Excepciones

Gestion de errores:

**.** La gestión de errores es la técnica que permite interceptar con éxito errores en tiempo de ejecución esperados y no esperados.

. En C# la gestión de errores se controla por medio de excepciones.

. Cuando se produce un error se ***lanza*** una excepción.

. El programa debe construirse usando diferentes técnicas de gestión de errores para ***atrapar*** las excepciones y administrarlas de manera conveniente.

Excepciones:

. Cuando algo va mal mientras un programa de C# se esta ejecutando, y si no se hace nada, el programa dejara de funcionar.

. Las excepciones detienen el flujo actual del programa, y si no se hace nada, el programa dejara de funcionar.

. Se producen por un error en el programa, por ejemplo, si se divide un numero por cero, o pueden ser el resultado de alguna entrada inesperada, por ejemplo, cuando un usuario selecciona un archivo que no existe.

. El programador debe habilitar su programa para que resuelva estos problemas sin bloquearse.

Objeto Exception:

. Todas las excepciones derivan de la clase **Exception**, que es parte del runtime de lenguaje común (CLR).

**. Ventajas:**

. Los mensajes de error no están representados por valores enteros o enumeraciones.

Los valores enteros de programación, como -3, desaparecen y en su lugar se utilizan clases concretas, como **OutOfMemoryException.**

. Cada clase de excepción puede residir dentro de su propio archivo de origen y no esta vinculada con las demás clases de excepción.

. Se generan mensajes de error significativos.

. Cada clase de excepción es descriptiva y representa un error concreto de forma clara y evidente.

. En lugar de un -3, se utiliza una clase llamada **OutOfMemoryException.**

**.** Cada clase de excepción contiene también información especifica

. Por ejemplo, una clase **FileNotFoundException** podría contener el nombre del archivo no encontrado.

Bloque Try – Catch:

. Los bloques try-catch son la solución que ofrece la orientación a objetos a los problemas de tratamiento de errores.

. La idea consiste en separar físicamente las instrucciones básicas del programa para el flujo de control normar de las instrucciones para tratamientos de errores.

. Asi, las partes del código que podrían lanzar excepciones se colocan en un bloque try, mientras el código para tratamiento de excepciones en el bloque try se pone en el bloque catch aparte.

**Ej:**

void MetodoExcepcion()

{

try

{

Console.WriteLine("Escriba un número");

int i = int.Parse(Console.ReadLine());

}

catch (OverflowException e)

{

Console.WriteLine(e.Message);

}

}

. El bloque ***try*** contiene una expresión que puede generar la excepción.

. En caso de producirse la excepción, el runtime detiene la ejecución normal y empieza a buscar el bloque ***catch*** que pueda capturar la excepción pendiente (basándose en su tipo).

. Si en la función inmediata no se encuentra un bloque ***catch*** adecuado, el runtime desenreda la pila de llamadas en busca de la función de llamada.

. Si tampoco ahí encuentra un bloque ***catch*** apropiado, busca la función que llamo a la función de llamada y así sucesivamente hasta encontrar un bloque ***catch*** (o hasta llegar al final, en cuyo caso se cerrara el programa).

. Si encuentra un bloque ***catch***, se considera que la excepción ha sido capturada y se reanuda la ejecución normal desde el cuerpo del bloque ***catch*** (que, en el caso dado, escribe el mensaje contenido en el objeto excepción OverflowException).

. Por lo tanto, el uso de bloques ***try-catch*** hace que las instrucciones para tratamiento de errores no se mezclen con las instrucciones lógicas básicas, por lo que el programa es mas fácil de interpretar.

Múltiples Catch:

. Un bloque de código en una instancia ***try*** puede contener muchas instrucciones, cada una de las cuales puede producir una o mas clases diferentes de excepción.

. Al haber muchas clases de excepciones distintas, es posible que haya muchos bloques ***catch*** y que cada uno de ellos capture un tipo especifico de excepción.

. La captura de una excepción se basa únicamente en su tipo.

. El runtime captura automáticamente objetos excepción de un tipo concreto en un bloque ***catch*** para ese tipo.

**Ej:**

try

{

Console.WriteLine("Escriba el primer número");

int i = int.Parse(Console.ReadLine());

Console.WriteLine("Escriba el segundo número");

int j = int.Parse(Console.ReadLine());

int k = i / j;

}

catch (OverflowException e)

{

Console.WriteLine(e.Message);

}

catch (DivideByZeroException e)

{

Console.WriteLine(e.Message);

}

Catch genérico:

. Un bloque ***catch*** general (Exception), puede capturar cualquier excepción independiente de su clase y se utiliza con frecuencia para capturar cualquier posible excepción que se pudiera producir por la falta de un controlador adecuado.

. Un bloque ***try*** no puede tener más que un bloque ***catch*** general.

. En caso de existir, un bloque ***catch*** general debe ser el ultimo bloque ***catch*** en el programa.

Throw:

. Cuando necesita lanzar una excepción, el runtime ejecuta una instrucción ***throw*** y lanza una excepción definida por el sistema.

. Esto interrumpe inmediatamente la secuencia de ejecución normal del programa y transfiere el control al primer bloque ***catch*** que pueda hacerse cargo de la excepción en función de su clase.

. Es posible utilizar la instrucción ***throw*** para lanzar excepciones propias.

. Pueden generar excepciones Common Lenguage Runtime (CLR), .NET Framework, las bibliotecas de otros fabricantes o el código de aplicación.

if (minuto < 1 || minuto >= 60)

{

string fallo = minuto + " no es un minuto válido";

throw new TiempoInvalidoException(fallo);

}

. En este ejemplo se emplea la instrucción ***throw*** para lanzar una excepción definida por el usuario, TimpoInvalidoException, si el tiempo analizado no es válido.

. En general, las excepciones esperan como parámetro una cadena con un mensaje significativo que se puede mostrar o quedar registrado cuando se captura la excepción.

. También es conveniente lanzar una clase adecuada de excepción.

catch (Exception e)

{

Throw e;

}

. Solo es posible lanzar un objeto si el tipo de ese objeto deriva directa o indirectamente de **System.Exception.**

**.** Se puede utilizar una instrucción ***throw*** en un bloque ***catch*** para volver a lanzar el mismo objeto excepción u otro nuevo.

catch (IOException e)

{

throw new FileNotFoundException(fallo);

}

. En el ejemplo anterior, el objeto **IOException** y toda la información que contiene se pierde cuando excepción se convierte en un objeto **FileNotFoundException.**

. Es mas conveniente ajustar la excepción, añadiendo nueva información, pero conservando la que tiene en la propiedad InnerException.

catch (IOException e)

{

throw new FileNotFoundException(fallo, e);

}

Bloque 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*** como resultado de la ejecución normal porque el flujo de control llega al final del bloque ***try***.

. Del mismo modo, también se ejecutarán las instrucciones del bloque ***finally*** si el control abandona un bucle ***try*** como resultado de una instrucción ***throw*** o una instrucción de salto como ***break*** o ***continue***.

. El bloque ***finally*** es útil en dos casos: para evitar la repetición de instrucciones y para liberar recursos tras el lanzamiento de una excepción.

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:

. Pruebas integrales o pruebas de integración son aquellas que se realizan en el ámbito del desarrollo del software una vez que se han aprobado las pruebas unitarias y lo que prueban es que todos los elementos unitarios que componen el software, 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.

[TestMethod]

public void Withdraw\_ValidAmount\_ChangesBalance ()

{

// arrange

double current Balance = 10.0;

double withdrawal = 1.0;

double expected = 9.0;

var account = new Checking Account ("JohnDoe", currentBalance);

// act

account.Withdraw(withdrawal);

double actual = account.Balance;

// assert

Assert.AreEqual(expected, actual);

}

. Clase Assert:

. Explicita 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.

. El Assert también puede ser manejado desde los atributos o etiquetas del método.

. A través de los atributos también se puede manejar un tiempo máximo para la prueba.

[TestMethod]

[Timeout(2000)] // Milliseconds o

[Timeout(TestTimeout.Infinite)] // Milliseconds

[ExpectedException(typeof(ArgumentException))]

public void Withdraw\_AmountMoreThanBalance\_Throws()

{

// arrange

var account = new CheckingAccount("John Doe", 10.0);

// act

account.Withdraw(20.0);

// assert es manejado en el ExpectedException

}

Ejecución de las pruebas:

. Abrir la ventana del Explorador de Pruebas:

. Ir al menú Pruebas/Ventanas/Explorador de Pruebas

. Ejecutar Una prueba:

. Click derecho sobre la prueba deseada.

. Ejecutar.

. Ejecutar todas las pruebas juntas:

. Ejecutar Todas…

Tipos Genéricos

Generics:

. Generics es el mecanismo de implementación de clases parametrizadas introducido en la versión 2.0 del lenguaje C#.

. Una clase parametrizada es exactamente igual a una clase de las habituales, salvo por un pequeño detalle: su definición contiene algún método que depende de un parámetro que debe ser especificado en el momento de la declaración de un objeto de dicha clase.

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

. List es una clase parametrizada:

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

. Dictionary es otro ejemplo, con dos parámetros:

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

Generics – Uso simple:

public class Mensaje<T>

{

private T miAtributo;

}

// ...

Mensaje<string> tipoTexto = new Mensaje<string>();

Mensaje<MiClase> tipoMio = new Mensaje<MiClase>();

Generics – Uso menos simple:

public class Mensajero<T, U>

{

private T miAtr1;

private U miAtr2;

private Dictionary<T, U> miDiccionario;

}

Mensaje<string, int> tipoTexto = new Mensaje<string, int>();

Mensaje<char, MiClase> tipoMio = new Mensaje<char, MiClase>();

Restricciones:

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

public class Mensajero<T> where T : Mensaje

{ }

class EjemploComplejo<K, V, U>

// Implemente interfaz

where U : System.IComparable<U>

// V tenga constructor por defecto

where V : new()

{ }

Tipos de restricciones:

where T : struct

El argumento de tipo debe ser un tipo de valor.

where T : class

El argumento de tipo debe ser un tipo de referencia.

where T : unmanaged

El argumento de tipo no debe ser un tipo de referencia y no debe contener ningún miembro de tipo de referencia en ningún nivel de anidamiento.

where T : new()

El argumento de tipo debe tener un constructor sin parámetros público. Cuando se usa conjuntamente con otras restricciones, la restricción new() debe especificarse en último lugar.

where T : <nombre de clase base>

El argumento de tipo debe ser o derivarse de la clase base especificada.

where T : <nombre de interfaz>

El argumento de tipo debe ser o implementar la interfaz especificada. Pueden especificarse varias restricciones de interfaz. La interfaz de restricciones también puede ser genérica.

where T : U

El argumento de tipo proporcionado por T debe ser o derivarse del argumento proporcionado para U.

. Algunas de estas restricciones son mutuamente excluyentes.

. Todos los tipos de valor deben tener un constructor sin parámetros accesible.

. La restricción struct implica la restricción new() y la restricción new() no se puede combinar con la restricción struct.

. La restricción unmanaged implica la restricción struct.

. La restricción unmanaged no se puede combinar con las restricciones struct o new().

Metodos y Generics

. Para que un método sea genérico, no hace falta que la clase lo sea también:

class Prueba

{

Public static void OpTest<T>(T s, T t) where T : class

{

System.Console.WriteLine(s == t);

}

public void OpTest2<T>(T s, T t) where T : class

{

System.Console.WriteLine(s == t);

} }

Interfaces

. Es un contrato que establece una clase en el cual esta clase asegura que implementara un conjunto de métodos.

. Son una manera de describir que debería hacer una clase sin especificar como.

. Es la descripción de uno o más métodos que posteriormente alguna clase puede implementar.

Generalidades:

. C# no permite especificar atributos en las interfaces.

. Todos los métodos **son públicos** (no se permite especificarlo).

. Todos los métodos son como “abstractos” ya que no cuentan con implementación (no se permite especificarlo).

. Se pueden especificar propiedades (sin implementación).

. Las clases pueden implementar varias interfaces.

. Las interfaces pueden “simular” algo parecido a la herencia múltiple.

Definir una interfaz:

[modificadores] interface INombreInterface

{

//Miembros de la interface

}

. Se utiliza la palabra reservada **interface.**

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

[modificadores] class NombreClase : INombreInterface

{

//Miembros de la interface

}

. Para que una clase implemente una interface se emplea el operador dos puntos (:).

. Para implementar una interface a una clase derivada, primero hay que indicar la clase base, luego la interface separadas por coma (,).

. Para sobreescribir los miembros de las interfaces **NO** se emplea la palabra **override**, ya que no fueron declaradas como *virtual* o *abstract* en la interface:

[modificadores] class NombreClase : IMiInterface1, IMiInterface2

{ }

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 explicita no pueden sobrescribir los métodos definidos explícitamente.

. Sintactimente 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.

Interfaz explicita:

void INombreInterface.NombreMetodo()

{

Console.WriteLine("Hola");

}

. Se coloca el nombre de la interfaz adelante del nombre del método que se esta implementando explícitamente y además no se le indica la visibilidad.

public interface IMiInterfaz

{

string MiMetodo();

}

public class PruebaInterfazImplicita : IMiInterfaz

{

string IMiInterfaz.MiMetodo()

{

return "Hola"; } }

Archivos de Texto

Streams:

. La clase **StreamWriter** escribe caracteres en archivos de texto.

. 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 especifico. Si el archivo existe, se sobrescribirá, sino se creará.

**. StreamWriter(string path, bool append)**

. Ídem anterior, si append es true, se agregará datos al archivo existente. Caso contrario, se sobrescribirá el archivo.

**. StreamWriter(string path, bool append, Encoding e)**

. Ídem anterior, donde se le puede especificar el tipo de codificación que se utilizara al escribir en el archivo.

using System.IO;

// ...

// Abro el archivo ubicado en una dirección de la máquina

StreamWriter sw = new StreamWriter("C:\\prueba.txt");

// Agrego una línea de texto

sw.WriteLine("Hola mundo!!");

// Agrego otra línea de texto

sw.WriteLine("Chau mundo!!");

// Cierro el archivo

sw.Close();

using System.IO;

// ...

// using maneja el archivo, encargándose de cerrarlo al finalizar

using (StreamWriter writer = new StreamWriter("bitacora.txt"))

{

writer.Write("Palabra ");

writer.WriteLine("Otras Palabras");

writer.WriteLine("Línea");

}

**. 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.

using System.IO;

// ...

// Abro el archivo ubicado en una dirección de la máquina

StreamReader sw = new StreamReader("C:\\prueba.txt");

// Leo una línea de texto

sr.ReadLine();

sw.Close(); // Cierro el archivo

Excepciones:

**. La ruta de acceso no es válida porque...**

. Es una cadena de longitud cero, contiene sólo espacios en blanco, o contiene

caracteres no válidos. (ArgumentException).

. La ruta de acceso es **Null** (ArgumentNullException).

**. File** **señala a una ruta de acceso que no existe.**

. FileNotFoundException

. DirectoryNotFoundException

**. El archivo está en uso por otro proceso o hay un error de E/S**

. IOException

**. La ruta supera la longitud máxima definida por el sistema**

. PathTooLongException

**. Un nombre de archivo o de directorio de la ruta de acceso contiene un signo de dos puntos (:) o tiene un formato no válido**

. NotSupportedException

**. El usuario no tiene los permisos necesarios para ver la ruta de acceso**

. SecurityException

Objetos útiles

File:

**. File.Exists(string path):**

. Es true si tiene los permisos necesarios y path contiene el nombre de un archivo existente; de lo contrario, es false.

. También devuelve false si path es null, una ruta de acceso no válida o una cadena de

longitud cero.

. Si no tiene permisos suficientes para leer el archivo especificado, no se produce ninguna

excepción y el método devuelve false, independientemente de la existencia del path.

**. File.Copy(string, string):**

. Copia un archivo existente en un archivo nuevo. No se permite sobrescribir un archivo del

mismo nombre.

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

Directory:

**. Directory.Delete(string path):**

. Elimina el directorio especificado, siempre y cuando esté vacío.

**. Directory.Delete(string, boolean):**

. Elimina el directorio especificado y, si está indicado, los subdirectorios y archivos que

contiene.

**. Directory.Exists(string):**

. Determina si la ruta de acceso dada hace referencia a un directorio existente en el disco.

**. GetFiles(String):**

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

Special Folders:

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

. **Environment.GetFolderPath**

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

. **Environment.GetFolderPath(Environment.SpecialFolder.Desktop)** //retorna el path al escritorio.

. **Environment.GetFolderPath(Environment.SpecialFolder. MyDocuments)** //carpeta de Mis Documentos

. **Environment.GetFolderPath(Environment.SpecialFolder. ProgramFiles)** // directorio de archivos de programa.

Serialización

. ¿Qué es?

. Es el proceso de convertir un objeto en memoria en una secuencia lineal de bytes.

. ¿Para qué sirve?

. Para pasarlo a otro proceso.

. Para pasarlo a otra máquina.

. Para grabarlo en disco.

. Para grabarlo en una base de datos.

Formatters

. Controlan el formato de la serialización.

. **Serialización a XML**

. Por defecto incluye solo las propiedades y atributos públicos.

. **Serialización Binaria**

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

. ¿Y después?

. Se construye el objeto mediante Deserialización – proceso inverso.

. Puede ser en el mismo proceso o no, en la misma maquina o no.

Serializacion XML

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

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

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

**.** La secuencia **XML** que genera **XmlSerializer** cumple con la recomendación 1.0 del W3C([www.w3.org](http://www.w3.org)) acerca del lenguaje de definición de esquemas **XML** (XSD).

. Además, los tipos de datos generados cumplen las especificaciones enumeradas en el momento titulado “XML Schema Part 2: Datatypes”.

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

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

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

. Solo pueden serializar los atributos y propiedades públicas.

. Los métodos no se pueden serializar.

XMLSerializer

**. XmlSerializer(Sytem.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 contedito de datos **XML**.

. Métodos:

. XmlTextWriter (string filename, System.Text.Encoding encoding)

. Crea una instancia de XmlTextWriter.

. El filename indica en que archivo se escribirá.

. Con encoding se indicará cual será la codificación.

**EJ:**

using System.Xml;

using System.Xml.Serialization;

Dato p = new Dato(); //Objeto a serializar.

XmlTextWriter writer; //Objeto que escribirá en XML.

XmlSerializer ser; //Objeto que serializará.

//Se indica ubicación del archivo XML y su codificación.

writer = new XmlTextWriter(ArchivoXml, Encoding.UTF8);

//Se indica el tipo de objeto ha serializar.

ser = new XmlSerializer(typeof(Dato));

//Serializa el objeto p en el archivo contenido en writer.

ser.Serialize(writer, p);

//Se cierra la conexión al archivo

writer.Close();

XmlTextReader

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

. Métodos:

. XmlTextReader (string url)

. Crea una instancia de XmlTextReader.

. El url indica en que archivo están los datos XML.

**Ej:**

using System.Xml;

using System.Xml.Serialization;

//...

//Objeto que alojará los datos contenidos en el archivo XML.

Dato aux = new Dato();

XmlTextReader reader; //Objeto que leerá XML.

XmlSerializer ser; //Objeto que Deserializará.

//Se indica ubicación del archivo XML.

reader = new XmlTextReader(ArchivoXml);

//Se indica el tipo de objeto ha serializar.

ser = new XmlSerializer(typeof(Dato));

//Deserializa el archivo contenido en reader, lo guarda en aux.

aux = (Dato)ser.Deserialize(reader);

//Se cierra el objeto reader.

reader.Close();

Serialización Binaria

. Para poder hace runa serialización binaria se debe agregar el marcador [Serializable]

[Serializable]

class MiClase

{

}

BinaryFormatter

. Serializa y Deserializa objetos en formato binario.

. Se encuentra en el espacio de nombres System.Runtime.Serialization.Formatters.Binary.

. Puede serializar atributos públicos y privados.

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

. Los métodos más importantes de la clase BinaryFormatter son:

. Serialize

. Deserialize

Métodos BinaryFormatter

**. BinaryFormatter()**

. Inicializa una nueva instancia de la clase BinaryFormatter.

**. Serialize (System.IO.FileStream seralizationStream, Object graph)**

. Serializa el objeto especificado y escribe en un archivo binario usando el serializationStream especificado.

**. Deserialize (System.IO.FileStream serializationStream)**

. Deserializa el archivo binario contenido por el serializationStream especificado.

FileStream

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

. Métodos:

. FileStream (string path, System.IO.FileMode mode)

. Inicializa una instancia de FileStream, indicando ubicación y el modo en que se

creará o abrirá el archivo.

. Read (byte[] array, int offset, int count)

. Lee un bloque de bytes y escribe los datos en el buffer dado.

. Seek (long offset, System.IO.SeekOrigin origin)

. Establece la posición del stream al valor dado.

. Write (byte[] array, int offset, int count)

. Escribe un bloque de bytes en el stream.

Ejemplo Serialización

using System.IO;

using System.Runtime.Serialization.Formatters.Binary;

//...

Dato aux = new Dato(); // Objeto a serializar.

Stream fs;

// Objeto que escribirá en binario.

BinaryFormatter ser; // Objeto que serializará.

//Se indica ubicación del archivo binario y el modo.

fs = new FileStream(ArchivoBinario, FileMode.Create);

//Se crea el objeto serializador.

ser = new BinaryFormatter();

//Serializa el objeto p en el archivo contenido en fs.

ser.Serialize(fs, p);

//Se cierra el objeto fs

fs.Close();

Ejemplo Deserialización

//Objeto que alojará los datos contenidos en el archivo binario.

Dato aux = new Dato();

Stream fs; // Objeto que leerá en binario.

BinaryFormatter ser; // Objeto que Deserializará.

//Se indica ubicación del archivo binario y el modo.

fs = new FileStream(pathBinario, FileMode.Open);

//Se crea el objeto deserializador.

ser = new BinaryFormatter();

//Deserializa el archivo contenido en fs, lo guarda en aux.

aux = (Dato)ser.Deserialize(fs);

//Se cierra el objeto fs.

fs.Close();

Conexión a SQL Server

. La cadena de conexión (Connection String) es donde se especificaran los datos (usuario, servidor, etc) de una conexión a una fuente de datos.

. Para SQL Server, a fin de ejemplo, utilizaremos cadenas similares a esta:

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

. Para generar una cadena diferente puede seguir el Wizard de la pestaña Data Source de Visual Studio.

Ejemplo de Conexión

using System.Data.SqlClient;

// ...

SqlConnection conexion;

conexion = new SqlConnection(connectionStr);

Command

. Representa un procedimiento almacenado o una instrucción de Transact-SQL que se ejecuta en una base de datos SQL Server.

. Un comando puede ser de diferentes tipos (Procedimiento Almacenado, etc), por ahora solo utilizaremos del tipo texto.

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

**Ej:**

SqlCommand comando;

comando = new SqlCommand();

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

comando.Connection = conexion;

Ejemplo de consulta Select

comando.CommandText = "SELECT nombre FROM Personas";

conexion.Open();

SqlDataReader oDr = comando.ExecuteReader();

whle (oDr.Read())

{

string aux = oDr["nombre"].ToString();

}

Ejemplo Insert, Update y Delete

String consulta;

consulta = "UPDATE Personas SET nombre = 'Fer' WHERE id = 1";

//consulta = "INSERT INTO Personas (nombre)

VALUES('Pedro')";

//consulta = "DELETE FROM Personas WHERE id = 1";

comando.CommandText = consulta;

conexion.Open();

comando.ExecuteNonQuery();

Threads

Hilos de Ejecucion

. En sistemas operativos, un **hilo de ejecución**, **hebra** o **subproceso** es una secuencia de tareas encadenadas muy pequeña que puede ser ejecutada por un sistema operativo.

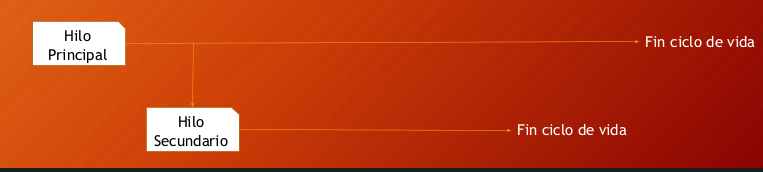
. Un hilo es simplemente una tarea que puede ser ejecutada al mismo tiempo que otra tarea.

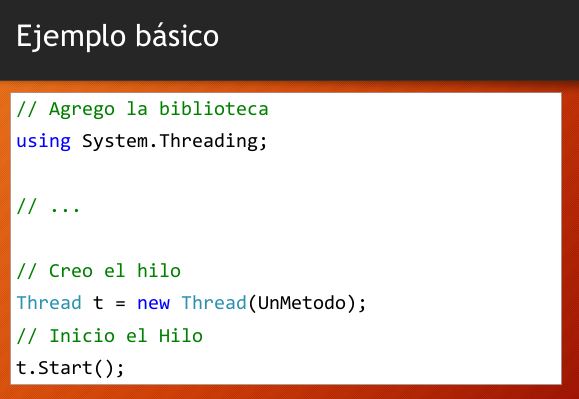
. Los hilos de ejecución que comparten los mismos recursos, sumados a estos recursos, son un conjunto conocido como un proceso.

. Un hilo secundario va tener su propia pila de ejecución, independientemente de la del hilo principal.

. 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.



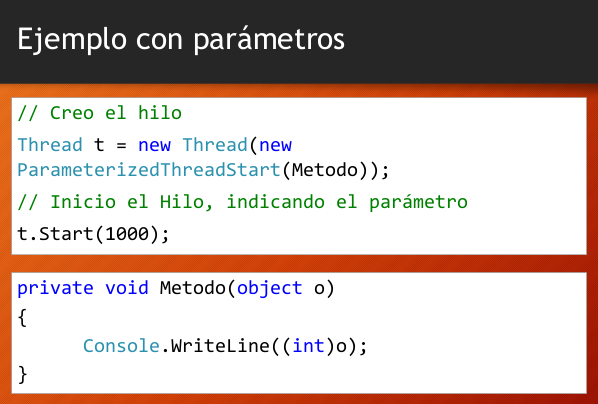


Hilos Parametrizados

. El método utilizado puede tener parámetros.

. Para esto deberemos utilizar ParameterizedThreadStart al instanciar el nuevo hilo.

. El parámetro se pasará mediante el método Start de dicho hilo.

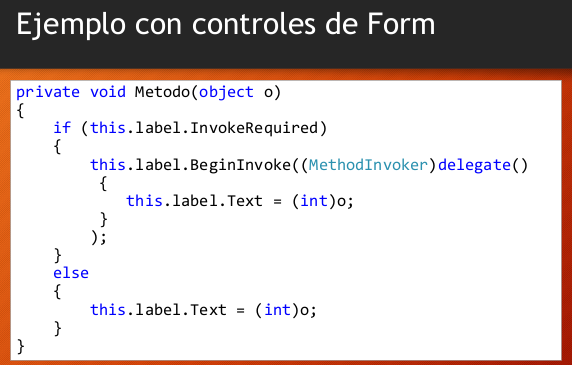


Hilos y Controles Visuales

. Si deseamos modificar un control visual de un formulario (TextBox, ComboBox, Label, etc) desde un hilo diferente al principal (“dueño” de estos controles) deberemos invocar a dicho hilo.

. Para esto le consultaremos al control si necesita ser invocado el hilo principal (InvokedRequired).

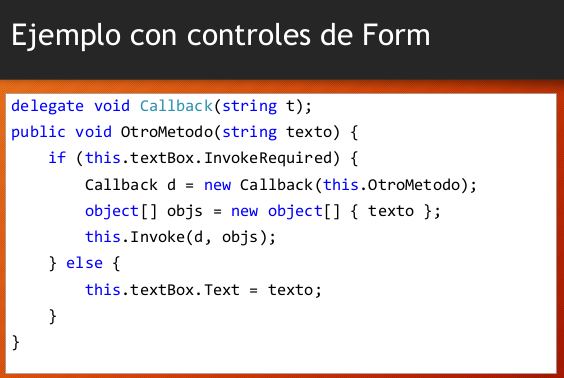
. Luego invocaremos dicho hilo (BeginInvoke) mediante un delegado.



. Dicha invocación puede necesitar parámetros.

. Para resolver este caso, utilizaremos un array de Object.

. Al realizar el Invoke (sincrónico, espera que un Thread finalice para ejecutar otro) o BeginInvoke (Asincrónico) se pasara el delegado y dicho array.



Eventos

. Un evento es el modo que tiene una clase dada de proporcionar notificaciones a sus clientes cuando ocurre algo en particular dentro del objeto.

. El uso más habitual para los eventos lo vemos en las interfaces graficas (evento Click de un botón, evento Load de un Form, etc).

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

. Un evento es un mensaje enviado por un objeto para indicar que se ha producido una acción invocada programáticamente o por un usuario.

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

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

Delegados y eventos

. El objeto que produce (desencadena) el evento se denomina *emisor* del evento.

. El procedimiento que captura el evento se denomina *receptor* o *manejador* del evento.

. En cualquier caso, el emisor no sabe que objeto o método responderá a los eventos que produzca.

. Por ello, es necesario tener un componente que enlace el emisor del evento con el receptor del evento.

. El Framework .NET utiliza un tipo de delegado para trabajar como un puntero a función entre el emisor y el receptor del evento.

. En la mayoría de casos, el Framework .NET crea el delegado y se ocupa de gestionar los detalles por nosotros.

. Sin embargo, es posible crear delegados para los casos en que se desee que un evento utilice diferentes controladores de eventos en diferentes circunstancias.

Delegados

. Los eventos se declaran mediante delegados.

. Un delegado es un tipo que representa referencias a métodos con una lista de parámetros determinada y un tipo de valor devuelto.

. Un objeto delegado encapsula un método de modo que se pueda llamar de forma anónima.

. Cuando ocurre el evento, se llama a los delegados que proporcionan los clientes para el evento.

. Los delegados son como los punteros de función C++, pero tienen seguridad de tipos.

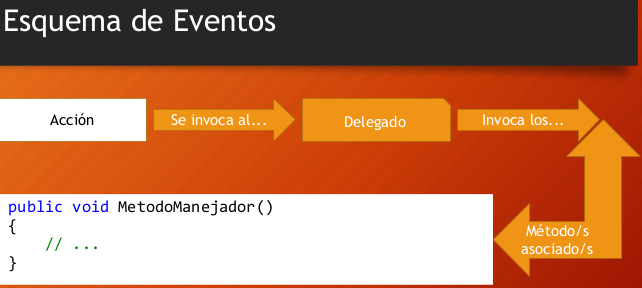
. Los delegados permiten pasar los métodos como parámetros.

. Los delegados pueden encadenarse entre sí; por ejemplo, se puede llamar a varios métodos en un solo evento.

. Un evento puede tener múltiples manejadores y viceversa.

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

public event MiDelegado ElEvento;

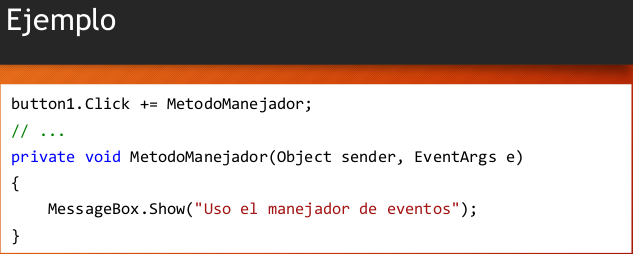


Manejadores

. Para asociar un evento a un manejador de eventos en tiempo de ejecución, hay que ‘agregarlo’ al evento del emisor.

objEmisor.Evento += MetodoManejador;

. La instrucción += agrega a la lista de invocación del evento del ‘emisor’, el nuevo manejador.



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

objEmisor.Evento -= MetodoManejador;

. La instrucción -= quita a la lista de invocación del evento del ‘emisor’, el manejador.

Eventos vs Delegados

. La suscripción (x += y) y de suscripción (x -= y) también son validas para objetos del tipo de los Delegados, así como el Invoke, BeginInvoke, etc. ¿Entonces, porque utilizamos eventos?

. El evento es al delegado lo que una propiedad a un atributo: sirve para encapsular.

. Un obejto del tipo de un delegado (MiDelgado d = new MiDelgado()) es reinstanciable, por ende se puede desasociar a todos sus manejadores en una línea. En cambio el evento no.

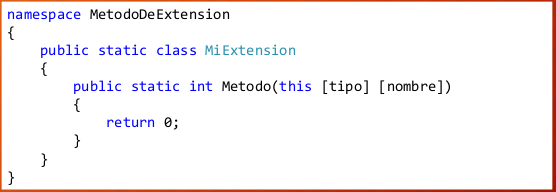
Metodos de Extension

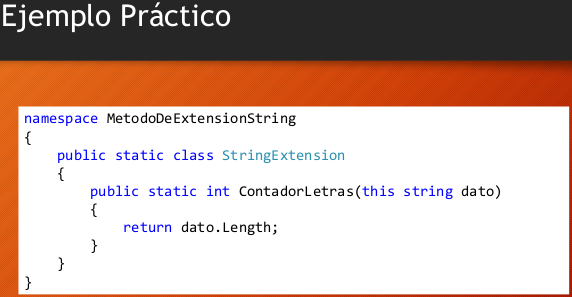
. Permiten “agregar” métodos a los tipos existentes sin crear un nuevo tipo derivado, recompilar o modificar de otra manera el tipo original.

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

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

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





Implementación

. El primer parámetro especifica en que tipo funciona el método, y el parámetro esta 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 validos tanto para clases como para interfaces.