CODES

Capacitación Web – POO

Programación en C#

codes
CONSULTORIA
Y DESARROLLO DE SISTEMAS

CMMI

ÍNDICE

ÍNDICE

1. INTRODUCCIÓN

El objetivo de estas diapositivas es repasar las características y sintaxis del lenguaje C#.

La información de esta guía está basada en la MSDN de Microsoft, la cual puede ser accedida desde la siguiente URL:

http://msdn.microsoft.com/es-es/library/67ef8sbd.aspx

ÍNDICE

Los programas en C# pueden constar de uno o varios archivos. Cada archivo puede contener cero o varios espacios de nombres. Un espacio de nombres (namespace) puede contener tipos, clases, structs, interfaces, enumeraciones y delegados, además de otros espacios de nombres:

```
using System;
namespace MiEspacioDeNombres
{
    class MiClase
    {
    }

    namespace MiEspacioDeNombresAnidado
    {
        class OtraClase
        {
        }
    }

    class MiClasePrincipal
    {
        static void Main(string[] args)
        {
            //Acá programo la funcionalidad principal...
        }
    }
}
```

ÍNDICE

3. ESPACIO DE NOMBRES (NAMESPACE)

.NET utiliza los espacios de nombres para organizar sus múltiples clases, de la forma siguiente:

```
System.Console.WriteLine("Hola, Mundo!");
```

System es un espacio de nombres y **Console** es una clase de ese espacio de nombres. Se puede utilizar la palabra clave **using** para que no se requiera el nombre completo, como en el ejemplo siguiente:

```
using System;
//.
Console.WriteLine("Hola, Mundo!");
```

Además, se pueden declarar espacios de nombres propios que pueden ayudar a controlar el ámbito de clase y nombres de método en proyectos de programación grandes. Utilice la palabra clave ***namespace*** para declarar un espacio de nombres:

```
namespace EspacioDeNombresEjemplo
{
    class ClaseDeEjemplo
    {
        public void MetodoEjemplo()
        {
            System.Console.WriteLine("Método en el EspacioDeNombresEjemplo");
        }
    }
}
```

ÍNDICE

Las clases se declaran mediante la palabra clave class:

```
public class Cliente
{
}
```

El nivel de acceso precede a la palabra clave **class**. Como, en este caso, se utiliza **public**, cualquiera puede crear objetos a partir de esta clase. El nombre de la clase sigue a la palabra clave **class**. El resto de la definición es el cuerpo de clase, donde se definen el comportamiento y los datos.

IMPORTANTE: Recordar que una clase define un tipo de objeto. Un objeto es una entidad concreta basada en una clase y, a veces, se denomina instancia de una clase.
Los objetos se pueden crear con la palabra clave **new** seguida del nombre de la clase en la que se basará el objeto, de la manera siguiente:

```
Cliente objetoCliente = new Cliente();
```

ÍNDICE

Un campo es una variable de cualquier tipo que se declara directamente en una clase. Los campos son miembros de su tipo contenedor. Por regla general, solo debe utilizar campos para variables con accesibilidad privada o protegida. Los datos que la clase expone al código de cliente se deben proporcionar a través de métodos y propiedades.
Los campos se declaran en el bloque de clase especificando el nivel de acceso del campo, seguido por el tipo de campo y después por el nombre del mismo:

```
public class Persona
{
    // campo privado de la clase
    private int legajo;

    // Una función public puede exponer de manera segura el miembro privado
    public int GetLegajo()
    {
        return this.legajo;
    }

    // Y un método public asigna de manera segura el valor al miembro
    public void SetLegajo(int parametroLegajo)
    {
        this.legajo = parametroLegajo;
    }
}
```

Una propiedad es un miembro que ofrece un mecanismo flexible para leer, escribir o calcular el valor de un campo privado. Las propiedades pueden utilizarse como si fuesen miembros de datos públicos, aunque en realidad son métodos especiales denominados ***descriptores de acceso***. De este modo, se puede obtener acceso a los datos con facilidad, a la vez que se promueve la seguridad y flexibilidad de los métodos.
En este ejemplo, la clase PeriodoDeTiempo almacena un período de tiempo. Internamente, la clase almacena el tiempo en segundos, pero una propiedad denominada *Horas* permite a un cliente especificar el tiempo en horas. Los descriptores de acceso de la propiedad *Horas* realizan la conversión entre horas y segundos.

```
class PeriodoDeTiempo
{
    private double segundos;

    public double Horas
    {
        get { return segundos / 3600; }
        set { segundos = value * 3600; }
    }
}
```

¿Cómo utilizar las propiedades, métodos y funciones?

```
//Ejemplo de cómo utilizar el método y la función definidas para acceder
//al miembro privado legajo
Persona objetoPersona = new Persona();
objetoPersona.SetLegajo(512);
System.Console.WriteLine(objetoPersona.GetLegajo());
```

```
//Ejemplo de cómo utilizar la propiedad definida para la clase PeriodoDeTiempo
PeriodoDeTiempo objetoPeriodo = new PeriodoDeTiempo();
objetoPeriodo.Horas = 3;
System.Console.WriteLine(objetoPeriodo.Horas);
```

ÍNDICE

Un método es un bloque de código que contiene una serie de instrucciones. Los métodos se declaran en una clase mediante la especificación del nivel de acceso como public o private, modificadores opcionales como abstract o sealed, el valor devuelto, el nombre del método y cualquier parámetro de método. Todos esos elementos constituyen la firma del método.
Los parámetros del método se encierran entre paréntesis y se separan por comas. Los paréntesis vacíos indican que el método no requiere ningún parámetro. Esta clase contiene dos métodos:

```
public class CuentaBancaria {
   private double saldo;

   public void IncrementarSaldo(double monto) {
      saldo = saldo + monto;
   }

   public double ExtraerDinero(double monto) {
      if(saldo < monto) {
         System.Console.WriteLine("No hay saldo");
      }
      else {
         saldo = saldo - monto;
      }
      return saldo;
   }
}
```

ÍNDICE

7. TIPOS

C# es un lenguaje fuertemente tipado. Todas las variables y constantes tienen un tipo, al igual que toda expresión que da como resultado un valor. Cada firma de método especifica un tipo para cada parámetro de entrada y para el valor devuelto. La biblioteca de clases .NET Framework define un conjunto de tipos numéricos integrados y tipos más complejos que representan una amplia variedad de construcciones lógicas, como el sistema de archivos, conexiones de red, colecciones y matrices de objetos y fechas. Un programa típico de C# usa los tipos de la biblioteca de clases, así como tipos definidos por el usuario que modelan los conceptos específicos del dominio problemático del programa.

El compilador utiliza información de tipos para asegurarse de que todas las operaciones que se realizan en el código cumplen la seguridad de tipos. Por ejemplo, si declara una variable de tipo int, el compilador permite utilizar la variable en operaciones de suma y resta. Si intenta realizar esas mismas operaciones con una variable de tipo bool, el compilador genera un error, como se muestra en el ejemplo siguiente:

```
int a = 5;
int b = a + 2; //OK

bool test = true;

// Error. Operador '+' no puede ser aplicado para
// cálculos con tipos 'int' y 'bool'.
int c = a + test;
```

7. TIPOS (II)

C# Tipo	.Net Framework (System) type	Signed?	Bytes en Ram	Rango
sbyte	System.Sbyte	Yes	1	-128 a 127
short	System.Int16	Yes	2	-32768 a 32767
int	System.Int32	Yes	4	-2147483648 a 2147483647
long	System.Int64	Yes	8	-9223372036854775808 a 9223372036854775807
byte	System.Byte	No	1	0 a 255
ushort	System.Uint16	No	2	0 a 65535
uint	System.UInt32	No	4	0 a 4294967295
ulong	System.Uint64	No	8	0 a 18446744073709551615
float	System.Single	Yes	4	Aprox. $\pm 1.5 \times 10^{-45}$ a $\pm 3.4 \times 10^{38}$ con 7 decimales
double	System.Double	Yes	8	Aprox. $\pm 5.0 \times 10^{-324}$ a $\pm 1.7 \times 10^{308}$ con 15 o 16 decimales
decimal	System.Decimal	Yes	12	Aprox. $\pm 1.0 \times 10^{-28}$ a $\pm 7.9 \times 10^{28}$ con 28 o 29 decimales
char	System.Char	N/A	2	Cualquier caracter Unicode
bool	System.Boolean	N/A	1 / 2	true o false

El tipo ***string*** representa una secuencia de cero o más caracteres Unicode. ***string*** es un alias de String en .NET Framework.

```
//Ejemplo de declaración
string nombre;
nombre = “Juan”;

//Ejemplo de declaración y asignación
string apellido = “Perez”;

// Las cadenas se concatenan con el operador +
string profesion = “analista ” + “de sistemas”;
//La línea anterior crea un objeto con valor “analista de sistemas”

//El operador [] se puede utilizar para tener acceso de sólo lectura
//a caracteres individuales de un objeto string:
string str = "test";
char x = str[2];  // x = 's';

//Los literales de cadena pueden contener cualquier literal de carácter. Se
//incluyen las secuencias de escape. Ejemplo: se usa la secuencia de
//escape \\ para la barra diagonal inversa y \n para una nueva línea:
string a = "\\Hola\n";
Console.WriteLine(a);
```

ÍNDICE

Puede almacenar distintas variables del mismo tipo en una estructura de datos de matriz. Para declarar una matriz especifique el tipo de sus elementos:

```
// Declarar una matriz uni-dimensional
int[] array1 = new int[5];

// Declarar y setear valores del array
int[] array2 = new int[] { 1, 3, 5, 7, 9 };

// Declarar un array de dos dimensiones
int[,] multiDimensionalArray1 = new int[2, 3];

// Declarar y especificar valores al array multi-dimensional
int[,] multiDimensionalArray2 = { { 1, 2, 3 }, { 4, 5, 6 } };

// Declarar un array de arrays
int[][] jaggedArray = new int[6][];

// Setear valores del primer array en el array de arrays
jaggedArray[0] = new int[4] { 1, 2, 3, 4 };
```

Para recorrer matrices:

```
//Ejemplo con FOR
for (int i = 0; i < arr.Length; i++)
{
    System.Console.WriteLine(”Valor ” + i + ”: ” + arr[i]);
}

//Ejemplo con FOREACH
int[] numbers = { 4, 5, 6, 1, 2, 3, -2, -1, 0 };
foreach (int i in numbers)
{
    System.Console.Write("{0} ", i);
}

//Ejemplo con WHILE
int i = 0;
while(i < arr.Length)
{
   System.Console.WriteLine(“Valor ” + i + “: ” + arr[i]);
   i++;
}
```

ÍNDICE

9. INSTRUCCIONES DE SELECCIÓN

Una instrucción de selección hace que el control del programa se transfiera a un determinado punto del flujo de ejecución dependiendo de que cierta condición sea true o no.
Las siguientes palabras clave se utilizan en instrucciones de selección:

-**if-else**: Una instrucción if identifica que sentencia se tiene que ejecutar en función del valor de una expresión Boolean. En una instrucción if-else, si la condición se evalúa como true, se ejecuta la sentencia then-statement. Si condition es false, else-statement ejecuta. Dado que la condición (condition) no puede ser simultáneamente verdadera (true) y falsa (false), las sentencias then-statement y else-statement de una instrucción if-else nunca pueden ejecutarse simultáneamente.

-**switch-case-default**: La instrucción *switch* es una instrucción de control que selecciona una sección *switch* para ejecutarla desde una lista de candidatos. Una instrucción *switch* incluye una o más secciones *switch*. Cada sección *switch* contiene una o más etiquetas *case* seguidas de una o más instrucciones.

Ejemplo de if-else

```
bool condition = true;

if (condition)
{
    Console.WriteLine("The variable is set to true.");
}
else
{
    Console.WriteLine("The variable is set to false.");
}
```

Ejemplo de switch-case-default

```
int caseSwitch = 1;
switch (caseSwitch)
{
    case 1:
        Console.WriteLine("Case 1");
        break;
    case 2:
        Console.WriteLine("Case 2");
        break;
    default:
        Console.WriteLine("Default case");
        break;
}
```

ÍNDICE

10. HERENCIA

La herencia, junto con la encapsulación y el polimorfismo, es una de las tres características principales (o pilares) de la programación orientada a objetos. La herencia permite crear nuevas clases que reutilizan, extienden y modifican el comportamiento que se define en otras clases. La clase cuyos miembros se heredan se denomina clase base y la clase que hereda esos miembros se denomina clase derivada. Una clase derivada solo puede tener una clase base directa. Sin embargo, la herencia es transitiva. Si ClassC se deriva de ClassB y ClassB se deriva de ClassA, ClassC hereda los miembros declarados en ClassB y ClassA.

Métodos abstractos y virtuales
Cuando una clase base declara un método como virtual, una clase derivada puede invalidar el método con su propia implementación. Si una clase base declara un miembro como abstracto, ese método **se debe** invalidar en cualquier clase no abstracta que herede directamente de dicha clase.

Clases base abstractas
Puede declarar una clase como abstracta si desea evitar la creación directa de instancias por medio de la palabra clave new. Si hace esto, la clase solo se puede utilizar si una nueva clase se deriva de ella. Una clase abstracta no tiene que contener miembros abstractos; sin embargo, si una clase contiene un miembro abstracto, la propia clase se debe declarar como abstracta. Las clases derivadas que no son abstractas por sí mismas deben proporcionar la implementación de cualquier método abstracto de una clase base abstracta.

Interfaces
Una interfaz es un tipo de referencia similar en cierto modo a una clase base abstracta compuesta únicamente por miembros abstractos. Cuando una clase implementa una interfaz, debe proporcionar una implementación para todos los miembros de la interfaz. Una clase puede implementar varias interfaces aunque solo puede derivar de una única clase base directa.

10. HERENCIA (III) – Ejemplo de clase abstracta

```
public abstract class Animal
{
    private int cantidadSaltos = 0;
    public string Nombre { get; set; }

    public abstract void Correr();

    public virtual void Saltar()
    {
        cantidadSaltos++;
    }

    public Animal()
    {
        Nombre = "Pepe";
    }

    public Animal(string nombre)
    {
        Nombre = nombre;
        cantidadSaltos = 0;
    }
}
```

```
//La clase Perro hereda
//de la clase Animal
public class Perro : Animal
{
    public Perro():base()  { }

    public Perro(string nombre)
      : base(nombre)
    {
    }

    public override void Correr()
    {
        //Hacer algo...
    }
}
```

```
interface IControl {
    void Paint();
}
interface ISurface {
    void Paint();
}
class ClaseEjemplo : IControl, ISurface
{
   //ISurface.Paint e IControl.Paint invocan este método
   public void Paint() {
      Console.WriteLine ("Paint method in SampleClass");
   }
}

class Test {
    static void Main() {
        SampleClass sc = new SampleClass();
        IControl ctrl = (IControl)sc;
        ISurface srfc = (ISurface)sc;

        // Las siguientes líneas invocan al mismo método.
        sc.Paint();
        ctrl.Paint();
        srfc.Paint();
    }
}
```

ÍNDICE

11. POLIMORFISMO

El término polimorfismo es una palabra griega que significa "con muchas formas" y tiene dos aspectos que lo caracterizan:

1. En tiempo de ejecución, los objetos de una clase derivada se pueden tratar como objetos de una clase base en lugares como parámetros de método y colecciones o matrices. Cuando esto sucede, el tipo declarado del objeto ya no es idéntico a su tipo en tiempo de ejecución.

2. Las clases base pueden definir e implementar métodos ***virtuales*** y las clases derivadas pueden ***invalidarlos***, lo que significa que proporcionan su propia definición e implementación. En tiempo de ejecución, cuando el código de cliente llama al método, CLR busca el tipo en tiempo de ejecución del objeto e invoca esta invalidación del método virtual. Así, en el código fuente puede llamar a un método de una clase base y provocar la ejecución de la versión de clase derivada del método.

11. POLIMORFISMO – Ejemplo

```
//Clase base
public class Figura
{
    public int X { get; private set; }
    public int Y { get; private set; }
    public int Alto { get; set; }
    public int Ancho { get; set; }

    // Método Virtual
    public virtual void Draw()
    {
        Console.WriteLine("Dibujando desde la clase base");
    }
}
```

11. POLIMORFISMO – Ejemplo

```
class Circulo : Figura
{
    public override void Draw()
    {
        // Código para dibujar un cículo...
        Console.WriteLine("Dibujando un círculo");
        base.Draw();
    }
}
class Rectangulo : Figura
{
    public override void Draw()
    {
        // Código para dibujar un rectángulo...
        Console.WriteLine("Dibujando un rectángulo");
        base.Draw();
    }
}
class Triangulo : Figura
{
    public override void Draw()
    {
        // Código para dibujar un triángulo...
        Console.WriteLine("Dibujando un triángulo");
        base.Draw();
    }
}
```

```
public class Program {
    static void Main(string[] args) {
        // El "casteo" no es requerido ya que una conversión implícita es
        // realizada desde una clase derivada a su clase base.
        System.Collections.Generic.List<Figura> figuras =
             new System.Collections.Generic.List<Figura>();
        figuras.Add(new Rectangulo());
        figuras.Add(new Triangulo());
        figuras.Add(new Circulo());

        // El método virtual "Draw" es invocado en cada una de las clases
        // derivadas (no desde la clase base)
        foreach (Figura figura in figuras)
        {
            figura.Draw();
        }
    }
}
/* Salida:
    Dibujando un rectángulo
    Dibujando desde la clase base
    Dibujando un triángulo
    Dibujando desde la clase base
    Dibujando un círculo
    Dibujando desde la clase base
*/
```

11. POLIMORFISMO

Cuando una clase derivada hereda de una clase base, obtiene todos los métodos, campos, propiedades y eventos de la clase base. El diseñador de la clase derivada puede elegir si

-**invalida** los miembros virtuales de la clase base
-**hereda** el método de clase base más parecido sin invalidarlo
-**define** una nueva implementación no virtual de los miembros que ocultan las implementaciones de la clase base

Una clase derivada solo puede invalidar un miembro de la clase base si éste se declara como virtual o abstracto. El miembro derivado debe utilizar la palabra clave ***override*** para indicar explícitamente que el método va a participar en la invocación virtual.

```
public class BaseClass {
    public virtual void DoWork() { }
    public virtual int WorkProperty {
        get { return 0; }
    }
}

public class DerivedClass : BaseClass {
    public override void DoWork() { }
    public override int WorkProperty {
        get { return 0; }
    }
}
```

ÍNDICE

12. GENÉRICOS

Los tipos genéricos se agregaron a la versión 2.0 del lenguaje C# y Common Language Runtime (CLR). Estos tipos agregan el concepto de parámetros de tipo a .NET Framework, lo cual permite diseñar clases y métodos que aplazan la especificación de uno o más tipos hasta que el código de cliente declara y crea una instancia de la clase o del método. Por ejemplo, mediante la utilización de un parámetro de tipo genérico T, se puede escribir una clase única que otro código de cliente puede utilizar sin generar el costo o el riesgo de conversiones en tiempo de ejecución u operaciones de conversión boxing, como se muestra a continuación:

```
public class GenericList<T> {
    void Add(T input) { }
}
class TestGenericList {
    private class ExampleClass { }
    static void Main() {
        // Declare a list of type int.
        GenericList<int> list1 = new GenericList<int>();

        // Declare a list of type string.
        GenericList<string> list2 = new GenericList<string>();

        // Declare a list of type ExampleClass.
        GenericList<ExampleClass> list3 = new GenericList<ExampleClass>();
    }
}
```

El espacio de nombres System.Collections.Generic contiene interfaces y clases que definen colecciones genéricas, permitiendo que los usuarios creen colecciones fuertemente tipadas para proporcionar una mayor seguridad de tipos y un rendimiento mejor que los de las colecciones no genéricas fuertemente tipadas.

En general, utilizamos la clase List<T>, la cual representa una lista de objetos fuertemente tipados a la que se puede obtener acceso por índice. La misma proporciona métodos para buscar, ordenar y manipular listas.

```
List<string> dinosaurs = new List<string>();
dinosaurs.Add("Tyrannosaurus");
dinosaurs.Add("Amargasaurus");
dinosaurs.Add("Mamenchisaurus");
dinosaurs.Add("Deinonychus");
dinosaurs.Add("Compsognathus");

foreach(string dinosaur in dinosaurs) { Console.WriteLine(dinosaur); }

Console.WriteLine("Count: {0}", dinosaurs.Count);
Console.WriteLine("\nContains(\"Deinonychus\"): {0}",
    dinosaurs.Contains("Deinonychus"));

Console.WriteLine("\nInsert(2, \"Compsognathus\")");
        dinosaurs.Insert(2, "Compsognathus");

foreach(string dinosaur in dinosaurs) { Console.WriteLine(dinosaur); }

Console.WriteLine("\ndinosaurs[3]: {0}", dinosaurs[3]);
Console.WriteLine("\nRemove(\"Compsognathus\")");
        dinosaurs.Remove("Compsognathus");

foreach(string dinosaur in dinosaurs)  { Console.WriteLine(dinosaur); }

dinosaurs.Clear();
```

FIN

Gracias

codes
CONSULTORIA
Y DESARROLLO DE SISTEMAS

CMMI