Lenguaje C# (Prog\_Lab2)

1. Conceptos Basicos

1.1. Tipos de Datos

Reference Types: Estos tipos son representados por una referencia a la ubicación en memoria del valor actual del objeto, similar a un puntero en C. Si asignan un Reference Type a una variable y luego la pasan a una función, cualquier cambio en el objeto se verá reflejado. NO se genera una copia.

Value Types: Estos tipos son representados por sus valores. Si asignan un Value Type a una variable se copia el valor.

Variables Escalares: Las variables escalares son constantes o variables que contienen un dato atómico y unidimensional.

Variables No Escalares: Las variables no escalares son array (vector), lista y objeto, que pueden tener almacenado en su estructura más de un valor.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Categoría** | **Clase** | **Descripción** | **C# Alias** | **Rango de valores** |
| **Enteros** | **Byte** | **Un entero sin signo (8-bit)** | **byte** | **0 a 255** |
|  | **SByte** | **Un entero con signo (8-bit)** | **sbyte** | **-128 a 127** |
|  | **Int16** | **Un entero con signo (16-bit)** | **short** | **-32768-32767/0 -65535** |
|  | **Int32** | **Un entero con signo (32-bit)** | **int** | **-2147483648- 2147483647/ 0-4294967295** |
|  | **Int64** | **Un entero con signo (64-bit)** | **long** |  |
| **Punto Flotante** | **Single** | **Un número de punto flotante de simple precisión (32-bit)** | **float** | **F** |
|  | **Double** | **Un número de punto flotante de doble precisión (64-bit)** | **double** | **D** |
|  | **Decimal** | **Un número decimal de 96-bit (con .)** | **decimal** | **M** |
| **Lógicos** | **Boolean** | **Un valor booleano (true o false)** | **bool** |  |
| **Otros** | **Char** | **Un caracter Unicode (16-bit)** | **char** |  |
|  | **Object** | **La raíz de la jerarquía de objetos** | **object** |  |
|  | **String** | **Cadena de caracteres unicode inmutable y de tamaño fijo** | **string** |  |

Valores Predeterminados

• Enteros, Punto flotante : 0 (cero)

• Lógicos: False

• Referencias: Null

Método: default(tipo)

Conversiones Básicas

• Implícitas: no interviene el programador. De un tipo menor a uno mayor.

• Explícitas: interviene el programador, ya que puede haber perdida de datos. Casteo

1.2. Formato de salida de Texto

{ N [, M ][: Formato ] }

• N: número del parámetro, empezando por cero. No es necesario que estén en orden.

• M: ancho usado para mostrar el parámetro, el cual se rellenará con espacios. Si M es negativo> justificado a la izquierda, positivo> justificado a la derecha. Tabulado.

• Formato: cadena que indicará un formato extra a usar con ese parámetro.

C (currency), D (decimal), E (científico), Fn (float+decimales), G (general), Nn(numero+decimales), X (hexa), P (%)

0 Se reemplazan los 0 por los dígitos indicado, si no hay se imprime 0

# Se reemplazan los # por los dígitos indicado, si no se imprime nada

. Separador decimal

, Separador miles

% multiplica por 100 + %

%. Multiplica por mil con separador de miles

/ el próx carácter es un literal

‘string’ el string se imprime sin modificación

; Separa secciones con distinto formato

dd/MM/yyyy fecha

Secuencias de escape:

@”” > todo el “” se interpreta como string

“” “” > p/imprimir comillas se doblan

1.3. Operadores aritméticos, lógicos + Sentencias condicionales e iterativas

Igual que en C salvo por foreach.

La sentencia foreach permite recorrer arreglos y colecciones

clase[] objeto = new clase[N];

foreach (clase elemento in objeto)

{ //auxNombre es un elemento de nombres. }

Switch:

case '1': goto case '0';

en C se puede dejar que un caso sea manejado por el siguiente, lo que se consigue si no se usa "break", mientras que C# siempre obliga a usar "break" o "goto" al final de cada cada caso, con la única excepción de que un caso no haga absolutamente nada que no sea dejar pasar el control al siguiente caso, y en ese caso se puede dejar totalmente vacío.

En el lenguaje C, que es más antiguo, sólo se podía usar "switch" para comprobar valores de variables "simples" (numéricas y caracteres); en C#, que es un lenguaje más evolucionado, se puede usar también para comprobar valores de cadenas de texto ("strings").

No necesito poner las firmas de las funciones antes de usarlas

1.4. Tipo por referencia

Convierten al parámetro formal en un alias para el argumento, que tiene que ser una variable. La palabra reservada se indica tanto en el argumento como en el param.

**in**: no permite que el método llamado modifique el valor del argumento. Tiene que estar inicializado. No es necesario en el llamado, solo en la declaración del método.

**ref**: la variable debe ser inicializada antes de pasarla. No es obligatorio modificar el valor en el método. Parámetro por referencia.

**out**: la variable no necesita estar inicializada antes de pasarla como argumento. El método llamado debe asignarle un valor antes del return. Parámetro de salida.

2. POO

2.1. Solución > Proyectos > NameSpaces > Clases > atributos y métodos > objetos

2.1.1. **Clase**:

[*modificador*] class *Identificador* { propiedades y métodos }

Identificador: sustantivos, UpperCamelCase

Modificadores: accesibilidad que tendrán sobre ella otras clases

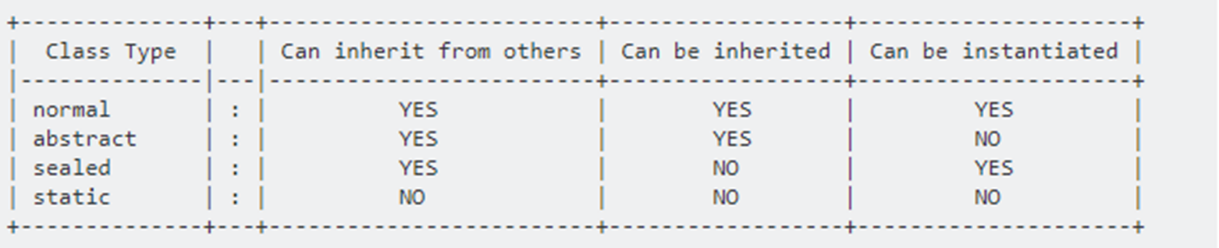
-abstract: la clase no podrá instanciarse. Superclases, estructura base para otras clases, base de jerarquía de clases

-sealed: no heredable (si puede heredar)

-static: No hereda, no heredable, no instanciable

-internal (\*): accesible en todo el proyecto

-public (\*): acc desde cualquier proyecto

-private (\*): accesor por defecto

2.1.2. **Atributos**:

[*modificador*] *tipo* *identificador*;

Identificador: lowerCamelCase. Si es privado \_

Modificador: accedibilidad

-private (\*) miembros de la misma clase. Accesor por defecto

-protected miembros de la misma clase y clases derivadas o hijas.

-internal miembros del mismo proyecto.

-internal protected miembros del mismo proyecto o clases derivadas.

-public Cualquier miembro. Accesibilidad abierta.

-static Indica que es una propiedad de clase.

2.1.3. **Métodos**:

[*modificador*] *retorno* *Identificador* ( [param] ) { Sentencias }

Identificador: verbos, UpperCamelCase

Modificador:

-extern Firma del método (para métodos externos).

-internal (\*) Accesible desde el mismo proyecto.

-public (\*) Accesible desde cualquier proyecto.

-private (\*) Sólo accesible desde la clase.

-protected (\*) Sólo accesible desde la clase o derivadas.

-abstract Sólo la firma del método, sin implementar. Solo en clases abstractas.

-virtual Permite definir métodos, con su implementación, que podrán ser sobrescritos en clases derivadas.

-override Reemplaza la implementación del mismo método declarado como virtual en una clase padre.

-static Indica que es un método de clase.

-sealed override Las clases derivadas no pueden reemplazar el método virtual/abstract de una clase superior

2.1.3.1. PARÁMETROS OPCIONALES

Por defecto, los parámetros de un método son requeridos. Se puede asignar valores por defecto a los parámetros de un método, haciéndolos opcionales. Cuando se llama al método, si ningún argumento fue suministrado se utilizará el valor por defecto.

método (tipo paramA, tipo paramB=valor) { }

2.1.3.2. Métodos de extensión

Permiten "agregar" métodos a clases o interfaces existentes sin crear un nuevo tipo derivado, recompilar o modificar de otra manera el tipo original.

Se definen como métodos estáticos, 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.

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.

public static class ClaseExtension

{ public static retorno Metodo(this claseAExtender [param])

{ }

}

2.1.3.3. METODOS y PROP ESTATICOS (DE CLASE)

Atributos Estáticos

Son atributos o propiedades asociadas a la clase y no a una instancia de la misma. No se necesita instanciar un objeto para acceder a estos atributos. No pueden acceder a los atributos no-estáticos, los cuales son específicos de cada instancia. Se declaran utilizando la palabra reservada static. Se llaman utilizando el nombre de la clase + punto + nombre del atributo o propiedad. Son constantes.

Métodos Estáticos

Son operaciones asociadas a la clase y no a una instancia de la misma. Son lo más parecido a las funciones de los lenguajes estructurados. Se utilizan para procesar datos de entrada y retornar un resultado sin necesidad de acceder al estado / atributos de un objeto concreto. No se necesita instanciar un objeto para llamar a estos métodos. No pueden acceder a los atributos no-estáticos, los cuales son específicos de cada instancia. Se declaran utilizando la palabra reservada static. Se llaman utilizando el nombre de la clase + punto + nombre del método. Solo se pueden llamar desde otros métodos estáticos.

2.1.4. **Constructor:**

modificador Clase (param) { this.propiedad= valorInicial; }

Para inicializar las propiedades al instanciar.

• Constructores por Defecto: Inicializa todos los campos con el valor predeterminado según su tipo.

• Constructores de instancia: inicializan objetos (atributos NO estáticos).

• Constructores estáticos: inicializan clases (atributos estáticos). Utilizan la palabra reservada static. No pueden recibir parámetros.

modificador Clase (tipo paramA, tipo paramB) : this (paramA)

{ this.propiedad= paramB; }

Cuando tengo otro constructor que comparte uno o más parámetros, lo reutilizo. El constructor que no recibe parámetros reutiliza a aquellos que si los reciben, pasando como argumento los valores deseados.

2.1.5. **Destructor**:

~Clase() { liberar memoria y cerrar ficheros }

No se suele usar, la memoria de los objetos la libera el Garbage Collector cuando se pierde la referencia.

2.1.6. **NameSpace**:

namespace *Identificador* { Miembros }

Es una agrupación lógica de clases y otros elementos. Toda clase está dentro de un NameSpace. Proporcionan un marco de trabajo jerárquico sobre el cuál se construye y organiza todo el código. Su función principal es la organización del código para reducir los conflictos entre nombres. Esto hace posible utilizar en un mismo programa componentes de distinta procedencia.

using nameSpace

2.1.7. **Directivas**:

Permiten a un programa identificar los NameSpaces que se usarán en el mismo. Permiten el uso de los miembros de un namespace.

* Using: Permite la especificación de una llamada a un método sin el uso obligatorio de un nombre completamente cualificado.
* Alias: Permite utilizar un nombre distinto para un Namespace. Generalmente se utiliza para abreviar nombres largos.

2.1.8. **Objetos**

*Clase* *identificador* = New *Clase* (*param*);

Instancia de clase. Es un tipo por referencia.

new: asigna memoria.

*Clase* (*param*): constructor para inicializar un objeto en esa memoria

2.1.9. **Interface**:

*modif* interface *identificador* { *retorno* *método* (*claseAbstracta* *param*) }

2.1.10. **Enumerados**

Son un conjunto propio de constantes con nombre. Estos tipos de datos permiten declarar un conjunto de nombres u otros valores literales que definen todos los valores posibles que se pueden asignar a una variable. Por dentro, estas constantes están asociadas con el tipo de dato int.

Normalmente es mejor definir un enum directamente dentro de un espacio de nombres para que todas las clases del espacio de nombres puedan acceder a él con igual comodidad. Sin embargo, un enum también se puede anidar dentro de una clase o struct.

Las enumeraciones, al igual que el resto de las constantes, son propiedades estáticas, de la clase.

modif **enum** identificador { nombreA, nombreN } si no se indican los valores, por defecto arranca en 0 y aumenta en 1

modif **enum** identificador { nombreA, nombreB=valor B, nombreN } puedo dar valor a alguno y los siguientes van en aumento

clase.identificador.nombreA return valorA

foreach (A a in (A[]) Enum.GetValues(typeof(A))) {} > para iterar enumerado

2.1.11. **Indexadores**

Permiten a la instancia de una clase ser indexada tal cómo un array/lista. No es necesario indexar con enteros.

private tipo[] atributo = new tipo[N]; // Declaro un array

public tipo **this**[int i] // Defino el indexador

{

**get** { **return** atributo [i]; } (this.atributo[i])

**set** { atributo[i] = **value**; } (this.atributo[i])

}

En el main: obj [ i ] = valor; > el valor se va a almacenar en el índice i del atributo del objeto. El índice que indico en el objeto va a ser en realidad el índice del atributo (array, lista, etc)

2.2. Pilares

**2.2.1. Encapsulamiento**:

Ocultamiento del estado, es decir, de los datos miembro de un objeto de manera que solo se pueda cambiar mediante las operaciones definidas para ese objeto. Se encarga de mantener ocultos los procesos internos.

2.2.1.1. Propiedades:

Miembro que proporciona un mecanismo flexible para leer, escribir o calcular el valor de un campo (privado). Métodos especiales denominados descriptores de acceso (gettes+setters ) / refactorizar

modif tipoDato nombreProp

{ **get** { **return** this.atributo; }

**set** { **this**.atributo = **value**; }

}

En el Main se usan como atributos en lugar de métodos: obj.Prop = valor; > set

var = obj.Prop; > get

En C:

public void SetProp (tipo valor) { prop = valor; }

public tipoProp GetProp ( ) { return this.prop; }

Atajos:

prop + tab tab > para crear propiedad + accesores

ctrl+r+e > set+get / refactorizar

**2.2.2. Herencia**:

*modif* class *ClaseDerivada* **:** *ClaseBase* { prop+métodos nuevos}

Definir una clase a partir de otra. La clase hija/derivada hereda las propiedades y métodos (algunos o todos) de la clase padre/base, todo excepto los constructores y finalizadores. Jerarquía de clases. Una clase derivada no puede ser más accesible que su clase base. Los miembros private no son visibles en las clases derivadas, pero si se heredan.

Tipos:

Herencia Simple: Una clase derivada puede heredar sólo de una clase base (los lenguajes .NET soportan este tipo de herencia)

Herencia Múltiple: Una clase derivada puede heredar de una o más clases base (C++ soporta este tipo de herencia).

2.2.2.1. Constructores: Base

modif *ClaseDerivada*(param) **:** **base** (param) { sentencias adicionales }

Para llamar a un constructor de la clase base desde un constructor de la clase derivada se usa la palabra reservada base. Si la clase derivada no hace una llamada explícita a un constructor de la clase base, el compilador de C# usará implícitamente un constructor de la forma :base() (constructor sin parámetros de la clase base). Cuando instancio un obj de una clase derivada, antes de llamar a su constructor se llama a los constr de las clases bases, de la más general a la más específica.

2.2.2.2. New: redefinir métodos

Se puede redefinir algo que ya existe en la clase base con la palabra reservada new.

modif **new** retorno MetodoBase () {}

2.2.2.3. Base: ampliar métodos

No reemplaza al método, lo amplía

modif **new** retorno MetodoBase()

{ **base**. MetodoBase(); > llamo al método de la clase base

nuevas instancias que se agregan al método anterior

}

**2.2.3. Abstracción**:

**Clases abstractas**:

El modificador **abstract** se utiliza para indicar que una clase está incompleta y que sólo se va a utilizar como una clase base. No puede instanciarse. Las clases abstractas se sitúan en la cima de la jerarquía de clases.

Aunque es posible tener variables y valores cuyos tipos en tiempo de compilación sean abstractos, tales variables y valores serán null o contendrán referencias a instancias de clases no abstractas derivadas de los tipos abstractos. (ClaseAbs var = new ClaseDerivada(); )

Se permite que una clase abstracta contenga miembros abstractos, aunque no es necesario.

Cuando una clase no abstracta se deriva de una clase abstracta, la clase no abstracta debe incluir implementaciones reales de todos los miembros abstractos heredados; por lo tanto, reemplaza a estos miembros abstractos con la palabra **override**. Una clase derivada de la clase abstracta sí se puede instanciar.

No se puede sellar una clase abstracta.

Las declaraciones de **métodos abstractos** sólo se permiten en clases abstractas. Aunque, por definición formal, un método abstracto es también implícitamente un método virtual, no puede tener el modificador virtual. Una declaración de método abstracto introduce un nuevo método virtual, pero no proporciona una implementación del método, solo la firma.

Por esto es necesario que las clases derivadas no abstractas proporcionen su propia implementación mediante el reemplazo del método. Son métodos y propiedades que se declaran sin implementación.

modif **abstract** class ClaseAbstracta

{ modif **abstract** retorno Metodo (param);

}

modif class ClaseDerivada : ClaseAbstracta

{ modif **override** retorno Metodo (param){ implementacion }

}

Modif abstract tipoDato nombreProp { get; set; } > propiedad abstracta

**2.2.4. Polimorfismo**:

Sobrecarga: funciones que tienen el mismo nombre, pero distintos parámetros y quizá se comporten de la misma forma.

Polimorfismo: funciones que se llaman igual, reciben el mismo número de parámetros, pero se aplican a objetos distintos.

objA.Metodo /= objB.Metodo

Clases cuyas propiedades son otra clase.

El polimorfismo basado en herencia implica la definición de métodos y propiedades en una clase base y sobrescribirlos con nuevas implementaciones en clases derivadas. Implica la definición de métodos y/o propiedades en una clase base, mediante el uso de la palabra reservada virtual y sobrescribirlos con nuevas implementaciones en clases derivadas con la palabra reservada override.

Cuando creo un método que recibe una clase abstracta que tmb lo puedo usar con las clases derivadas.

2.2.4.1. Virtual + override

-virtual Permite definir métodos, con su implementación, que podrán ser sobrescritos en clases derivadas.

-override Reemplaza o amplía en una clase derivada la implementación del mismo **método, propiedad, indexador o evento** declarado como **virtual, abstract u override** en una clase padre. El método base reemplazado debe tener la misma firma que el método override (modif de acceso, tipo y nombre).

El método invalidado por una declaración override se conoce como método base invalidado. No se puede reemplazar un método estático o no virtual.

No se pueden usar los modificadores new, static o virtual para modificar un método override.

Determinante de la implementación del método real a invocar:

método virtual> el tipo en tiempo de ejecución de la instancia para la que tiene lugar la invocación (abstract/virtual+override)

método no virtual> tipo en tiempo de compilación de la instancia (new)

claseBase a =new claseDerivada() > va a llamar a los elementos no virtuales de claseBase y a los virtuales de claseDerivada

modif class ClaseBase

{ modif **virtual** retorno Metodo (param){ }

}

modif class ClaseDerivada : ClaseBase

{ modif **override** retorno Metodo (param){ }

}

3. SOBRECARGA

3.1. Métodos y Constructores

Dos o más métodos en una clase pueden compartir el mismo nombre. Se sobrecargan cambiando el número, el tipo, el orden de los parámetros, modificador de parámetro (out o ref). Los constructores se sobrecargan para instanciar objetos de diferentes formas.

En los métodos reutilizo los métodos con más parámetros dentro de los que tienen menos. En los constructores los que tienen más parámetros utilizan a los que tienen menos con el operador this.

3.2. Operadores

[*acceso*] static *TipoRetorno* operator *Operador* (tipo paramA, tipo paramB) { }

*varTipoRetorno* = paramA operador paramB;

Consiste en modificar su comportamiento cuando este se utiliza con una determinada clase. El modificador de acceso no podrá ser de un ámbito mayor que el de la clase.

Operadores Sobrecargables:

+, -, !, ~, ++, - -, true, false > Unarios

+, -, \*, /, %, &, |, ^, <<, >> > Binarios

==, !=, <, >, <=, >= > Comparación, se deben sobrecargar en pares

No Sobrecargables:

&&, || > Condicionales Lógicos.

[] > Indexador de Array. Se pueden definir indexadores.

() > Casting. Se pueden definir nuevos operadores de conversión.

+=, -=, \*=, /=, %=, &=, |=, ^=, <<=, >>= > Asignación. El operador +=, por ejemplo, es evaluado usando el operador +, el cual puede ser sobrecargado.

=, ., ?:, ->, new, is, sizeof, typeof > Estos operadores no se pueden sobrecargar.

3.3. Conversión / casting

Implicitos

[acceso] static implicit operator *nombreTipo*(*tipo* *param*) { }

Explícitos

[acceso] static explicit operator *nombreTipo*(*tipo* *param*) { }

4. FORMS

Programación orientada a eventos. Estilo de programación, no paradigma.

Ante una acción se invoca un método.

Partial Class: permite separar el código de una clase en dos archivos fuentes diferentes. El diseñador de formularios utiliza esta técnica para escribir en un archivo aparte todo el código que él mismo genera.

5. Variables No Escalares

5.1. Arrays

Un array puede ser unidimensional, multidimensional o anidado (jagged).

El valor por defecto es el del tipo de dato.

Los Arrays son reference type, heredan de la clase abstracta System.Array. Implementan la interfaz IEnumerable por lo tanto se pueden iterar usando foreach.

[acceso] tipo[] identificador = new tipo[TAMAÑO]; > unidimensional

tipo[] identificador = {valor1, valorN}; > declarar e inicializar

tipo identificador [A][B]; > declara A arrays de B elementos

[acceso] tipo[ , ] identificador = new tipo[FILAS, COLUMNAS]; > multidimensional (matriz)

tipo[,] identificador = {{valor1}, {valorN}}; > declarar e inicializar

tipo[ ][ ] identificador = new tipo[A][ ]; > anidado: array de arrays, puede tener elementos de distinto tamaño

identificador [A] = new tipo [B]; > p/c/ A indico el tamaño de B

5.1.1. Array/matriz de objetos

clase[ ] identificador = new clase [N]; > reservo memoria para el array

for(byte i=0; i<N; i++) { identificador [i] = new clase (); } > reservo memoria para el objeto

Puedo declarar un array de clase base y objetos de clases derivadas.

5.2. Estructuras o registros

Agrupación de datos, llamados campos, no necesariamente del mismo tipo. Se accede a los campos con ‘.’

struct

tipo identificador; > p/declarar una variable del tipo de la estructura

6. Colecciones

Una colección es una clase, de modo que antes de poder agregar elementos a una nueva colección, debe declararla.

6.1. Genéricas

TipoColección<tipoElementos> identificador = new TipoColección < tipoElementos >();

Clases en el NameSpace System.Collections.Generic

Es útil cuando todos los elementos de la colección tienen el mismo tipo de datos. Permiten agregar sólo el tipo de datos deseado.

Dictionary: Representa una colección de pares de clave y valor que se organizan por claves. El tipo de elemento es KeyValuePair<TKey,TValue>

List: Representa una lista de objetos que pueden ser obtenidos mediante un índice. Proporciona métodos para buscar, ordenar y modificar listas.

Queue (FIFO): Representa una colección de objetos con el orden primero en entrar, primero en salir.

Stack (FILO): Representa una colección de objetos con el orden último en entrar, primero en salir.

SortedList: Representa una colección de pares de clave y valor que se ordenan por claves según la implementación de la interfaz IComparer<T> asociada.

6.2. No genéricas

TipoColección identificador = new TipoColección();

Incluidas en el NameSpace System.Collections.

No almacenan los elementos como objetos de un tipo específico, sino como objetos de tipo Object.

Para acceder a los elementos se debe hacer un casteo, excepto para aquellos del primer tipo agregado.

ArrayList: Representa una matriz de objetos cuyo tamaño aumenta dinámicamente según sea necesario.

Hashtable: Representa una colección de pares de clave y valor que se organizan por código hash de la clave.

Queue: Representa una colección de objetos con el orden primero en entrar, primero en salir (FIFO).

Stack: Representa una colección de objetos con el orden último en entrar, primero en salir (LIFO).

6.3. Concurrentes

System.Collections.Concurrent

Proporcionan operaciones eficaces y seguras para subprocesos con el fin de obtener acceso a los elementos de colección desde varios subprocesos (hilos).

Deben utilizarse en lugar de sus equivalentes en los espacios de nombres System.Collections.Generic y System.Collections cuando varios subprocesos tienen acceso a la colección simultáneamente.

BlockingCollection<T>on BlockingCollection<T>

ConcurrentDictionary<TKey, TValue>on BlockingCollection<T>

ConcurrentDictionary<TKey, TValue>

ConcurrentQueue<T>

ConcurrentStack<T>.

Casi todas las estructuras dinámicas contienen el método GetEnumerator que devuelve un enumerador para recorrer todos los elementos.

2do parcial:

7. Excepciones

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

Cuando se produce un error se lanza una excepción. Las excepciones detienen el flujo actual del programa, y si no se hace nada, el programa dejará de funcionar. El programador debe habilitar su programa para que resuelva estos problemas sin bloquearse. Estas excepciones se atrapan y se administran.

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

clases de excepciones:

1. Exception

1.1. SystemException: run-time error that cannot be handled programmatically in a meaningful way, not handled by using exception handling.

1.2. ApplicationException

PathTooLongException

IOException

FileNotFoundException

NullReferenceException

**throw new** claseException(mensaje,InnerException);

Cuando hay un error. 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. Una exc no manejada detiene el programa.

**try** **{** partes del código que podrían lanzar excepciones **}**

**catch (Exception**(o clases derivadas) identificador**)**

**{** msj de error y/o acciones correctoras **}**

**finally**

**{** conjunto de instrucciones que es necesario ejecutar sea cual sea el flujo de control **}**

identificador: opcional, es una variable local de sólo lectura en el ámbito del bloque 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. Es posible que haya muchos bloques catch y que cada uno de ellos capture un tipo específico de excepción.

Un bloque catch general (Exception), puede capturar cualquier excepción independientemente de su clase y se utiliza con frecuencia para capturar cualquier posible excepción que se pudiera producir por la falta de un controlador adecuado. Debe ser el último bloque catch en el programa. Un bloque try no puede tener más que un bloque catch general.

Seleccionar el código que puede lanzar un error> click der > Snippet > Surround with (o Ctrl+ks) > try

Crear nuevos tipos de excepciones: crear nueva clase que herede de Exception, con nuevos atributos, propiedades y métodos, usando el constructor :base(msj, innerExc)

catch(Exception e) { throw; } != catch(Exception e) { throw e; } > the full call stack is not preserved

8. TEST UNITARIOS

A test fails if it times out, throws an unexpected exception, or contains an assert statement that produces a Failed result.

Nuevo proyecto > Test > Test Unitario o seleccionar método, clase, namespace + rightclick + créate unit tests

Ejecución: Pruebas > Ventana > Explorador de Pruebas > Click derecho sobre la prueba deseada > Ejecutar

Patrón AAA:

Arrange: inicializa objetos y establece el valor de los datos que se pasa al método en pruebas.

Act: invoca al método en pruebas con los parámetros organizados.

Assert: comprueba si la acción del método en pruebas se comporta de la forma prevista. Se puede tener más de un método assert en el mismo test.

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.

**AssertFailedException** exception is thrown whenever a test fails.

Para detener el método > **return;**

Para testear que no haya exc: try{ método } catch(Exc) { Assert.Fail(); }

namespace A

{ **[TestClass]**

class B

{

**[TestMethod]**

public void nombreTest()

{

// arrange

var expected =valor;

// act

var actual = metodoTesteado();

// assert

Assert.AreEqual(expected, actual);

}

}

}

[TestMethod]

[ExpectedException(typeof(claseException))] // [Timeout(TestTimeout.Infinite)]

public void nombreTest ()

{

// arrange

// act

// assert es manejado en el ExpectedException

}

[Theory]

[InlineData(x,y)] > le paso los argumentos

public void nombreTest(x,y)

Unit test traits: the traits are set by adding metadata to the unit test directly. This enables you to more easily include or exclude these specific tests as part of a continuous deployment pipeline.

[TestMethod()]

[Owner(“ ”)]

[TestCategory(“ ”)]

[Priority(n)]

Los atributos [TestClass] y [TestMethod] son para que los reconozca el test explorer. Se puede tener otro métodos sin el atributo que sean llamados por el que si lo tiene.

Requerimientos de los métodos: sin parámetros, retorno void, atributo [TestMethod].

9. GENERICS

Introducen el concepto de **parámetros de tipo**, indicados entre **<>**, lo que le permite diseñar clases y métodos que aplazan la especificación de uno o varios tipos hasta que el código de cliente declare y cree una instancia de la clase o el método.

El parámetro de tipo T se usa en diversas ubicaciones (tipo de parámetro, tipo de retorno, tipo de atributo) donde normalmente se usaría un tipo concreto. Cada repetición de T luego se sustituye en tiempo de ejecución por el tipo indicado en la declaración de un objeto de la clase. Fuera de la misma clase, siempre que me refiera a la clase debo indicar argumento de tipo, incluso en métodos estáticos.

Puede limitar las clases genéricas para habilitar el acceso a métodos en tipos de datos determinados.

En el nombre del constructor no se incluye <T>

modif class Clase**<**T**>** **where** T: restricciónA, B

{

modif T miAtributo; //atributos y métodos que utilizan el tipo T, entre otros

}

Clase<tipo> var = new Clase<tipo> (); > indico el tipo en la instancia

|  |  |
| --- | --- |
| Restricción | Descripción |
| T : struct | El argumento de tipo debe ser un tipo de valor. Implica la restr new() |
| T : class | debe ser un tipo de referencia. |
| T : notnull | debe ser un tipo de valor o de referencia que no acepta valores NULL |
| T : unmanaged | no debe ser un tipo de referencia y no debe contener ningún miembro de tipo de referencia en ningún nivel de anidamiento. Implica la restr struct. No se puede combinar con struct o new() |
| T : new() | debe tener un constructor sin parámetros público. Cuando se usa conjuntamente con otras restricciones, la restricción new() debe especificarse en último lugar. No se puede combinar con la restr struct. |
| T : <nombreClaseBase> | debe ser o derivarse de la clase base especificada. |
| T : <nombreInterfaz> | debe ser o implementar la interfaz especificada. Pueden especificarse varias restricciones de interfaz. La interfaz de restricciones también puede ser genérica. |
| T : U | El argumento de tipo proporcionado por T debe ser o derivarse del argumento proporcionado para U. |
| T : System.Enum |  |

Métodos genéricos:

class Clase > no es necesario que la clase sea genérica

{

modif retorno Metodo<T>(T s, T t) where T : restricciones > se pueden sumar restricciones a las de la clase

{ }

}

En el llamado se puede omitir el argumento de tipo y el compilador lo deducirá basado en los argumentos de método que se pasan. La inferencia de tipos se produce en tiempo de compilación antes de que el compilador intente resolver las firmas de método sobrecargadas.

Los métodos genéricos pueden sobrecargarse con los parámetros de tipo.

10. INTERFACES

[modificadores] **interface** **I**NombreInterface { Miembros de la interface }

[modificadores] class NombreClase : [ClaseBase,] INombreInterface > implementación

{

//Implementación de los miembros de la interface, sin override, public

// retorno INombreInterface.NombreMetodo() {}

}

Miembros: propiedades, métodos, eventos, indexadores. Sin modificadores de acceso (todos públicos), sin implementación, no pueden ser estáticos. Solo la firma: retorno nombre(param);

No puede contener constantes, atributos, operadores, constructores de instancia, finalizadores o tipos.

Implementación: miembro public, no estático, misma firma. La clase o struct debe implementar todos los miembros de la interfaz implementada, si no quiero implementar > throw NotImplementedException()

Una clase puede implementar un miembro una sola vez, las clases derivadas heredan la implementación de la clase base, aunque puede modificarlo (override) si la implementación es virtual.

Pueden ampliar las propiedades e indexadores (agregando un accesor que no se indica en la interfaz), a no ser que se utilice la implementación explícita, en ese caso deben ser iguales.

Si una clase implementa dos interfaces que tienen un miembro con la misma firma, ambas van a tomar la implementación como propia a no ser que sea explícita.

Implementación explícita: no se indica visibilidad.

P/usarlo se castea (implícita o explícitamente) el objeto a Interface

((Intef)obj).metodo > expl

Interface var = new Clase(); var.metodo(); > impl

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

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

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

Una interfaz puede heredar otra interfaz, aplican las reglas de polimorfismo (new, virtual, override)

Las interfaces pueden ser argumentos de tipo en clases genéricas

11. ARCHIVOS DE TEXTO

using System.IO;

Clase StreamWriter

Clase StreamReader

Escribir:

**StreamWriter** sw = new StreamWriter (stringRuta); > Abrir archivo ubicado en una dirección de la máquina

sw.**WriteLine**(""); > escribir

sw.**Close**(); > Cerrar archivo

try{ if (!Directory.Exists(ruta)) //GetDirectoryName(Archivo.txt)

throw new DirectoryNotFoundException();

**using**( StreamWriter sw = new StreamWriter(strArchivo) ) // Al finalizar se cierra el archivo, incluso si hay una exc

{ sw.WriteLine("datos"); } return;}

Leer:

**StreamReader** sw = new StreamReader (stringRuta); > Abro el archivo ubicado en una dirección de la máquina

sr.**ReadLine**(); > Leo una línea de texto

sw.Close(); > Cierro el archivo

try{

if(!File.Exists(ruta))

throw new FileNotFoundException();

string linea;

string info = "";

**using**(**StreamReader** str = new StreamReader(ruta))

{ while ((linea = str.**ReadLine**()) != null) // Para leer un archivo completo

{ info += (linea + "\n"); } } return info; }

El código debería ir dentro de bloques try{}catch(){} finally{ s.Close() }

Ruta:

“D://carpeta//archivo.txt” > ruta absoluta “//” o @””

“..//carpeta//archivo.txt” > ruta relativa

Environment.GetFolderPath(Environment.SpecialFolder.Carpeta) > ruta de carpetas del sistema: Desktop,MyDocuments,etc

AppDomain.CurrentDomain.BaseDirectory + "Archivo" + ".txt"

14. 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.

14.1. Serialización a XML:

Serializa los **atributos y propiedades** (los valores de propiedad, get) **públicos** de un objeto en una secuencia XML.

No convierte (serializa) los métodos, indexadores, atributos privados ni propiedades de sólo lectura (salvo colecciones de sólo lectura).

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

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

El orden en que se van a guardar y leer es el orden en que aparecen las propiedades.

Clase **XmlSerializer**: crea archivos C# (.cs) y los compila en archivos .dll en el directorio especificado por la variable de entorno TEMP; la serialización se produce con esos archivos DLL.. Metodos Serialize y Deserialize

Clase **XmlTextWriter**: Provee una manera de generar archivos con contenido de datos XML.

Clase **XmlTextReader**: Provee una manera de leer archivos con contenido de datos XML.

using System.Xml;

using System.Xml.Serialization;

Serializar:

Dato origen = new Dato(); // Objeto a serializar

**XmlTextWriter** writer = new XmlTextWriter(Archivo.xml, Encoding.UTF8); // Objeto que escribirá en XML. Se indica ubicación del archivo XML y su codificación.

**XmlSerializer** ser = new XmlSerializer(**typeof**(Dato)); // Objeto que serializará. Se indica el tipo de objeto a serializar.

ser.**Serialize**(writer, origen); //Serializa el objeto p en el archivo contenido en writer.

writer.Close(); //Se cierra la conexión al archivo

try

{ if (!Directory.Exists(ruta)) //GetDirectoryName(Archivo.xml)

throw new DirectoryNotFoundException();

**using** (**XmlTextWriter** writer = new XmlTextWriter(Archivo.xml, System.Text.Encoding.UTF8))

{ **XmlSerializer** ser = new XmlSerializer(typeof(Dato));

ser.Serialize(writer, origen); } return;}

Deserializar:

Dato destino = new Dato(); //Objeto que alojará los datos contenidos en el archivo XML.

**XmlTextReader** reader = new XmlTextReader(Archivo.xml); // Objeto que leerá XML. Se indica ubicación del archivo XML.

**XmlSerializer** ser = new XmlSerializer(typeof(Dato)); // Objeto que Deserializará. Se indica el tipo de objeto a serializar.

destino = (Dato)ser.**Deserialize**(reader); //Deserializa el archivo contenido en reader, lo guarda en aux.

reader.Close(); //Se cierra el objeto reader.

try

{ if(!File.Exists(archivo)

throw new FileNotFoundException();

//Dato aux = new Dato();

**using** (**XmlTextReader** reader = new XmlTextReader(Archivo.xml))

{ **XmlSerializer** ser = new XmlSerializer(typeof(Dato));

aux = (Dato)ser.**Deserialize**(reader); } return aux; }

14.1. Serialización Binaria:

Puede serializar **atributos públicos y privados**.

se debe agregar el marcador [Serializable] a la clase, antes de la definición.

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

Clase **BinaryFormatter**: Metodos Serialize y Deserialize

Clase **FileStream**

using System.IO;

using System.Runtime.Serialization.Formatters.Binary;

[Serializable]

Serializar:

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

**FileStream** fs = new FileStream(Archivo.dat, FileMode.Create); //Objeto que escribirá en binario. Se indica ubicación del archivo binario y el modo.

**BinaryFormatter** ser = new BinaryFormatter(); //Objeto que serializará.

ser.**Serialize**(fs, origen); //Serializa el objeto origen en el archivo contenido en fs.

fs.Close(); //Se cierra el objeto fs

try{ if (!Directory.Exists(PathArchivo.dat)) //GetDirectoryName(Archivo.dat)

throw new DirectoryNotFoundException();

**using ( FileStream** fs = new FileStream(Archivo.dat, FileMode.Create)) //OpenOrCreate

**{ BinaryFormatter** ser = new BinaryFormatter();

ser.**Serialize**(fs, origen); } return; }

Deserializar:

try {

Dato dest = new Dato(); //Objeto que alojará los datos contenidos en el archivo binario.

**FileStream** fs = new FileStream(Archivo.dat, FileMode.Open); //Objeto que leerá en binario. Se indica ubicación del archivo binario y el modo.

**BinaryFormatter** ser = new BinaryFormatter(); //Objeto que Deserializará.

dest = (Dato)ser.**Deserialize**(fs); } // Deserializa el archivo contenido en fs, lo guarda en dest.

finally { fs.**Close**(); } //Se cierra el objeto fs.

try

{ if(!File.Exists(Archivo.dat) )

throw new FileNotFoundException();

//Dato aux = new Dato();

**using** (**FileStream** fs = new FileStream(Archivo.dat, FileMode.Open))

{ **BinaryFormatter** ser = new BinaryFormatter();

aux = (Dato)ser.**Deserialize**(fs); } return aux;}

15. BASE DE DATOS

Crear Base de datos:

Microsoft SQL Server Management Studio 18 > Databases > rightclick> New Database > Nombre > Ok

Tables> rightclick > New > Table

id: Null no, rightclick set primary key, identity specification > is identity: yes

connectionString: es donde se especificarán los datos (usuario, servidor, etc.) de una conexión a una fuente de datos.

@”Server=localhost\SQLEXPRESS;Database=master;Trusted\_Connection=True;”

@"Server = .\SQLEXPRESS ; Database = NombreDatabase ; Trusted\_Connection = true;"

using System.Data.SqlClient;

opción sin using:

public static class DAO

{ private static **SqlConnection** connection;

private static **SqlCommand** command;

private static string connectionString;

static DAO()

{ connectionString = @"Server = .\SQLEXPRESS ; Database = NombreDatabase ; Trusted\_Connection = true;";

connection = new SqlConnection(connectionString);

command = new SqlCommand();

command.Connection = connection; // lo puedo indicar en el ctor new SqlCommand(comando, connection);

command.CommandType = System.Data.CommandType.Text;

}

public static void ComandoSQL(datos para cada campo del registro u objeto que contenga los datos)

{ try

{ connection.**Open**(); //p/abrir la conexiòn con la base de datos

string comando = "comandoSQL nombreCampo =@nombreCampo;";

command.**Parameters.AddWithValue**("@nombreCampo", nombreCampo);

command.**CommandText** = comando;

command.**ExecuteNonQuery**();

}

finally

{ if (connection != null && connection.State == System.Data.ConnectionState.Open)

connection.**Close**(); }

}} //fin clase

opción con using: los objetos dentro tienen que implementar la interfaz IDisposable

**public static class DAO**

{ private static string connectionString=@"Server = .\SQLEXPRESS ; Database = NombreDatabase ; Trusted\_Connection = true;";

//Insert, Update y Delete:

**public static void ComandoSQL**(datos para cada campo del registro u objeto que contenga los datos)

{ using (**SqlConnection** connection = new SqlConnection(DAO.connectionString))

{ string comando = "comandoSQL nombreCampo =@nombreCampo;"; //se puede recibir por param

**SqlCommand** command = new SqlCommand(comando,connection);

command.**Parameters.AddWithValue**("@nombreCampo", nombreParam); //p armar relaciones entre un identificador y un param

connection.**Open**(); //p/abrir la conexiòn c/ la base de datos

command.**ExecuteNonQuery**();

}

}

//Select:

public static List<Clase> **ComandoSQLConsulta**() //cada registro va a ser un elemento de la lista

{ using (**SqlConnection** connection = new SqlConnection(DAO.connectionString))

{ string comando = "**SELECT** campo FROM tabla;";

**SqlCommand** command = new SqlCommand(comando, connection);

connection.**Open**();

**SqlDataReader** dataReader = command.**ExecuteReader**(); //s/doc dentro de otro using

List< Clase > lista = new List< Clase>();

while(dataReader.**Read**()) //Read() se desplaza al sig registro

{ int id = dataReader.**GetInt32**(0); //el num es el orden de columna en el comando

string? nombre = null;

if (!dataReader.**IsDBNull**(1))

nombre = dataReader. **GetString** (1);

Clase cliente = new Clase (id, nombre);

lista.**Add**(clase);

}

dataReader.Close();

return lista; }}

} //fin clase

16. THREADS

Secuencia de tareas encadenadas. Un hilo es una tarea que puede ser ejecutada al mismo tiempo que otra tarea.

using System.Threading;

Thread t = new Thread(new **ThreadStart**(método)); // Creo el hilo, con el tipo delegado ThreadStart

t.**Start**(); // Inicio el Hilo

private void Metodo(){ while(true) codigo}

Thread t = new Thread(new **ParameterizedThreadStart**(Metodo)); // Si el método tiene parametros

t.**Start**(argumento); // Inicio el Hilo, indicando el parámetro

private void Metodo(object o){ { while(true) codigo}

if(this.hilo != null && this.hilo.IsAlive) > finalizo

{ this.hilo.**Abort**(); }

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

private void Metodo(object o)

{ if (this.label.**InvokeRequired**)

{ this.label.**BeginInvoke**((MethodInvoker)delegate()

{ this.label.Text = (int)o;

}

}

else{

this.label.Text = (int)o; }

}

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 pasará el delegado y dicho array.

delegate void **Callback**(); // delegate void **Callback**( object o);

public void OtroMetodo(object param)

{ if (this.textBox.**InvokeRequired**) {

**Callback** d = new Callback(this.OtroMetodo);

object[] objs = new object[] { param };

this.**Invoke**(d, objs);

//o this.textBox.Invoke(new Callback(this.OtroMetodo), new object[] { param } ); // llamará al método dsd el hilo q lo creó } else {

this.textBox.Text = texto;

}

}

17. EVENTOS

Leer:

<https://docs.microsoft.com/en-us/dotnet/api/system.xml.serialization.xmlserializer?view=netframework-4.8>

<https://docs.microsoft.com/en-us/dotnet/api/system.xml.xmltextwriter?view=netframework-4.8>

<https://docs.microsoft.com/en-us/dotnet/api/system.xml.xmltextreader?view=netframework-4.8>

<https://docs.microsoft.com/en-us/dotnet/api/system.io.filestream?view=netframework-4.8>

<https://docs.microsoft.com/en-us/dotnet/api/system.runtime.serialization.formatters.binary.binaryformatter?view=netframework-4.8>

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Tema | PPT 2018/campus | PPT clase | Biblio | Doc oficial |
| 1. Conceptos Basicos | Ok 1 | Ok ( igual + .NET) | ok |  |
| POO/ | Ok 2 | ok | Ok |  |
| 2. Objetos | Ok 3 | igual | ok |  |
| 4.1 Sobrecarga Metodos/constr | Ok 4.1 | igual |  |  |
| 3. SOBRECARGA DE OPERADORES | Ok 4.2 | igual | ok |  |
| 4. FORMS | Ok 6.1 | Igual + ejemplos | - |  |
| 5. Arrays y | Ok arrays 6.2 | igual | ok |  |
| 7 Colécciones | 7 ok | igual | 7 ok |  |
| 8. ENCAPSULAMIENTO | Prop, index,enum ok | Igual + abstracción ok | Clase 8 ok |  |
| 9. HERENCIA | ok | igual | ok |  |
|  |  | Encapsulamiento |  |  |
| 10., ABSTRACT y VIRTUAL | ok | igual | Virtual+override |  |
| 11 SOBRECARGA DE MÉTODOS(override), POLIMORFISMO | ok | igual | ok |  |
| 15. Excepciones | ok | igual | ok | ok |
| 16. TEST UNITARIOS | ok | igual | - | Ok |
| 17. GENERICS | ok | igual | - | Faltan dos entradas |
| 18. INTERFACES | Ok | igual | - | ok |
| 19.1. ARCHIVOS DE TEXTO | Ok | igual |  | ok |
| 19.2. SERIALIZACIÓN | Ok | igual |  |  |
| 21. BASE DE DATOS | Ok | igual |  | ok |
| 22. THREADS | ok | igual | - |  |
| 23. EVENTOS |  | igual |  |  |
| 25. MÉTODOS DE EXTENSIÓN | Ok | igual | - |  |