**Resumen SP Programación 2**

**Gestión 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 está ejecutando, se inicia una excepción.
* Las excepciones detienen el flujo actual del programa, y si no se hace nada, el programa dejará de funcionar.
* Se producen por un error en el programa, por ejemplo, si se divide un número 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).
* Cada clase de excepción puede residir dentro de su propio archivo de origen y no está vinculada con las demás clases de excepción.

**Bloques try - catch**

* 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 un 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 llamó a la función de llamada y así sucesivamente hasta encontrar un bloque **catch** (o hasta llegar al final, en cuyo caso se cerrará 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 de la diapositiva, 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 más fácil de interpretar.

**Throw**

* L 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 Language Runtime (CLR), .NET Framework, las bibliotecas de otros fabricantes o el código de aplicación.

**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.

**Interfaces**

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

**Implementación explícita**

* 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.

**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.

**Serialización**

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

¿Para qué sirve?

* Para pasarlo a otro proceso, otra máquina, para grabarlo en disco, para grabarlo en una base de datos.

**Formaters**

* Controlan el formato de la serialización.
* Serialización a XML: Por defecto incluye sólo las propiedades y atributos públicos.
* Serialización Binaria: Por defecto incluyen todos los atributos y propiedades, ya sean públicas o privadas.

**XML**

* Se debe colocar un constructor por defecto en las clases a serializar.
* Sólo se guardaran los atributos o propiedades públicas.
* Si hay relación de herencia, se deberá colocar [XmlInclude(typeof(Clase))] en la clase base e indicando cada clase heredada.

**Base de datos**

* La cadena de conexión (Connection String) es donde se especificarán los datos (usuario, servidor, etc.) 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 (Procedimiento Almacenado, etc.), por ahora solo utilizaremos del tipo texto.
* El comando deberá estar asociado a una conexión, en la cual ejecutará sus acciones.

**Threads**

* 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 en conjunto conocidos como un proceso.

**Eventos**

* Un evento es el modo que tiene una clase en particular de proporcionar notificaciones a sus clientes cuando ocurre algo en particular dentro del objeto.
* El uso más habitual para los eventos lo vemos en las interfaces gráficas (evento Click de un botón, evento Load de un Form, etc.).
* Los eventos proporcionan un medio apropiado para que los objetos puedan señalizar cambios de estado que pueden resultar útiles para los clientes de ese objeto.
* Un evento es un mensaje enviado por un objeto para indicar que se ha producido una acción invocada programáticamente o por un usuario.
* Cada evento tiene un emisor que produce el evento y un receptor que lo captura.
* Utilizando eventos, los componentes de la interfaz avisan a la lógica de negocios que el usuario ha ejecutado alguna acción sobre los componentes de la misma (por ejemplo: presionar el botón del Mouse o presionar una tecla).

**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 qué 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.

**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 de 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.

**Manejadores**

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

**Métodos de extención**

* Permiten "agregar" métodos a los tipos existentes sin crear un nuevo tipo derivado, recompilar o modificar de otra manera el tipo original.
* Son una clase especial de método estático, pero se les llama como si fueran métodos de instancia en el tipo extendido.
* En el caso del código de cliente, no existe ninguna diferencia aparente entre llamar a un método de extensión y llamar a los métodos realmente definidos en un tipo.
* Se definen como métodos estáticos, pero se les llama usando la sintaxis de método de instancia.
* 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.