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

2. POO

2.1. Solución > Proyectos > NameSpaces > Clases > 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

-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 un método de clase.

2.1.3. **Métodos**:

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

Identificador: verbos, UpperCamelCase

Modificador:

-abstract Sólo la firma del método, sin implementar.

-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.

-static Indica que es un método de clase.

-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.

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)

2.1.4. **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.

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*) }

2.2. Pilares

**Abstracción**:

**Herencia**:

*modif* class *Clase2* : *Clase1* { prop+métodos nuevos}

La clase hija/derivada hereda las propiedades y métodos de la clase padre/base. Jerarquía de clases.

**Encapsulamiento**:

modificadores private, gettes+setters (ctrl+r+e) / refactorizar

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

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

**Polimorfismo**:

Clases cuyas propiedades son otra clase.

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

Atajos:

prop + tab tab > para crear propiedad + accesores

ctrl+r+e > set+get

2.3. 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.

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) { }

*TipoRetorno* = 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

5. Arrays y Colécciones

6. ENCAPSULAMIENTO

7. HERENCIA

8. SOBRECARGA DE MÉTODOS, POLIMORFISMO, ABSTRACT y VIRTUAL

9. Excepciones

10. TEST UNITARIOS

11. GENERICS

12. INTERFACES

13. ARCHIVOS DE TEXTO

14. SERIALIZACIÓN

15. BASE DE DATOS

16. THREADS

17. EVENTOS

18. MÉTODOS DE EXTENSIÓN

|  |  |  |  |
| --- | --- | --- | --- |
| Tema | PPT 2018/campus | PPT clase | Biblio |
| 1. Conceptos Basicos | Ok | ok | ok |
| POO/ | ok | ok | Ok |
| 2. Objetos | ok | ok | ok |
| 3. SOBRECARGA DE OPERADORES | ok | ok | ok |
| 4. FORMS |  |  | - |
| 5. Arrays y Colécciones |  |  |  |
| 6. ENCAPSULAMIENTO |  |  |  |
| 7. HERENCIA |  |  |  |
| 8. SOBRECARGA DE MÉTODOS, POLIMORFISMO, ABSTRACT y VIRTUAL |  |  |  |
| 9. Excepciones |  |  |  |
| 10. TEST UNITARIOS |  |  |  |
| 11. GENERICS |  |  |  |
| 12. INTERFACES |  |  |  |
| 13. ARCHIVOS DE TEXTO |  |  |  |
| 14. SERIALIZACIÓN |  |  |  |
| 15. BASE DE DATOS |  |  |  |
| 16. THREADS |  |  |  |
| 17. EVENTOS |  |  |  |
| 18. MÉTODOS DE EXTENSIÓN |  |  |  |

Imprimir ppts 2018 y clase, compararlos