Programacion II

2do parcial

# HERENCIA

* Es una Relacion entre clases en la cual comparten estructura y comportamiento definido en otra clase.
* Una sublase o derivada hereda de una clase base.
* Cada clase que hereda de otra posee:
  + Los atributos de la clase base ademas de los propios.
  + Soporta todos o algunos de los metodos de la clase base.

**Proposito**: organizar mejor las clases y agruparlas en funcion de atributos y comportamientos comunes, a la vez que cada una se especializa según sus particularidades.

La herencia permite crear nuevas clases a partir de otras ya existentes en lugar de crearlas de cero.

Clase base o padre

🡪Clases derivadas o Hijas.

Tipos de Herencia:

|  |  |
| --- | --- |
| **HERENCIA SIMPLE** | **HERENCIA MULTIPLE** |
| Puede heredar SOLO de una clase base | Puede heredad de una o mas clases base.  No permitida en C# (es permitida en C++) |

**Declaración**:

[mod] class NombreClase : NombreBase

🡪Operador:Le indica que es una subclase que Procede.

* Hereda todo salvo contructores
* Los miembros publicos de base se convierten implicitamente en miembros publicos de la clase derivada.
* Solo los miembros clase base tienen acceso a los miembros privados de esta clase, aunque la clase derivada tambien los hereda.
* Una clase derivada no puede ser mas accesible que la clase base.

**Modicador Protected:** modificador de visibilidad. Es equivalente a public entre hija y base. No asi pudiendo ver desde main.

Cuando hereda un miembro protected, tambien lo es en la clase derivada.

Los metodos de una clase derivada solo tienen acceso a sus propios miembros heredados con proteccion. No pueden acceder a los miembros protegidos de la clase base a traves de referencias a esta.

Para llamar a un constructor de clase base desde derivada se utiliza la siguiente **declaracion:**

[mod] Constructor(Args) : base(arg){}

En vez de utilizar :this en el constructor utilizamos :base para mandar a base.

Esto lo toma por defecto:

EJEMPLO. Constructor(){} es igual a Constructor():base(){}

El compilador generará un mensaje de error si el contructor indicado no coincide con ningun constructor de la clase base.

**Modificador Sealed:** No se puede heredar de esa clase. (Clase sellada) EJ: System.String

**Declaración:** [mod] sealed class ClaseNombre

# ABSTRACTAS

* Son un tipo de clase
* Además de miembros de clase normales, poseen miembros de clase abstractos (son métodos y propiedades que se declara sin implementación).
* Todas las clases que derivan de una abstracta tienen que implementar esos métodos y propiedades abstractas.
* **Nunca** pueden ser instanciadas. (No se puede crear una instancia de una clase abstracta). En este sentido son como las interfaces.
* Obliga la clase base a que sus derivadas tengan un método y/o propiedad.
* Las clases abstractas se sitúan en la cima de la jerarquía de clases.
* Establecen la estructura y significado del código.
* Facilitan la creación de marcos de trabajo.

**Declaración:** [mod] abstract class NombreClase

* Solo se puede poner un miembro abstracto en una clase abstracta.

## Métodos Abstractos y utilización del Virtual

**Declaración:**

[mod] abstract tipoRetorno NombreMetodo(Arg) ;

* Siempre van en base. Obliga a que las derivadas tengan ese método. Va sin implementación, puede tomar argumentos y se coloca un ; .
* Solo las clases abstractas pueden declarar métodos abstractos.

UN METODO ABSTRACTO POR DEFINICION ES VIRTUAL PERO NO SE PUEDE DECLARARLO EN FORMA EXPLICITA

### Override y Override (clase abstracta)

* Se coloca en las clases derivadas de una clase abstracta (en el caso de abstractos).
* Sobrecarga el método en base.

### Virtual

* Se coloca en método base.
* Un método declarado como VIRTUAL indica que puede ser sobrescrito parcial o totalmente.
* Junto con OVERRIDE agrega al método virtual de la base.

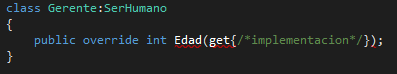
## Propiedades Abstractas

**Declaración:**

[mod] abstract tipo NombrePropiedad{get ; set ; }

* Solo clases abstractas pueden declarar propiedades abstractas.
* Los bloques no tienen implementación y finalizan en ;

Para **implementar** en clase derivada: EJEMPLO:



# POLIMORFISMO

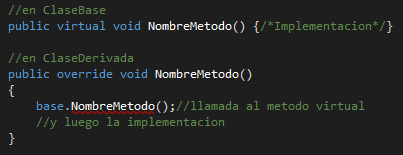
* Es la propiedad de los objetos de permitir invocar genéricamente un comportamiento (método) cuya implementación será dejada al objeto correspondiente recién en tiempo de ejecución (override/virtual)
* Es la capacidad de tratar objetos diferentes de la misma forma.
* Tiende a existir en las relaciones de herecia.
* El polimorfismo basado en herencia implica la definición de métodos en una clase base y sobrescribirlos con nuevas implementaciones en clases derivadas.
* La definición del método reside en la clase base (Se utiliza virtual).
* La implementación del método reside en la clase derivada (Se utiliza override).
* La invocación es resuelta en tiempo de ejecución.

**Declaraciones:**

[mod] virtual tiporetorno NombreMetodo (arg)

[mod] override tiporetorno NombreMetodo (arg)

EJEMPLO:



# INTERFACES

* Para crear, es como una clase. **New Item 🡪 Interface**

**Declaracion:**

[mod] interface InombreInterface { //modificadores }

* Es una clase abstracta sin atributos en donde todos sus métodos son abstractos.
* Es una clase **sin atributos**.
* Todos los métodos son abstractos y públicos (Esto es por default osea **NO** debo colocar ni el abstract ni el public).
* Se puede especificar propiedades (sin implementación)
* Las clases pueden implementar varias interfaces.
* La utilidad es similar a una herencia múltiple.
* NO se utiliza la palabra override para la implementación.

**Declaracion:**

[mod] class NombreClase : InombreInterface

🡪Implementa la interface a la clase.

EJ: interface Iguardar { bool Serializar();}

Para implementar en una clase derivada:

Class NombreClase : NombreClaseBase , Inombreinterface

# SERIALIZACIÓN

**FUNCION:**

Proceso de convertir un objeto en memoria en una secuencia lineal de bytes.

Sirve para pasarlo:

* A otro proceso.
* A otra máquina.

Sirve para grabarlo:

* En un disco.
* En una base de datos.

## Archivo de Texto

* La clase StreamWriter: escribe caracteres en archivo de texto.
* La clase StreamReader: lee desde un archivo de texto existente.
* Se encuentran en System.IO (espacio de nombres).
* El modo de uso es parecido a StringBuilder.

### STREAMWRITER

#### StreamWriter(string path)

* Creo un objeto de clase StreamWriter y le paso entre “ “ (comillas) el path de la ruta donde quiero guardar. Si el archivo existe se sobrescribirá.

#### StreamWriter(string path, bool append)

* Si append es true se agrega los datos al archivo existente, caso contrario sobrescribe el archivo.

#### StreamWriter(string path, bool append, Encondig e)

* Igual al anterior, donde se le puede especificar el tipo de codificación.

EJEMPLO (utilizando el segundo caso):

Dentro del método:

StreamWriter sw = new StreamWriter(@ “D:\Miboleto.txt”,true);

**Utilizo el @ si o si para que el path no tenga problemas con las \. También puedo utilizar \\**

sw.WriteLine(bol.ToString());

🡪muestro desde el objeto

sw.Close(); **//Siempre debo cerrarlo.**

### STREAMREADER

#### StreamReader(string path)

Misma función que Writer, pero lectura.

#### StreamReader(string path, Encondig e)

Misma función que Writer, pero lectura y además especificando el tipo de codificación.

#### Close();

Siempre debo cerrarlo.

EJEMPLO:

* Tiene la misma forma que el writer. Creo un objeto de tipo StreamReader.

Luego:

sr.ReadLine(obj.ToString()); //**lee todo el stream y lo retorna en string.**

o

sr.ReadToEnd(obj.ToString()); **//lee todo el stream y lo retorna como una cadena de caracteres.**

## Serialización XML

* Solo serializa los atributos públicos y los valores de propiedad de un objeto en una secuencia XML
* No convierte métodos, indexadores, atributos privados ni propiedades solo lectura (salvo colecciones solo lectura).
* **Clase central 🡪 XMLSERIALIZER**

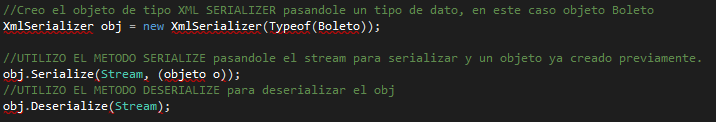
**🡪SERIALIZE**

**🡪DESEARILIZE**

* Una clase debe tener un constructor por defecto para que XmlSerializer pueda serializarla.
* **Solo serializa atributos y propiedades públicos.**
* **NO** se puede serializar métodos.

Modo de uso:

EJEMPLO:



### XMLTEXTWRITER

Modo de uso:

XmlTextWriter write = new XmlTextWriter(string path, Encondig e)

🡪String con la ruta 🡪codificación

### XMLTEXTREADER

Modo de uso:

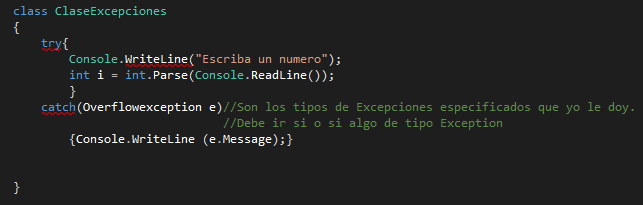
XmlTextReader read = new XmlTextReader(string path)

🡪le paso la ruta del URL

# GESTION DE ERRORES

* Permite interceptar errores en tiempo de ejecución.
* Errores derivan de la clase EXCEPTION.
* Cada clase de excepción es descriptiva y representa un error en concreto de forma clara y evidente. Conteniendo así, información específica.

EJEMPLO:



## Try/Catch

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

Mecanismo:

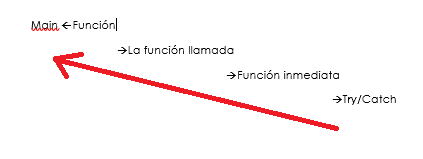
Si se produce la excepción detiene la ejecución y empieza a buscar un bloque **catch** que puede capturar la excepción pendiente (basándose en su tipo).

Si no la encuentra en su función, desenreda la pila de llamadas en busca de la función de llamada.

Si tampoco la encuentra ahí, busca la función que llamo a la función de la llamada y así sucesivamente hasta encontrar un catch (o hasta llegar al main y ahí se cerrará el programa).

RESUMEN:

Donde “🡪” significa que desenreda pila llamadas.

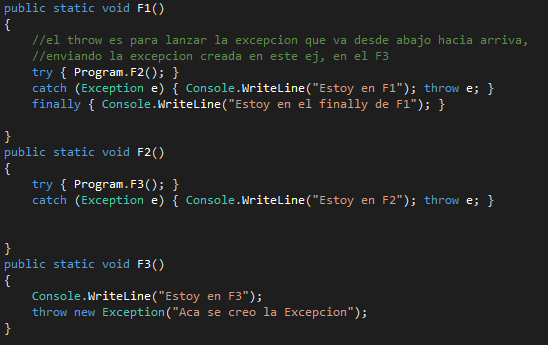


* Su uso hace que las instrucciones para tratamiento de errores no se mezclen con las instrucciones lógicas básicas, por lo que el programa es más fácil de interpretar.
* Se pueden anidar **bloques catch**. El último debe ser el genérico Exception (catch general).

### Instrucción THROW

* Es para lanzar excepciones

EJEMPLO:



### Bloque FINALLY

* Ejecuta **SI O SI**.
* Útil para liberar recursos ( o también se puede utilizar **Using**).