar asociado a una conexión, en la cual ejecutará sus acciones.

THREADS

Un hilo es simplemente una tarea que puede ser ejecutada al mismo tiempo que otra tarea, los hilos de ejecución que comparten los mismos recursos, sumados a estos recursos, son en conjunto conocidos como un proceso.
El proceso sigue en ejecución mientras al menos uno de sus hilos de ejecución siga activo, en el momento en el que todos los hilos de ejecución finalizan, el proceso no existe más y todos sus recursos son liberados.

Hilos Parametrizados

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

Hilos y Controles Visuales

Si deseamos modificar un control visual de un formulario desde un hilo diferente al principal 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.

EVENTOS

Un evento es una forma en particular que tiene una clase de mostrar cuando ocurre algo en particular en un objeto, el uso más habitual para los eventos lo vemos en las interfaces gráficas. Los eventos proporcionan el hecho de poder señalizar cambios que pueden ser útiles. Un evento es un mensaje enviado por un objeto para indicar que se ha producido una acción invocada programáticamente o por un usuario, cada uno de estos eventos tiene un emisor que es quien lo produce y un receptor que es el que lo captura.

Delegados y Eventos

El objeto que produce el evento se denomina emisor del evento, y el procedimiento que lo captura receptor o manejador. En cualquier caso, el emisor no sabe que objeto o método responderá a los eventos que produzca y por eso es necesario tener un componente que enlace el emisor del evento con el receptor del evento.
El enlace que se utiliza entre estos dos es el delegado, que trabaja como un puntero a función entre emisor y receptor, es posible crear delegados para los casos en que se deseen que un evento utilice diferentes controladores de eventos en diferentes circunstancias.

Delegados

Los eventos se declaran mediante delegados, estos tienen 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 mono que se pueda llamar de forma anónima.
Los delegados son como los punteros de función de C++, pero tienen seguridad de tipos, permiten pasar los métodos como parámetros y pueden encadenarse entre sí. Un Evento puede tener múltiples manejadores y viceversa.

Manejadores

La instrucción += agrega a la lista de invocación del evento del 'emisor', el nuevo manejador.
La instrucción -= quita de la lista de invocación del evento del 'emisor', el manejador.

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.

Implementación

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.