**EXCEPCIONES**

**Gestiones de errores:**

Interceptar errores en tiempo de ejecuciones esperados y no esperados.

Se controla por medio de excepciones

Error lanza excepcion

Programa debe usar diferentes tecnicas de gestion de errores para “Atrapar” las excepciones y administrarlas de manera conveniente

Excepciones detienen el flujo del programa, si no gestiona la excepcion, el programa deja de funcionar

Ej: Division x 0, ruta de archivo erronea (Lanza excepcion)

**OBJ Exception**

Todas las excepciones derivan de EXCEPTION parte de RUNTIME LENGUAJE COMUN (CLR)

Las excepciones no estan representadas por numeros enteros, o enumerados, sino por clases concretas, por ej OutOfMemoryException.

Cada clase puede residir en su propio archivo y no estar vinculada con las demas clases de excepciones

Mensajes de errores significativos.

Cada clase Excepcion es descriptiva y representa error concreto de forma clara y evidente

Cada clase contiene informacion especifica :

FileNotFoundExcepcion – Podria contener el nombre del archivo no encontrado

**BLOQUE TRY – CATCH**

Solucion a problema de tratamiento de errores

Sirve para separar las instrucciones basicas para generar el flujo de control de tratamiento de errores

En el TRY van las intrucciones que podrian lanzar excepciones y en el CATCH el tratamiento ante dicha excepcion

**Try**

**{**

**// CODIGO A CONTROLAR**

**}**

**Catch(Exception identificador)**

**{**

**Control de la excepcion en caso de que se produzca.**

**Tiene que ser System.Exception o clase derivada**

**El identificador es opcional, variable local**

**}**

Al producirse una excepcion, el RUNTIME LENGUAJE COMUN detiene ejecucion y busca bloque Catch para asi capturarla, basandose en su tipo

De no encontrarse en la funcion inmediata, el bloque CATCH, el RUNTIME LENGUAJE COMUN desenreda la pila de llamadas en busqueda de la funcion de llamada, asi sucesivamente, de no encontrar bloque Catch que controle a la Excepcion, se cierra el programa.

Si se encuentra el bloque Catch, se captura excepción y se reanuda ejecución normal desde el bloque catch

Try catch, separa las instrucciones básicas del tratamiento de errores.

**Multiples Catch**

Si existe un bloque try que pueda generar varias excepciones de distintas clases, puede que haya varios Catch que capturen los distintos tipos de excepciones.

La captura de excepción se basa en su tipo

Runtime captura obj Excepcion de un tipo concreto en un bloque Catch para ese tipo

**Catch Generico**

Un bloque **Catch(Exception)** puede capturar cualquier tipo de excepción, independiente a su clase.

No se puede tener mas de un bloque Catch Generico, de existir, este debe ser el ultimo bloque Catch

**Throw**

Cuando se necesita lanzar una excepción el Runtime lenguaje común, realiza la instrucción throw y lanza excepción definida por el sistema.

Interrumpe el flujo normal del programa y transfiere el control al bloque catch que pueda hacerse cargo de la excepción.

Se puede utilizar Throw para lanzar excepciones.

Pueden generar excepciones: CLR, .NET Framework, bibliotecas de otros fabricantes o código de aplicación.

En gral, las excepciones esperan un mensaje como parámetro el cual se puede mostrar o guardar cuando se captura dicha excep.

Es recomendado lanzar excepciones de clase especificas.

Solo se puede lanzar excepción si esta deriva de System.Exception.

Se puede usar throw en bloque catch para relanzar la excepción o lanzar una nueva.

El innerExcepcion, se utiliza para guardar datos de la excepción anteriormente lanzada.

**Bloque Finally**

Contiene instrucciones que deben ejecutarse sea cual sea el flujo de control.

Es útil para no repetir instrucciones y liberar recursos tras lanzamiento excepción.

**TEST UNITARIOS**

Casos de pruebas para verificar el correcto funcionamiento del sistema o subsistema.

Pruebas de **integración**, se realizan una vez que se hayan aprobado las pruebas unitarias, y comprueban que todos los unitarios funcionen correctamente en grupo.

Pruebas funcionales (ejecucion, revisión y retroalimentación de funcionalidades), se realizan mediante diseños de modelos de pruebas que evalúan cada uno de las opciones con las que cuenta el paquete informatico.

**PATRON AAA**

**Arrage:**

**Act:**

**Assert**: clase, determina si el método de prueba se supero o no, a través de métodos estaticos que analizan condición True o False.

Asi también puede ser manejado por atributos o etiquetas. [ExceptedException][Timeout()]

**TIPOS GENERICOS**

**Generics:**

Implementación de tipo de clase parametrizadas.

Es exactamente igual, solo que recibe un parámetro que debe ser especificado en el momento de su declaración.

Utiles si a un no se conoce el tipo de dato a trabajar.

**public class Mensaje<T>**

**{**

**private T miAtributo;**

**}**

Se utilizan restricciones, especificando el tipo de dato a utilizar

**public class Mensajero<T> where T : Mensaje**

**public class Mensajero<T> where T : new()**

**Restricciones existents:**

where T: struct ----- > el argumento debe ser del tipo valor

where T : class ------ > argumento debe ser del tipo referencia

where T : unmanaged ----- > el argumento no debe ser de referencia, asi tampoco ninguno de sus miembros deben serlo

where T : <nombre de clase> ----- > argumento debe ser o debe derivar de tal clase

where T :<nombre interface> ---- > debe ser o implementar interface especificada

where T : U --- > el argumento T debe ser o derivar del argumento U

**Datos Importantes sobre restricciones:**

Los tipos de valor debe tener un constructor accesible sin parámetros.

Restricción struct implica la restricción new(), haciendo asi que la restricción new() no se pueda combinar con la restricción struct.

Restriccion unmanaged implica restricción struct, por lo tanto no se puede combinar con struct o new().

**PARA QUE UN METODO SEA GENERICO NO ES NECESARIO QUE SU CLASE TAMBIEN LO SEA**

**public static void OpTest<T>(T s, T t) : where T : Class**

**INTERFACES**

Establece que en una clase se implementara un conjunto de métodos.

Describe que debería hacer una clase sin especificar como.

Descripcion de métodos que alguna clase puede implementar.

**Datos curiosos:**

* En C# no permite especificar atributo en interfaces.
* Los métodos son públicos (Implicitamente).
* Los métodos son como “Abstractos” ya que no llevan implementación.
* Se pueden especificar propiedades sin implementación.
* Clases pueden implementar varias interfaces.
* Pueden “simular” herencia multiple.
* Se implementa con los : como herencia.
* Si se desea implementar en una clase derivada, se pone la clase madre, IInterface.
* No utiliza OVERRIDE para sobreescrbir métodos ni virtual

Interface INombreInterface

{ }

Por convención, el nombre de la interfaz lleva una “I” antes del identificador.

**Implementacion explicita:**

* Ocultan miembros a clases que la implementan. (No tienen acceso)
* Evitar ambigüedad cuando una clase implementa dos interfaces con miembros con la misma firma
* Clases derivadas de clase que implementa interface no puede sobreescribir métodos explicitos
* Void INombreInterfaz.NombreMetodo() { }
* Nombre de la interface seguida de un .NombreMetodo, no lleva visibilidad
* Se debe castear con el nombre de la Interface si quiere ser usado fuera de esta un método explicito ((IMiInterfaz)obj).MiMetodo(); // **obj.MiMetodo(); NO EXISTE**

**ARCHIVOS DE TEXTO**

**Streams:**

* Clase StreamWriter, escribe caracteres en arch de texto
* Clase StreamReader leer desde archivo texto
* Ambas se encuentran en System.IO;

**StreamWriter**

StreamWriter(string path), el path es donde se creara el archivo, de existir se sobreescribira.

StreamWriter(string path, bool append), path donde se guarda archivo, si el append es true y el archive existe se escribira al final, sino con false y true pero sin existir se creara.

StreamWriter(string path, bool append, Encondig e), idem anterior, se puede especificar el tipo de codificacion.

.Write(string v), escribe cadena sin provocar salto de línea

.WriteLine(string v), escribe una cadena en un archivo provocando salto de línea

.Close(), cierra archivo

Using(StringWriter writer =new StreamWriter(path)) { se encarga de la aperture y cierre del mismo }

**StreamReader**

.StreamReader(path), especifica de donde leera datos.

.StreamReader(string path, Encoding e), idem, especifico codificacion que utilizara para leer.

.Read(), lee un carácter del stream, avanzando carácter a carácter. Retorna entero.

.ReadLine(), lee una línea de caracteres del stream y retorna un string.

.ReadToEnd(), lee todo el stream y retorna un string.

.Close(), cierra obj StreamReader.

**Excepciones:**

Accesos no validos: **ArgumentException**(caracteres no validos, espacios en blanco)

**ArgumentNullException**(ruta de acceso NULL)

Ruta de acceso a File no existe: **FileNotFoundExcepcion**

**DirectoryNoFoundException**

Archivo en uso: **IOException**

Ruta muy larga : **PathTooLongException**

Nombre archivo o ruta de acceso contiene : o formato no valido: **NoSupportedException**

Tema de permisos de acceso: **SecurityException**

**UTILES:**

**File:**

* File.Exists(string path), devuelve true si el path es correcto y tienen los permisos necesarios, false si el path es null, acceso no valido o longitud cero y por ultimo false si no tiene los permisos necesarios sin lanzar exception.
* File.Copy(string, string), copia archivo en uno nuevo, no puede sobreescribir existente
* File.Delete(string), elimina archivo especificado

**Directory:**

* Delete(string path) elimina directorio, siempre y cuando este vacio.
* Delete(string, bool) elimina directorio especificado, y subdirec y archivos si contiene e indica
* Exists(string) verifica si la ruta de acceso a directorio es valida
* GetFiles(string) devuelve nombre de archivos de directorio especificado

**Special Folders**

* Por medio de GetFolderPath de Environment podemos obtener dirección de una carpeta
* A través del enumerado Environment.SpecialFolder podemos acceder a carpetas del sistema sin conocer rutas de archivo:
* EJ: Environments.GetFolderPath(Environment.SpecialFolder.Desktop) = Retorna Path a escritorio

**SERIALIZACION**

Using System.Xml; System.Xml.Serialization;

Convierte objeto en memoria en secuencia lineal de bytes

Sirve para: Pasarlo a otro proceso, a otra maquina, grabarlo en disco/base de datos.

**Serializacion a XML**: serializa solo propiedades y atributos publicos

**Serializacion Binaria**: incluye todos los atributos y propiedades sean públicos o privados.

Se recontruye objeto mediante **Deserializacion**.

**XML:**

No convierte métodos, indexadores, atributos privados ni propiedades solo lectura (excepto Colecciones)

Clase centras es : **XmlSerializer** y sus métodos mas importantes son **Serialize** y **Deserialize**.

XmlSerializer crea archivos .cs y los compila en DLL en el directorio especificado por TEMP

**Se necesita si o si un constructor por defecto para poder serializar.**

Los métodos no se pueden serializar.

* XmlSerializer(System.Type type), recibe typeof(obj), puede serializar información del tipo de dato especificado
* Serialize(System.IO.Stream stream, obj), Serializa el objeto especificado por parametro en el stream especificado.
* Deserialize(system.IO.Stream stream), Deserializa XML contenido en el stream especificado

**XmlTextWriter**

(string filename, System.Text.Enconding encoding)

Crea instancia XmlTextWriter, el filename indica el archive en el cual se escribira, encoding el tipo de codificacion

**XmlTextReader**

Provee manera de Leer archivo XML

(string url)

Crea instancia XmlTextReader, url indica el path del archivo XML

A la hora de leer se debe castear lo devuelto por el Deserialize. (Dato)ser.Deserializer(reader);

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

**SERIALIZACION BINARIA**

Para poder serializar en binario se debe agregar la etiqueta [Serializable] antes de la clase

**BinaryFormatter**

Serializa y Deserializa objetos en formato binario.

Se encuentra en namespace System.Runtime.Serialization.Formatters.Binary

Puede serializar tanto atributos públicos como privados

La clase a serializar debe contar con constructor por defecto

Los métodos de clase mas importante son Serialize y Deserialize

BinaryFormatter(), instancia de clase.

Serialize(System.IO.FileStream serializationStream, Obj), serialize el objeto especificado, y escribe un archivo binario en el stream especificado.

Deserialize(System.IO.FileStream serilizationSteam), deserializa el archivo binario contenido en el stream especificado.

**FileStream**

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

.FileStream(string path, System.IO.FileMode mode), indica ubicacion y modo en que se creara o abrira archive.

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

.Seek(long offset, System.IO.SeekOrigin origin), establece posicion del stream al valor dado)

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

**SQL**

**Conexion a SQL SERVER:**

La cadena de conexion(Connection String) es donde se especifican datos(usuario,servidor,etc) de una conexion con una fuente de datos.

“Data Source = .\\SQLSERVER; Initial Catalog = NombreBaseDeDatos; Integrated Security=True”

Using System.Data.SqlCliente;

**SqlConnection** conexion = new SqlConnection(connectionStr);

**COMMAND:**

Procedimiento almacenado o instruccion Transact-SQL que se ejecuta en una base de datos SQL.

Un comando puede ser de diferentes tipos (Procedimiento Almacenado, etc) = POR AHORA TEXTO

Comando debe estar asociado a una conexion en la cual se ejecutaran sus acciones

**SqlCommand** comando = new SqlCommand();

comando.CommandType = System.Data.CommandType.Text;

comando.Connection = conexion(SqlConnection);

comando.CommandText = “SELECT nombre FROM Personas”;

“UPDATE Personas SET nombre ‘Fer’ WHERE id = 1”;

“INSERT INTO Personas (nombre) VALUES(‘Pedro’);

“DELEFE FROM Personas WHERE id = 1”;

conexion.Open();

comando.ExecuteNonQuery();

**SqlDataReader** oDr = comando.ExecuteReader();

While(oDr.Read())

{ string aux = oDr[“nombre”].toString(); }

**THREAD**

**Hilos de Ejecucion:**

* Hilo, hebra o subproceso es una secuencia de tareas encadenadas muy pequeña
* Tarea que puede ser ejecutada al mismo tiempo que otra tarea
* Hilos de ejecución que comparten los mismo recursos, en conjunto con este ultimo, conforman un proceso
* Proceso sigue en ejecución mientras la menos uno de sus hilos siga activo
* Cuando finalizan todos los hilos, el proceso no existe mas y se liberan recursos.

**Ejemplo Hilo:**

Thread t = new Thread(UnMetodo);

t.Start();

**Ejemplo hilo con parámetros:**

Thread t = new Thread(new ParametrizedThreadStart(Metodo));

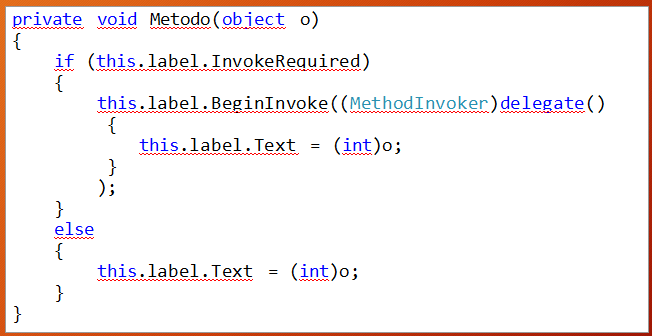
t.Start(1000);

**Metodo en cuestión:**

Prívate void Metodo(obj o)

{ Console.WriteLine((int)o); }

**Ejemplo con FORM:**



**EVENTOS**

Modo que tiene una clase de proporcionar información a sus clientes cuanto ocurre algo en particular dentro del objeto.

Uso mas habitual en interfaces graficas, por Ej Evento click, Evento Load.

Medio para señalar cambio de estado de un objeto, los cuales pueden ser utiles para los clientes de este.

Cada evento tiene un emisor que provoca el evento y un receptor que lo captura.

**Delegados y Eventos:**

Objeto que desencadena evento es el Emisor.

Procedimiento que captura el evento es el Receptor, manejador del evento.

Emisor no sabe que objeto o método respordera al evento que produzca, es necesario tener componente que enlace el emisor del evento con el receptor.

Framework .NET utiliza delegado para trabajar como puntero a la función entre emisor y receptor.

Se pueden crear delegados para que eventos utilicen diferentes controladores en diferentes circunstancias.

Los eventos de declaran mediante delegados.

Delegado es un tipo que representa referencia a métodos con una lista de parámetros determinada y un tipo devuelto.

Delegado encapsula método para que se llame de forma anónima.

Evento - > Delegado -> Cliente

Delegados permiten para métodos por parámetro, pueden encadenarse entre si, por ejemplo llamar a varios métodos en un solo evento.

Un evento puede tener multiples manejadores.

**Manejadores:**

Para asociar manejador a evento, se debe agregar al evento del emisor:

objEmisor.evento += MetodoManejador;

**+=** Para agregar manejador a evento

prívate void MetodoManejador(obj sender, EventArg e) {

MessageBox.Show(“Se uso manejador”);

}

**-=** quita manejador de un evento

objEmisor.evento -= MetodoManejador;

**METODO EXTENSION**

Permiten agregar métodos a tipos ya existentes.

Clase especial de método estatico, pero se llaman como si fueran de instancia.

No existe diferencia entre llamar a método extensión y llamar a método definido de un tipo.

public **static** int Metodo(**this** [tipo] [nombre])

{ return 0; }

Primer parámetro especifica en que tipo funciona el método, precedido del modificador this.

Metodo de extensión se encuentran dentro del ámbito cuando el namespace se importa explícitamente mediante using.

Son validos para clases como para interface