

Preguntas detonadoras



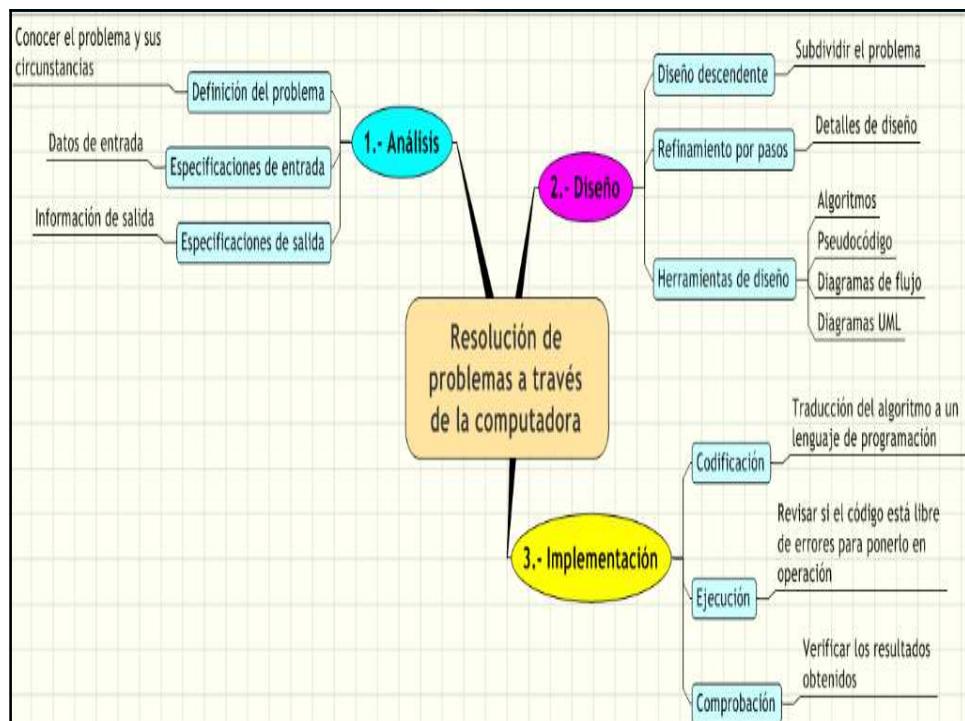
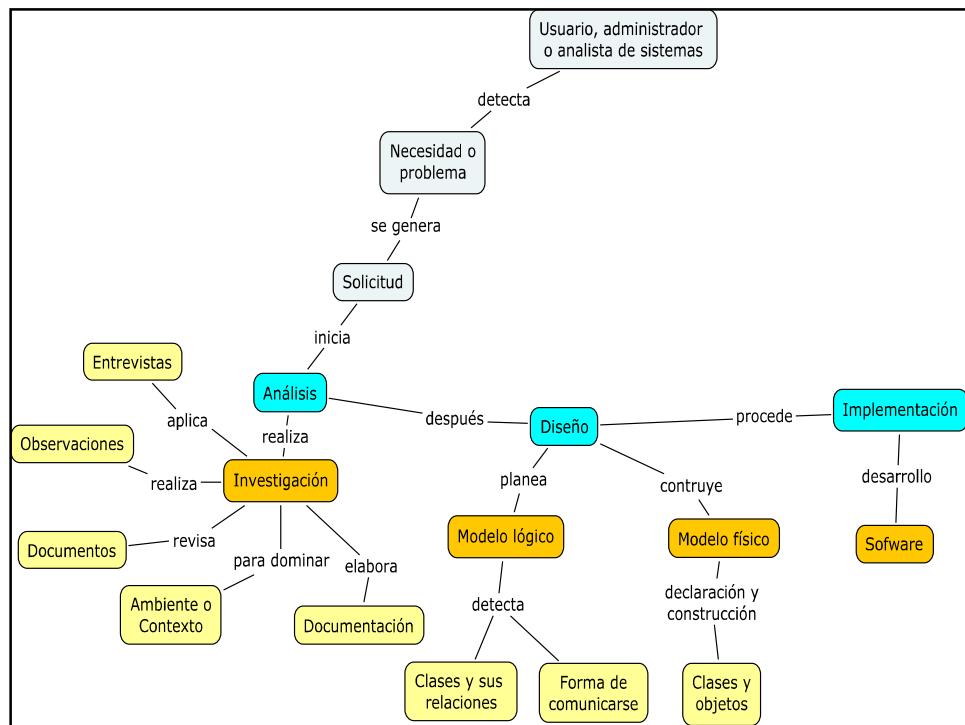
- ¿Qué es una clase?
- ¿Qué es un objeto?
- ¿Representa lo mismo una clase que un objeto?
- Diversos objetos creados a partir de la misma clase, ¿son iguales?
- ¿Qué significa el término instancia?
- ¿Cómo se logra que un objeto almacene datos y también realice acciones?
- ¿Qué es un atributo, propiedad y método?
- ¿Cuál es la diferencia entre atributo y propiedad?
- ¿Cómo se diseña el modelo de una aplicación orientada a objetos?

3

Resolución de problemas a través de la computadora

- 1. Análisis:** ¿Qué ...?
 - ¿Qué problema debe resolverse?
 - ¿Qué datos se requieren?
 - ¿Qué resultados debe arrojar el Sistema?
- 2. Diseño:** ¿Cómo ...?
 - ¿Cómo atacar el problema?
 - ¿Cómo plantear el modelo de solución?
 - ¿Cómo aplicar el modelo de solución?
- 3. Implementación:** ¿Con qué ...?
 - ¿Con qué lenguaje se desarrolla el modelo?
 - ¿Con qué plataforma de desarrollo?
 - ¿Con qué recursos de hardware y software?

4



Programación Orientada a Objetos

- POO es un conjunto de técnicas que pueden utilizarse para desarrollar programas eficientemente.
- Los objetos son los elementos principales de construcción.
- La Orientación a Objetos (OO) es el estilo dominante de programación, descripción y modelado de hoy en dia.

7

La POO es ...

“Un método de implementación en el que los programas se organizan como colecciones cooperativas de objetos, cada uno de los cuales representa una instancia de alguna clase y cuyas clases son todos miembros de una jerarquía de clases unidas mediante relaciones”

Grady Booch

8

El modelo de Objetos

- Objetos en el mundo real
 - Atributos
 - Propiedades
 - Métodos
- Abstracción
- Clases y Objetos
- Encapsulamiento
- Mensajes
- Constructores
- Destructor
- Herencia
 - Simple
 - Múltiple

- Clases abstractas
- Clases parametrizadas
- Interfaces
- Sobrescritura
- Sobrecarga
- Polimorfismo



9

Objetos en el mundo real



Lavadora



Perro



Televisión



Persona



Factura

10

Podemos darnos cuenta que...



- Los objetos poseen **características** que los distinguen entre sí.
- Los objetos tienen **acciones** asociadas a ellos.

11

Ejemplo: PERRO



- **Características:**

- Nombre: “FIDO”
- Raza: “Chihuahua”
- Color: “Café”
-etc...

- **Acciones:**

- Ladrar [“Guau Guau”]
- Comer [“Chomp Chomp”]
- Dormir [“Zzzzzzz”]
- ...etc...

12

¿Cómo modelar un objeto real en un programa?

- Las “características” son **ATRIBUTOS** o datos.
- Las “acciones” son **MÉTODOS** u operaciones.



FIDO : Perro
Nombre: FIDO
Raza: Chihuahua
Color: Café
Ladrar()
Comer()
Dormir()

Abstracción de un objeto “Perro” en software

13

Todos los objetos tienen Estado, Comportamiento e Identidad

Valor de sus características (Atributos) 	Acciones que puede realizar (Métodos) 	Pertenece a una clase y tiene un nombre único 
ESTADO	COMPORTAMIENTO	IDENTIDAD

14

Abstracción

- Se refiere a “quitar” atributos, propiedades y métodos de un objeto y quedarse solo con aquellos que sean necesarios (relevantes para el problema a solucionar).



Objeto Perro “Real”:

Características o atributos:

(Nombre, Raza, Color, **Edad, Tamaño, etc.**)

Acciones o métodos:

(Ladrar, Comer, Dormir, **Jugar, Caminar, etc.**)

FIDO : Perro

Nombre: FIDO
Raza: Chihuahua
Color: Café

Ladrar()
Comer()
Dormir()

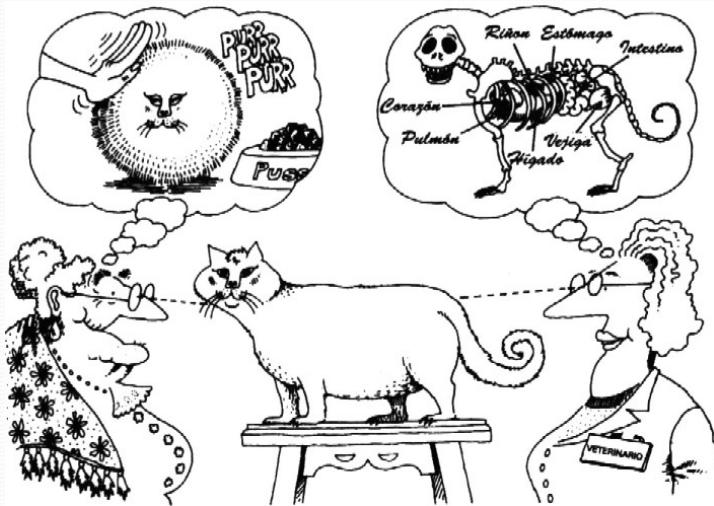
Nótese que en la “Abstracción” del perro quitamos varias características y acciones.

Abstracción de un “Perro”

15

Abstracción

La abstracción se centra en las características esenciales de algún objeto, en relación a la perspectiva del observador.



16

Abstracción

Las clases y objetos deben estar al nivel de abstracción adecuado: ni demasiado alto ni demasiado bajo.



MIRA EL CUADRUPEDO ACOSTADO EN LA TERRAZA

17

Encapsulamiento



- Permite incluir en una sola entidad información y operaciones que controlan dicha información.
- Permite:
 - Componentes públicos [Accesibles, Visibles].
 - Componentes privados [No accesibles, Ocultos].
 - Restricción de accesos indebidos.

18

El encapsulamiento oculta detalles de la implementación de un objeto.

Encapsulamiento

19

Ejemplo: Encapsulamiento

La Televisión oculta algunos componentes y operaciones de la persona que la ve.

- Los objetos encapsulan lo que hacen. Ocultan la funcionalidad interna de sus operaciones, de otros objetos y del mundo exterior.

20

Ejemplo Encapsulamiento



Componentes privados - Ocultos
(NO Accesibles desde el exterior)
Circuitos, cables

Aunque TODOS los componentes de un objeto se comuniquen entre sí internamente, algunos componentes son visibles al exterior y otros permanecen ocultos por motivos de seguridad e integridad del objeto.

Componentes accesibles desde el exterior
(Interfaz público)
Botones para cambiar el canal, subir/bajar el volumen

21

Mensajes entre Objetos

Los objetos realizan acciones cuando reciben mensajes



Mensaje recibido: Encender
Acción realizada: Se muestra imagen

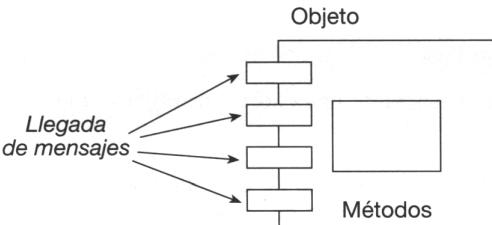
Mensaje recibido: Enciende la TV
Acción realizada: Envía orden de encendido a la TV

22

Mensajes: Comunicación entre objetos



- **Mensaje.**- Orden que se envía al objeto para indicarle realizar una acción.
- **Mensaje.**- Llamada a un método (o función) del objeto.



The diagram illustrates the concept of a message protocol. It shows a central box labeled "Objeto" (Object) containing several horizontal rectangles representing "Métodos" (Methods). An external arrow labeled "Llegada de mensajes" (Message arrival) points towards the object, indicating that messages are sent to the object to invoke its methods.

Al conjunto de mensajes a los cuales puede responder un objeto se llama “Protocolo del Objeto”

23

Clase



- Es una descripción de las características y acciones para un tipo de objetos.
- Una clase NO es un objeto. Es solo una plantilla, plano o definición para crear objetos.

24

Clase



- Contiene todas las características comunes de ese conjunto de objetos
- Clase = Modelo = Plantilla = Esquema = Descripción de la anatomía de los objetos.
- A partir de una clase se pueden crear muchos objetos independientes con las mismas características.

25

Objeto



- Unidad que combina datos y funciones.
 - Datos = Atributos = Características
 - Funciones = Métodos = Procedimientos =Acciones
- Un objeto es creado a partir de una clase.
- Los datos y funciones están Encapsulados.
- Posee un nombre único (identificador).
- Un objeto es del tipo de una clase
- “Un objeto es la instancia de una clase”
- Un objeto es un ejemplar específico creado con la estructura de una clase.

26

Instancia

- Es la creación o manifestación concreta de un objeto a partir de su clase



27

Clases y Objetos

- “FIDO” es UN “PERRO”
- “FIDO” es del TIPO “PERRO”
- “FIDO” es un OBJETO
- “PERRO” es la CLASE de “FIDO”
- “CHESTER” es OTRO “PERRO”
- “CHESTER” también es del TIPO “PERRO”
- “CHESTER” es otro OBJETO
- “PERRO” también es la clase de “CHESTER”



28

Atributos

- Representan los datos de los objetos
- Son controlados a través de la declaración de variables
- Es importante identificar el tipo de dato
- Se debe seleccionar sólo aquellos atributos necesarios para el modelo planteado (abstracción)



29

Atributos

Variables

De instancia

Se crea una copia por cada objeto creado

De clase (estáticas)

Una sola variable para todos los objetos generados

30

Ejemplo: Atributos de un estudiante

□ Atributos:

- claveMatrícula: "A-233"
- nombre: "Bruno López Takeyas"
- grado: 3
- grupo: 'A'
- promedio: 87.4

```
string claveMatricula;  
string nombre;  
int grado;  
char grupo;  
float promedio;
```

31

Métodos

- Son las acciones que realizan los objetos y definen su comportamiento

□ Atributos:

- claveMatrícula: "A-233"
- nombre: "Bruno López Takeyas"
- grado: 3
- grupo: 'A'
- promedio: 87.4

```
void Leer()  
void Investigar()
```

□ Acciones:

- Leer()
- Investigar()

32

Propiedades

- Son mecanismos que permiten acceder a los atributos de un objeto.
- Algunos autores asumen que las propiedades son sinónimos de los datos
- En un sentido estricto, las propiedades actúan como un canal de comunicación para acceder a un atributo, ya sea para consultar o modificar su valor.
- Descriptores de acceso: **get** y **set**.

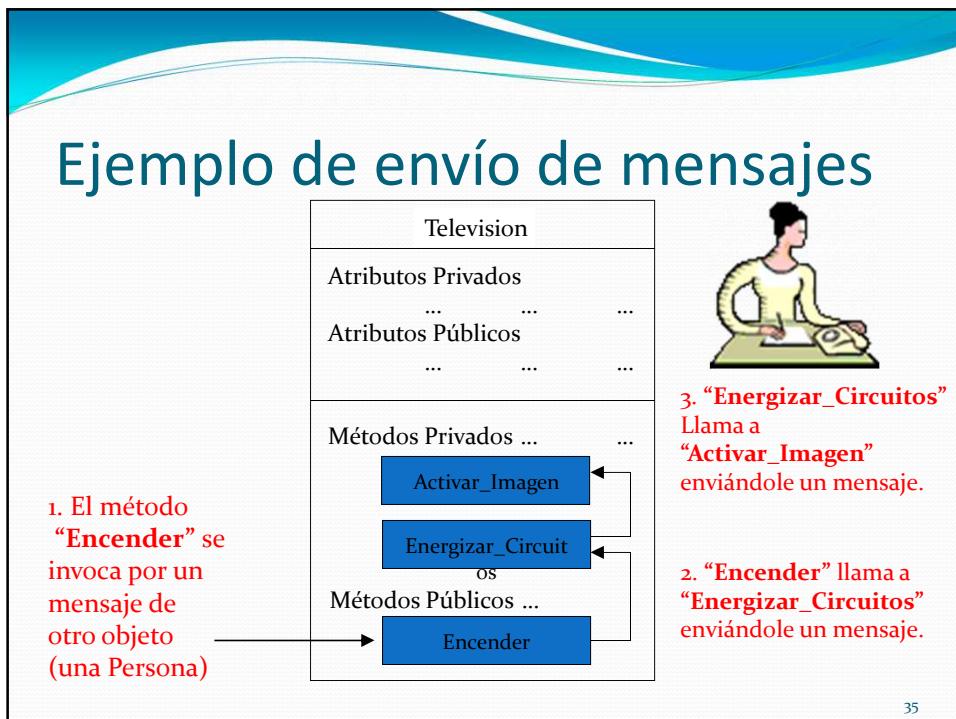
33



Anatomía de un mensaje

- Identidad del receptor
 - Método que ha de ejecutar
 - Información especial (argumentos o parámetros)
-
- Ejemplos:
 - miTelevision.Encender()
 - miTelevision.Apagar()
 - miTelevision.CambiarCanal(45)
 - miPerro.Comer("Croquetas")
 - miEmpleado.Contratar ("Juan", 3500)
 - miFactura.Imprimir()

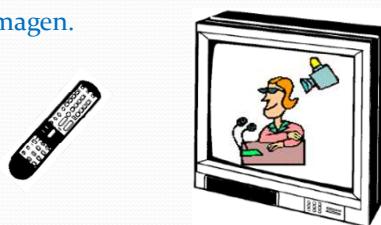
34



Ejemplo de constructor y destructor

Cada vez que se **enciende** la Television...

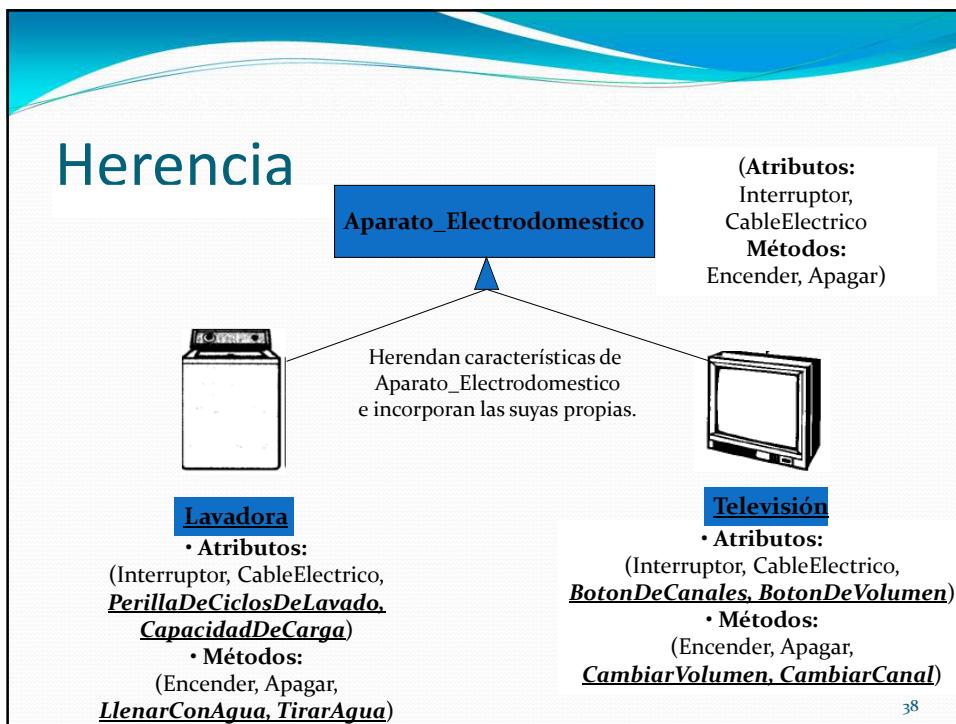
- Se deben energizar los circuitos
- Se debe activar el cinescopio
- ...Para posteriormente mostrar la imagen.



Cada vez que se **apaga** la Television...

- Se deben des-energizar los circuitos
- Se debe des-activar el cinescopio
- ...Para posteriormente apagar la imagen

37



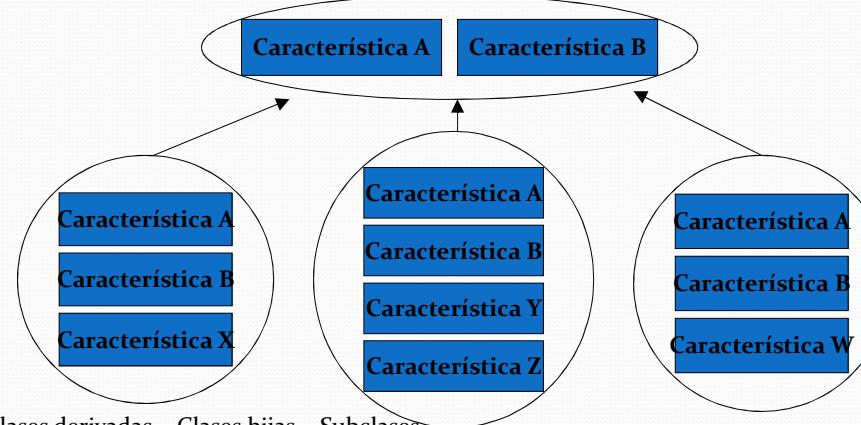
Herencia

- Capacidad para utilizar características previstas en antepasados o ascendientes.
- Permite construir nuevas clases a partir de otras ya existentes, permitiendo que éstas les “transmitan” sus propiedades.
- Objetivo: Reutilización de código.

39

Herencia - Jerarquía de clases

Clase Base = Super clase = Clase madre = Clase padre

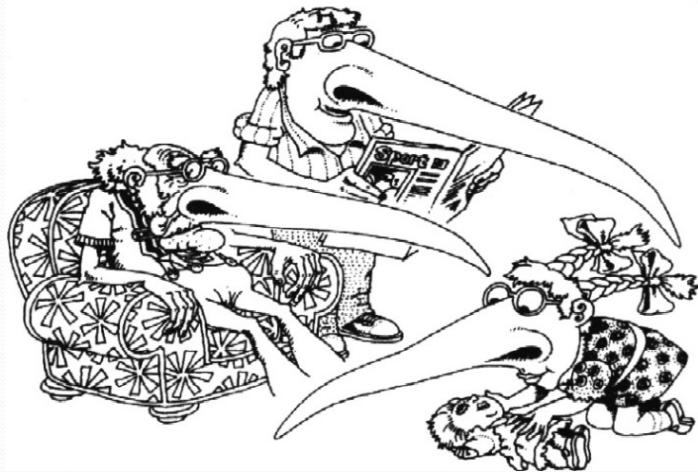


Clases derivadas = Clases hijas = Subclases

40

Herencia

Una subclase hereda el comportamiento y la estructura de su Super Clase

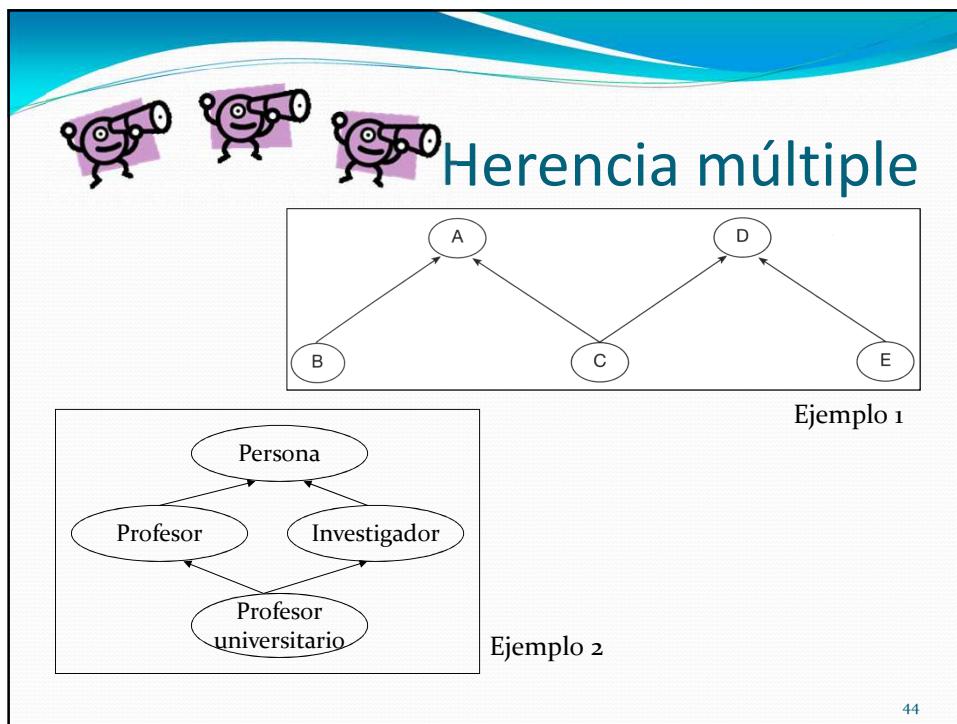
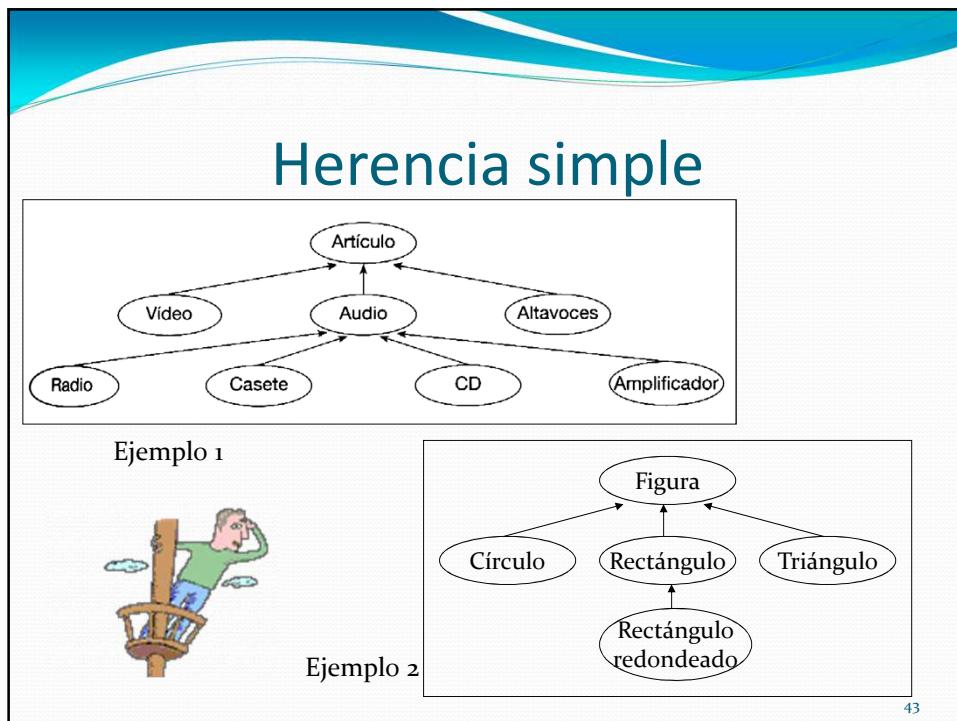


41

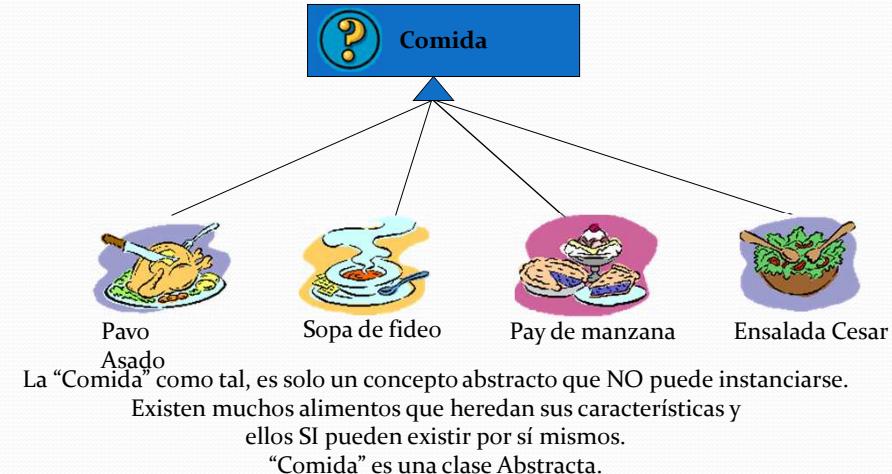
Tipos de Herencia

- **Herencia Simple.**- Una clase puede tener sólo un ascendiente. [Una subclase puede heredar de una única clase].
- **Herencia múltiple (en malla).**- Una clase puede tener más de un ascendiente inmediato. [Heredar de más de una clase].

42



Clase abstracta



45

Clase abstracta



- Es una clase que sirve como clase base común, pero **NO** puede tener instancias.
- Una clase abstracta solo puede servir como clase base (solo se puede heredar de ella).
- Sus clases "hijas" SI pueden tener instancias.

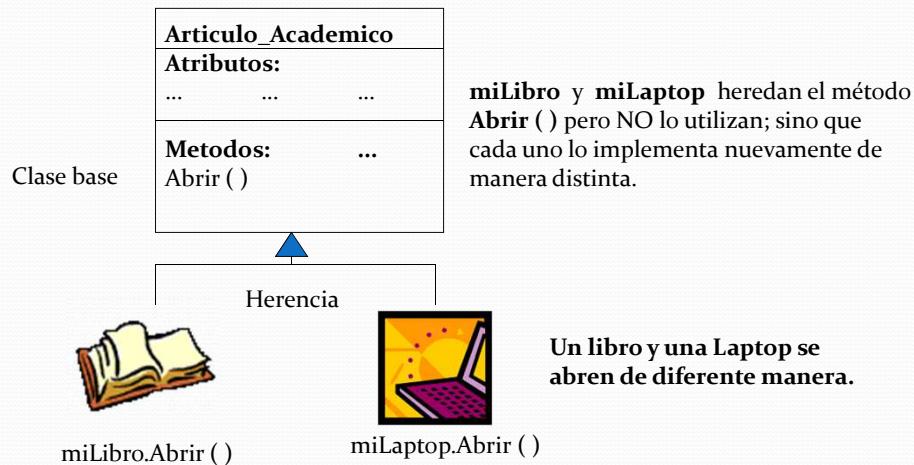
46

Anulación / Sustitución / sobrescritura [Overriding]

- Sucede cuando una clase “B” hereda características de una clase “A”, pero la clase “B” re-define las características heredadas de “A”.
- Propiedades y métodos pueden heredarse de una superclase. Si estas propiedades y métodos son re-definidos en la clase derivada, se dice que han sido “Sobrescritos”.

47

Anulación / Sustitución / sobrescritura [Overriding]



48

Sobrecarga [Overload]

- La sobrecarga representa diferentes maneras de realizar una misma acción.
- En los programas se usa el mismo nombre en diferentes métodos con diferentes firmas [número, orden y tipo de los parámetros].
- El código de programación asociado a cada sobrecarga puede variar.
- Ejemplos:
 - miEmpleado.Contratar("Juan", "Ventas", 2500)
 - miEmpleado.Contratar("Juan")
 - miEmpleado.Contratar("Juan", 2500)

49

Ejemplo de Sobrecarga [Overload]



miPuerta.Abrir (Adentro, Afuera)



miPuerta.Abrir (Afuera, Adentro)



miPuerta.Abrir ()

50



Polimorfismo

Se refiere a:

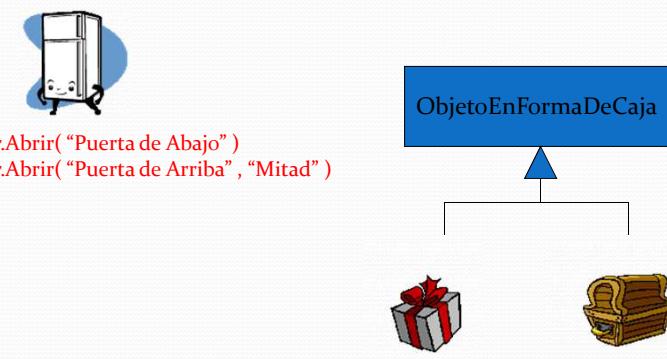
1. Es el uso de un mismo nombre para representar o significar más de una acción.
 - La sobrecarga es un tipo de Polimorfismo.
2. Que un mismo mensaje pueda producir acciones totalmente diferentes cuando se recibe por objetos diferentes del mismo tipo.
 - Un usuario puede enviar un mensaje genérico y dejar los detalles de la implementación exacta para el objeto que recibe el mensaje en tiempo de ejecución.
 - Para este caso, se utiliza herencia y sobrescritura (Override).

51



Polimorfismo

POLI = Múltiples MORFISMO = Formas



```
graph TD; A[ObjetoEnFormaDeCaja] --> B[miRegalo]; A --> C[miCofre]
```

miRefrigerador.Abrir("Puerta de Abajo")
miRefrigerador.Abrir("Puerta de Arriba" , "Mitad")

miRegalo.Abrir() miCofre.Abrir()

52

Software

- **NClass** es software para el diseño de diagramas de clases.
- Puede descargarse de manera gratuita en:
 <http://nklass.sourceforge.net>

53

Otros títulos del autor

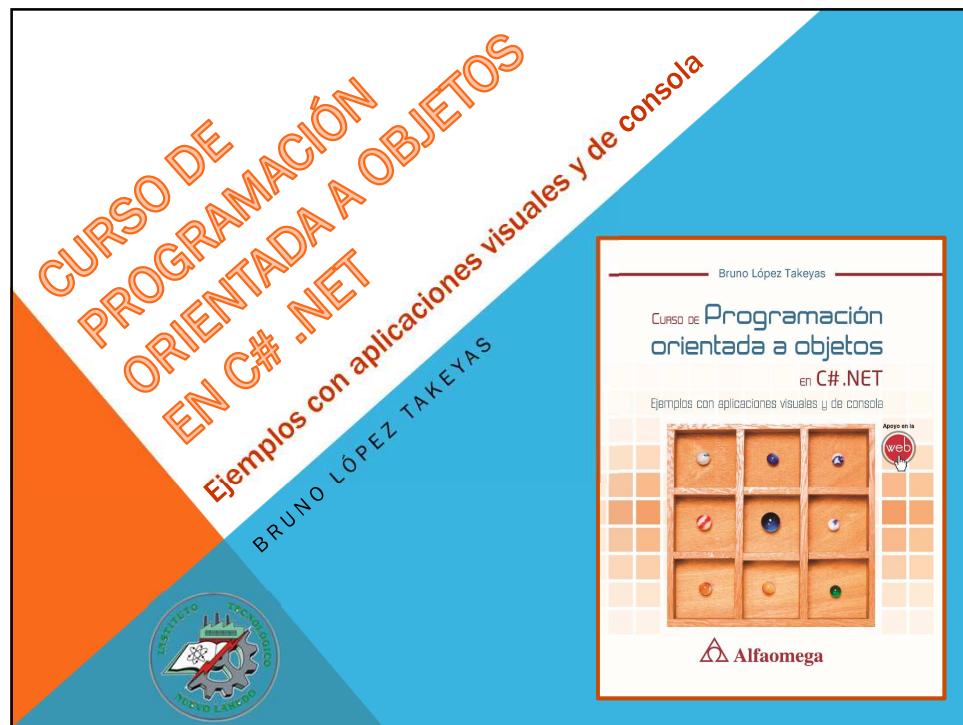
<http://www.itnuevolaredo.edu.mx/Takeyas/Libro>



Introducción a la ISC y al diseño de algoritmos
Bruno López Takeyas
Segunda edición
PEARSON

ESTRUCTURAS DE DATOS ORIENTADAS A OBJETOS
PSEUDOCÓDIGO Y APLICACIONES EN C# .NET
BRUNO LÓPEZ TAKEYAS
Alfaomega

 bruno.lt@nlaredo.tecnm.mx  Bruno López Takeyas



Preguntas detonadoras



- ¿Cómo se pueden establecer restricciones de acceso a los componentes definidos en una clase?
- ¿Qué es un mutator y un accessor? ¿Para qué sirven?
- ¿A qué se refiere la sentencia this? ¿Para qué sirve? ¿Cuándo se utiliza?
- ¿Cuándo se recomienda utilizar una propiedad autoimplementada?
- ¿Se pueden definir varios métodos con el mismo nombre en una clase?
- El constructor de una clase, ¿crea un objeto?
- El destructor de una clase, ¿elimina un objeto?

3

Espacios de nombres (namespace)

- Organizan los diferentes componentes
- Un programa puede contener varios namespaces
- Un namespace puede contener muchas clases
- El programador puede crear sus propios namespaces
- Para accesar a namespaces se usa la directiva using:
 - using System;
 - using System.Array;

4

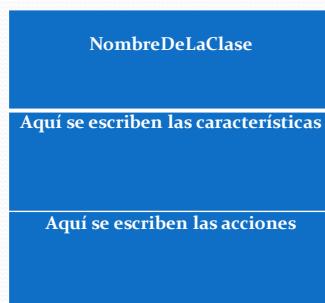
Clases y objetos

- Una clase es básicamente un plano para un tipo de datos personalizado.
- Cuando se define una clase, se utiliza cargándola en la memoria.
- Una clase que se ha cargado en la memoria se denomina *objeto* o *instancia*.
- Se crea una instancia de una clase utilizando la palabra clave de C# *new*

5

Clases en UML

- Cada clase se representa en un rectángulo con tres compartimentos:
 - Nombre
 - Atributos
 - Métodos y propiedades



6

Cómo declarar una clase

```
class nombre_de_la_clase  
{  
    ... contenido de la clase ...  
}
```

7

Dentro de la clase...

- Se pueden declarar variables, propiedades, métodos, delegados, eventos, etc.
- Cada elemento puede tener un modificador de acceso.
- Un modificador de acceso especifica quienes están autorizados a “ver” ese elemento.
- Si no se especifica ningún modificador de acceso, se asume que se trata de un elemento “private”.

8

Modificadores de acceso

- **public**

Accesible a todos los elementos

- **private**

Accesible solo a esa misma clase

- **protected**

Accesible solo a la misma clase y métodos de sus clases derivadas. No accesible desde el exterior.

- **internal**

Accesible solo a ese ensamblado

- **protected internal**

Accesible desde el mismo ensamblado, la misma clase y métodos de sus clases derivadas

9

Modificadores de acceso

Modificador de acceso	Accesible desde ...				
	Clase donde se declaró	Subclase (Mismo assembly)	Subclase (Distinto Assembly)	Externamente (Mismo Assembly)	Externamente (Distinto Assembly)
private	SI	NO	NO	NO	NO
internal	SI	SI	NO	SI	NO
protected	SI	SI	SI	NO	NO
protected internal	SI	SI	SI	SI	NO
public	SI	SI	SI	SI	SI

10

Ejemplo: Variables con modificadores de acceso

```
class claseA
{
    //Si no se indica, es private
    int numero;
    private int numero1;

    public int numero2;
    protected int numero3 = 99;
    internal int numero4;
    protected internal int numero5;
}
```

11

Representación de modificadores de acceso en C# y UML

Modificador de acceso	Codificación en C#	Representación en UML
Privado	private	-
Público	public	+
Protegido	protected	#
Interno	internal	~
Protegido interno	protected internal	#

12

Miembros estáticos y de instancia

- ***Miembro estático (static)***: Sólo se crea una copia del miembro de la clase. Se crea cuando se carga la aplicación que contiene la clase y existe mientras se ejecute la aplicación
- ***Miembro de instancia***: Se crea por default. Se crea una copia para cada instancia de la clase.

13

Miembros estáticos y de instancia

- Un miembro estático es un método o campo al que se puede obtener acceso sin hacer referencia a una instancia determinada de una clase
- No es necesario crear una instancia de la clase contenedora para llamar al miembro estático
- Cuando se tiene acceso a métodos estáticos, puede utilizar el nombre de clase, no el nombre de instancia

14

Declaración de miembros estáticos

- Cuando declara un campo de clase estático, todas las instancias de esa clase compartirán ese campo
- Una clase estática es una cuyos miembros son todos estáticos

```
static void Main(string[] args)
{
    Console.WriteLine("Tec Laredo");
}
```

Miembro estático

15

Atributos o campos

- Un atributo o campo es un dato común a todos los objetos de una determinada clase.
- Las variables declaradas dentro de una clase son ejemplos de atributos o campos

16

Ejemplo de una clase

CuentaBancaria

```
+ strNombreCliente: string  
+ dtmFechaDeInversion: DateTime  
- _dblSaldo: double  
  
+ CalcularDiasDeInversion() : int  
- ConsultarSaldo() : double  
+ Depositar(dblCantidad: double, dtmFecha: DateTime) : void  
+ OtorgarBono() : bool
```

17

Ejemplo de una clase

```
class CuentaBancaria  
{  
    // Declaración de los atributos  
    public string strNombreCliente;  
    public DateTime dtmFechaDeInversion;  
    private double _dblSaldo;  
  
    // Definición de los métodos  
    public int CalcularDiasDeInversion() {  
        // Aquí se escribe el código de este método  
    }  
  
    private double ConsultarSaldo() {  
        // Aquí se escribe el código de este método  
    }  
  
    public void Depositar(double dblCantidad, DateTime dtmFecha) {  
        // Aquí se escribe el código de este método  
    }  
  
    public bool OtorgarBono() {  
        // Aquí se escribe el código de este método  
    }  
}
```

18

Crear objetos: Instanciación

- Una vez creada la clase, ya es posible “consumirla” mediante la instancia.
- La instancia es el proceso de crear objetos a partir de una clase.

- Ejemplo en dos pasos:

```
ClaseA miobjetoA;  
miobjetoA = new ClaseA();
```

1. Definir el tipo de “miObjetoA”
2. Adquirir memoria sin inicializar para el nuevo objeto usando new.
3. Ejecutar el constructor para inicializar la memoria y convertirla en un objeto usable

- Ejemplo en un paso:

```
ClaseA miobjetoA = new ClaseA();
```

19

Objetos en UML

- En UML un objeto se representa como un rectángulo con un nombre subrayado



Los objetos no se crean automáticamente cuando se declaran

20

Declaración de objetos globales

```
namespace CircunferenciaFormas
{
    public partial class Form1 : Form
    {

        // Aquí se declaran los objetos globales

        // Declaración del objeto global
        CuentaBancaria cuentaCheques;
        // Creación del objeto global
        cuentaCheques = new CuentaBancaria();

        // Declaración y creación del objeto global en una sola línea
        CuentaBancaria cuentaAhorros = new CuentaBancaria();

        public Form1()
        {
            InitializeComponent();
        }

        ...
    }
}
```

Accediendo a los miembros de los objetos

Asignación:

miobjetoA . Numero2 = 50;

The diagram shows the assignment statement "miobjetoA . Numero2 = 50;". Three arrows point to different parts of the statement: one arrow points to "miobjetoA" with the label "Nombre del objeto"; another arrow points to the ". " operator with the label "Selección"; and a third arrow points to "Numero2" with the label "Miembro".

Lectura:

miOtraVariable = miobjetoA . Numero2;

22

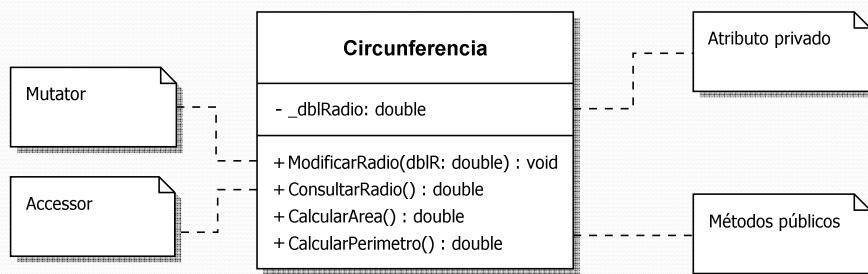
Métodos

- Contienen instrucciones para ejecutar al momento de ser invocados.
- Un método contiene:
 - Modificador de Acceso (Determina su visibilidad)
 - Tipo de dato (Devuelto al finalizar su ejecución)
 - Identificador (Nombre con el cual se invoca)
 - Parámetros (Cero o mas variables que recibe el método)

 No es recomendable diseñar clases cuyos métodos implementen código para capturar y/o mostrar los valores de sus atributos, ya que se pierde la reusabilidad de esa clase en aplicaciones con otras plataformas

23

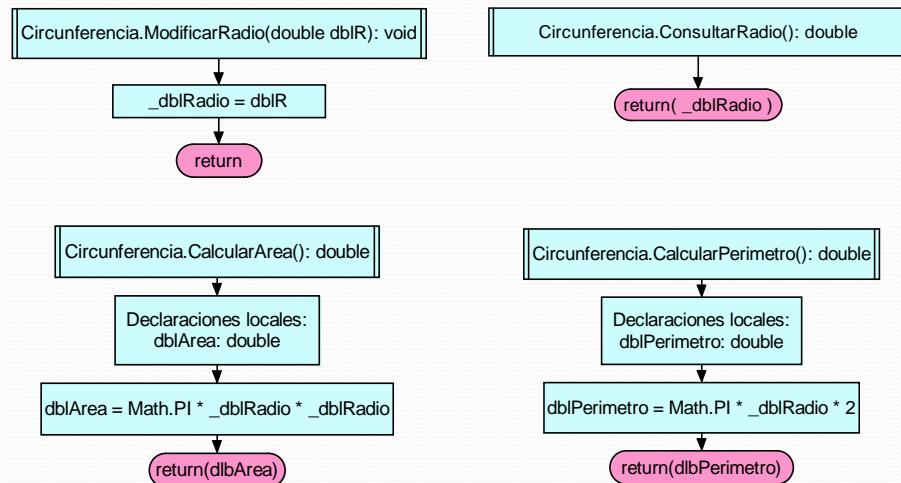
Uso de mutator y accessor



 Al trabajar con objetos, primero deben introducirse los valores de sus atributos y después ejecutar las acciones invocando sus métodos.

24

Diagramas de flujo de los métodos de una clase



25

Codificación de la clase

```

class Circunferencia
{
    // Declaración del atributo privado
    private double _dblRadio;

    // Mutador
    public void ModificarRadio(double dblR)
    {
        _dblRadio = dblR;
    }

    // Accesor
    public double ConsultarRadio()
    {
        return (_dblRadio);
    }

    // Método público para calcular el área
    public double CalcularArea()
    {
        // Declaración de variable local
        double dblArea;

        dblArea=Math.PI * _dblRadio * _dblRadio;

        return (dblArea); // Devuelve el resultado
    }

    // Método público para calcular el perímetro
    public double CalcularPerimetro()
    {
        // Declaración de variable local
        double dblPerimetro;

        dblPerimetro=Math.PI * _dblRadio * 2;

        return (dblPerimetro); // Devuelve el resultado
    }
}
  
```

26

La referencia *this*

- Para hacer referencia (explícita) a un elemento que se encuentra dentro de la misma clase (ésta) se utiliza “this”.

```
class Articulo
{    private double precio = 0;
    public void PonerPrecio ( double precio )
    {        this.precio = precio;
    }
    public double ConsultarPrecio()
    {        return this.precio;
    }
}
```

Muy útil cuando existen distintos elementos con el mismo nombre, pero con distinto significado dentro de un ámbito determinado.

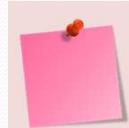
27

Ejemplo de uso de la referencia *this*

Persona

- nombre: string
- edad: int

+ ModificarDatos(nombre: string, edad: int) : void



El uso de la palabra *this* para referirse a los miembros internos de una clase es opcional, pero es necesaria cuando un parámetro y un atributo comparten el mismo nombre

28

Codificación de la referencia *this*

```
class Persona
{
    // Declaración de los atributos privados
    private string nombre;
    private int edad;

    // Mutator
    public void ModificarDatos(string nombre, int edad)
    {
        this.nombre = nombre;
        this.edad = edad;
    }
}
```

29

Propiedades

- Son mecanismos para acceder a los valores (variables) que poseen los objetos.
- Algunos autores las consideran sinónimos de los datos, sin embargo no siempre lo son (solamente en el caso de las propiedades autoimplementadas).
- Se pueden implementar:
 - **Declarando la variable como public**
 - No sería posible especificarla como Solo-Lectura o Solo-Escritura
 - No se puede implementar código adicional asociado a la Lectura o Escritura de la variable.
 - **Declarando un método de acceso a la variable**
 - Empobrece la legibilidad del código. Puede crear confusión.
 - **Utilizando accesadores get{ } y set { } para las propiedades**
 - Recomendado en C#

30

Implementando propiedades con accesadores get, set

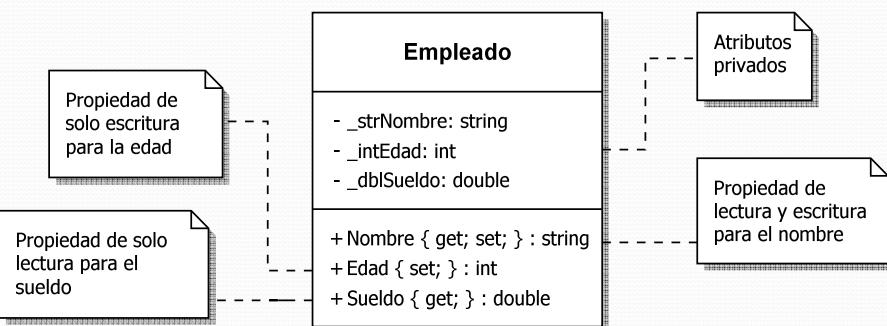
(Manera recomendada)

```
class Persona
{
    private int _intEdad;
    public int Edad
    {
        get
        {
            return _intEdad;
        }
        set
        {
            _intEdad = value;
        }
    }
}

class Programa
{
    static void Main()
    {
        Persona unaPersona = new
        Persona();
        unaPersona.Edad = 25;
        System.Console.WriteLine
            (unaPersona.Edad);
    }
}
```

31

Ejemplo de clase con propiedades



32

Codificación de la clase y sus propiedades

```
class Empleado
{
    // Declaración de los atributos privados
    private string _strNombre;
    private int _intEdad;
    private double _dblSueldo;

    // Propiedad pública de lectura y escritura del nombre
    public string Nombre
    {
        get
        {
            return _strNombre;
        }
        set
        {
            _strNombre = value;
        }
    }

    // Propiedad pública de solo escritura para la edad
    public int Edad
    {
        set
        {
            _intEdad = value;
        }
    }

    // Propiedad pública de solo lectura para el sueldo
    public double Sueldo
    {
        get
        {
            return _dblSueldo;
        }
    }
}
```

33

Validar datos mediante propiedades

```
// Propiedad pública de solo escritura para la edad

public int Edad
{
    set
    {
        if(value>=0) // valida que la edad sea un número positivo
            _intEdad = value; // modifica el valor del atributo
        else // si no
            _intEdad = 0; // arbitrariamente le asigna el valor cero
    }
}
```

34

Propiedades de solo lectura

- Si se desea que una propiedad sea de solo lectura, solo se debe incluir "get".

```
class Cuenta
{
    private double _dblSaldo=0;
    public double Saldo
    {
        get { return _dblSaldo; }
    }
}
class Programa
{
    static void Main()
    {
        Cuenta miCuenta = new Cuenta();
        //miCuenta.Saldo = 32; ←
        System.Console.WriteLine(miCuenta.Saldo);
        System.Console.ReadLine();
    }
}
```

Realizar
una
asignación
provocaría
un error

35

Nivel de acceso asimétrico en las propiedades

- Si una propiedad posee "get" y "set", entonces uno de los dos puede tener un modificador de acceso explícito.
- El nivel de acceso explícito de "get" o "set" debe ser mas restrictivo que el usado en la propiedad a la que pertenece.
- No se pueden utilizar en interfaces o implementación de ellas.

"get" sigue siendo public

```
class Empleado
{
    private int _intNumero;
    [public] int Numero
    {
        get
        {
            return _intNumero;
        }
        [private] set
        {
            _intNumero = value;
        }
    }
}
```

36

Propiedades autoimplementadas

Propiedades
autoimplementadas

Empleado

+ Nombre { get; set; } : string
+ Edad { get; set; } : int
+ Sueldo { get; set; } : double

 La propiedad autoimplementada debe tener los descriptores de acceso get y set; es decir, no puede ser de solo lectura o solo escritura

37

Codificación de propiedades autoimplementadas

```
class Empleado
{
    // Propiedades autoimplementadas
    public string Nombre
    {
        get;
        set;
    }

    public int Edad
    {
        get;
        set;
    }

    public double Sueldo
    {
        get;
        set;
    }
}
```

38

Sobrecarga

- Se refiere a la posibilidad de crear métodos que posean el mismo nombre y difieran en la cantidad y/o tipo de parámetros (Firma).
 - No es posible crear 2 funciones con el mismo identificador que solo difieran en el tipo de dato devuelto.
 - No es posible que 2 métodos sobrecargados solo difieran en el modificador “static”.

```
static bool impresion(int num)  
static char impresion(int num)
```



39

Ejemplo de sobrecarga

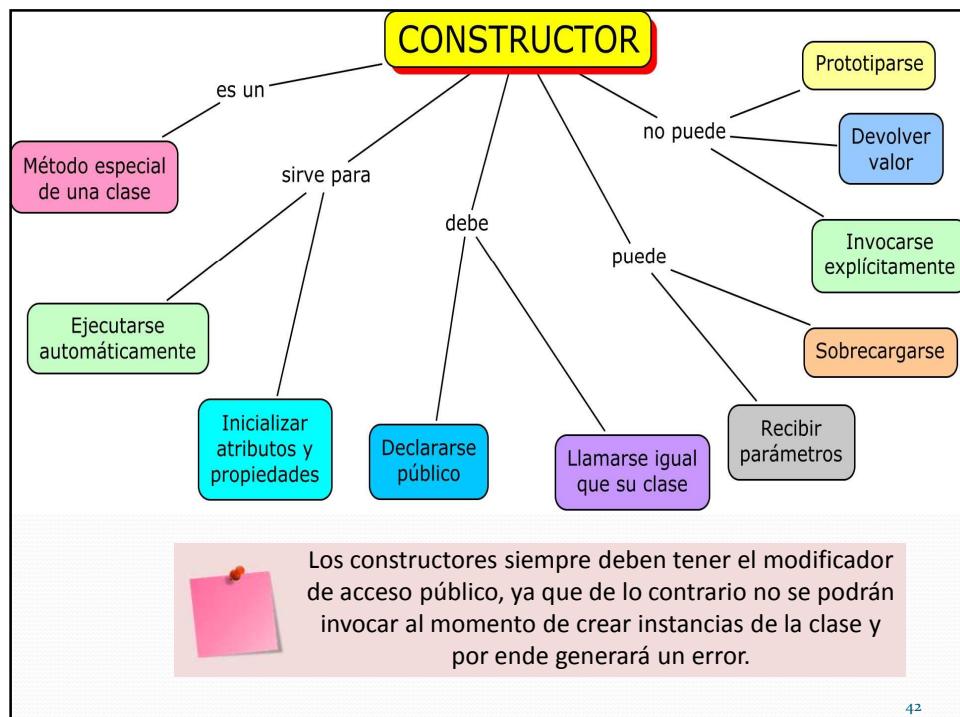
```
class Impresion  
{    public void Imprimir()  
    {        System.Console.WriteLine("Sistema de Usuarios");  
    }  
    public void Imprimir(string mensaje)  
    {        System.Console.WriteLine(mensaje);  
    }  
    public void Imprimir (string titulo, string mensaje)  
    {        System.Console.WriteLine("==== " + titulo + " ====");  
        System.Console.WriteLine(mensaje);  
    }  
}  
class Programa  
{    static void Main()  
{  
    Impresion im = new Impresion ();  
    im.Imprimir();  
    im.Imprimir("Siga las instrucciones en pantalla", "Use minusculas");  
    im.Imprimir(" Gracias! ");  
    System.Console.ReadLine();  
}
```

40

Método constructor

- Método especial que es invocado automáticamente cada vez que un objeto nuevo es creado (new).
- Debe poseer el mismo nombre de la clase.
- No posee tipo de dato de retorno.
- Si la clase no posee un constructor, C# crea uno vacío por default.

41



42

Ejemplo de constructor

```
class Persona
{
    private string _strNombre;

    public Persona()
    {
        _strNombre = "Desconocido";
    }
}

class Programa
{
    static void Main()
    {
        Persona unaPersona = new Persona();
        System.Console.WriteLine(unaPersona.Nombre);
        System.Console.ReadLine();
    }
}
```

Constructor.
(notar que tiene el mismo nombre de la clase)

Dentro del constructor se realizan las inicializaciones correspondientes.

Ejemplo de constructor que recibe parámetros

```
class Persona
{    private string _strNombre;

    public Persona(string strNombre)
    {
        _strNombre = strNombre;
    }
}

class Programa
{
    static void Main()
    {
        Persona otraPersona = new Persona("Ramon");
        System.Console.WriteLine(otraPersona.Nombre);
        System.Console.ReadLine();
    }
}
```

Se debe proporcionar el valor de los parámetros al momento de la creación del objeto.

Ejemplo: Sobrecarga del constructor

```
class Persona
{
    private string _strNombre;
    public Persona()
    {
        _strNombre = "Desconocido";
    }
    public Persona(string strNombre)
    {
        _strNombre = strNombre;
    }
}

class Programa
{
    static void Main()
    {
        Persona unaPersona = new Persona();
        System.Console.WriteLine(unaPersona.Nombre);

        Persona otraPersona = new Persona("Ramon");
        System.Console.WriteLine(otraPersona.Nombre);
        System.Console.ReadLine();
    }
}
```

Constructores

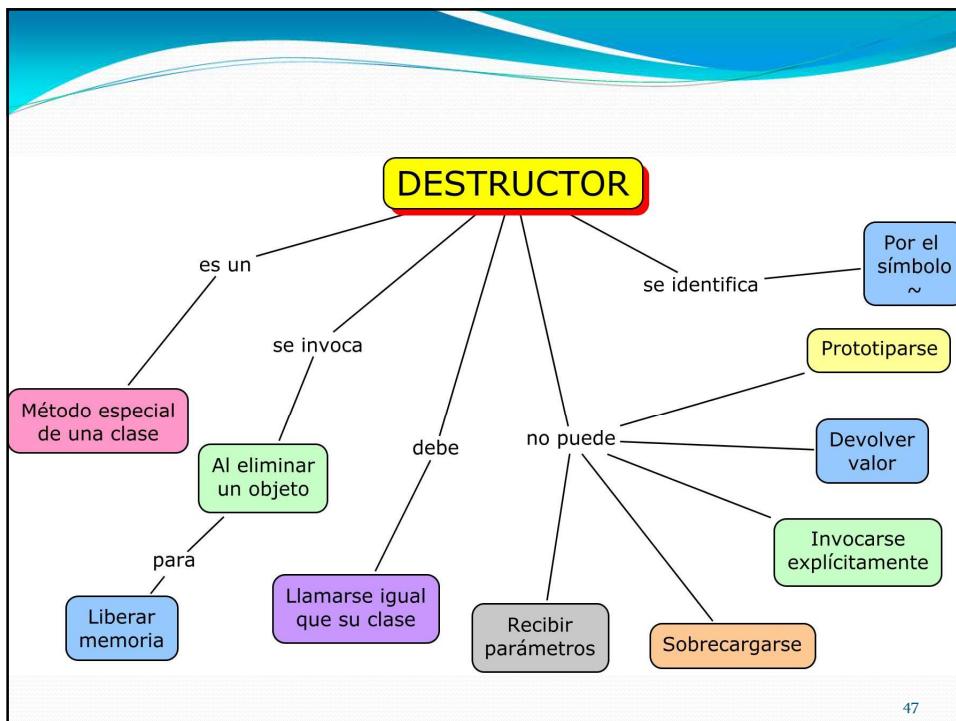
Dos constructores
= Dos diferentes
formas de
crear objetos

45

Método destructor

- No puede llamarse explícitamente.
- Se invoca automáticamente cuando el objeto es destruido.
- Invocado por el Garbage Collector (GC) justo antes de desasignar la memoria del objeto.
- Usado para cerrar archivos, conexiones de red, canales abiertos, etc.
- Sintaxis similar al método constructor, pero anteponiendo una tilde (~) al nombre.

46



Ejemplo de destructor

```
class Empleado
{
    ~Empleado()
    {
        System.Console.WriteLine("Se ha destruido el Empleado ");
    }
}
```

Un destructor NO puede sobrecargarse, y se ejecuta cuando el recolector de basura destruye el objeto.

48

Destrucción de objetos

- C# gestiona la memoria de modo automático mediante el “recolector de basura” (Garbage Collector o GC), quien se encarga de eliminar la memoria que en algún momento se solicitó y no se ocupa mas.
- En C# no existe el operador contrario a “new”.
- Por lo que, de manera general, NO es posible controlar exactamente el momento en el cual la memoria vuelve a estar disponible.

49

Destrucción de objetos

- Se puede indicar en el programa cuando un objeto deja de ser util, asignándole el valor “null”.

```
miObjeto = null;
```

- Cuando el GC lo note, tomará cartas en el asunto.
- El instante preciso en que lo hará, queda fuera del alcance del programador.

50

El Garbage Collector (recolector de basura)

- **Se asegura que:**

- Los objetos son destruidos una sola vez
- Se destruyen objetos que ya no son utilizados.

- **Trabaja cuando:**

- Hay poca memoria disponible
- La aplicación está finalizando
- El programador lo invoca manualmente (NO recomendado):

```
System.GC.Collect();
```

51

Consejos al diseñar clases y utilizar objetos

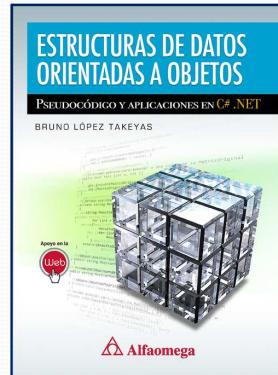
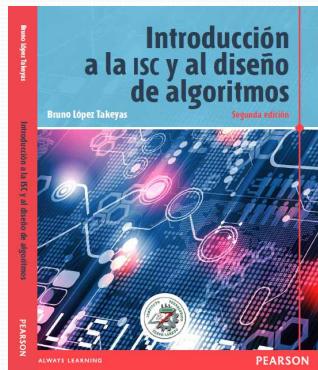
- No se deben capturar datos calculables
- Los datos calculables NO deben ser atributos
- Una vez creado el objeto, primero se deben introducir sus valores y después ejecutar sus métodos
- No se recomienda capturar ni imprimir datos desde un método ubicado dentro de una clase



52

Otros títulos del autor

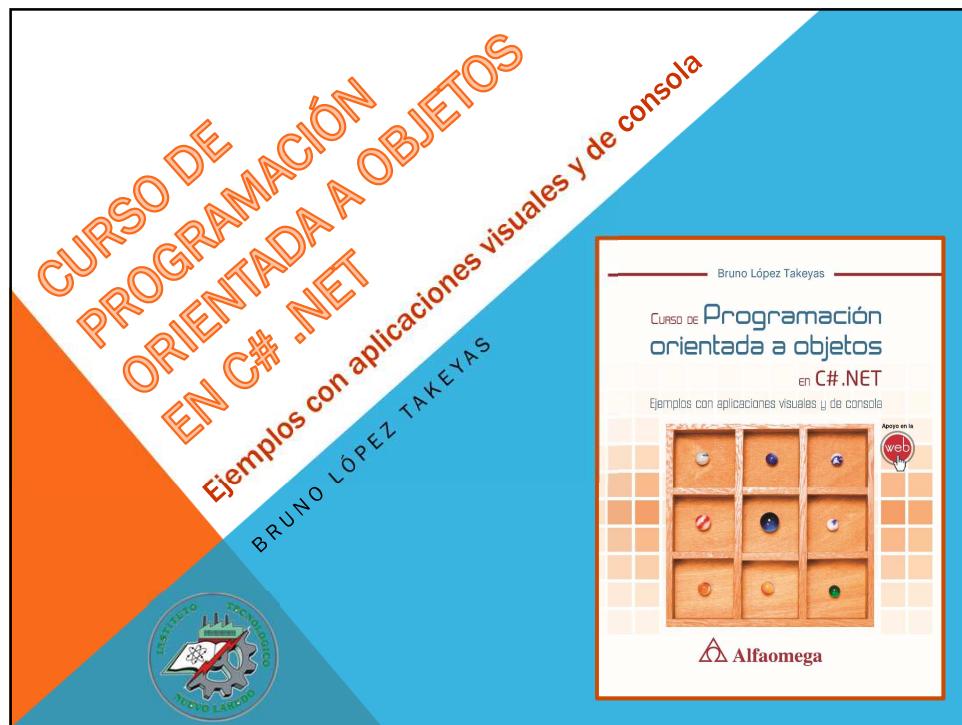
<http://www.itnuevolaredo.edu.mx/Takeyas/Libro>



bruno.lt@nlaredo.tecnm.mx



Bruno López Takeyas



Preguntas detonadoras



- ❑ ¿Qué es un método?
- ❑ ¿Cuáles son los tipos de métodos? ¿En qué se parecen?
¿En qué difieren?
- ❑ ¿Cómo se envían datos a los métodos?
- ❑ ¿Cuándo se recomienda enviar parámetros por valor?
¿Cuándo por referencia?
- ❑ ¿Por qué son importantes los métodos para los objetos?

3

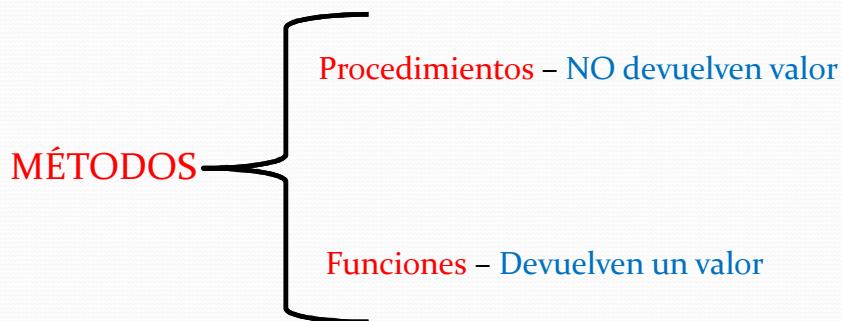
Métodos

- Contienen instrucciones para ejecutar al momento de ser invocados.
- Un método contiene:
 - Modificador de Acceso (Determina su visibilidad)
 - Tipo de dato (Devuelto al finalizar su ejecución)
 - Identificador (Nombre con el cual se invoca)
 - Parámetros (Cero o mas variables que recibe el método)

4

Métodos

- En C# las subrutinas se conocen como **métodos**, se codifican como parte de una clase y se clasifican en ...



5

Sintaxis de los métodos

```
< tipoValorDevuelto > < nombreMétodo > (< parámetros >)
{
    < cuerpo >
}

Ejemplo:
void Saludo( )
{
    Console.WriteLine("Hola");
}
```

A callout bubble points to the "void" keyword with the text: "void significa que NO devuelve valor (procedimiento)".

6

Ejemplo de un método (en la clase)

```
Modificador de acceso    Tipo de dato del valor regresado    Identificador    Parámetros
class Carro
{
    public void Encender()
    {
        System.Console.WriteLine("El Auto se ha
                                encendido!");
    }
}
```

7

Procedimientos

```
static void Imprimir()
{
    Console.WriteLine(Nombre);
    Console.WriteLine(Edad);
    Console.WriteLine(Sueldo);
}
```

8

Funciones

```
static int Sumar() // Devuelve un valor de tipo numérico entero  
  
static double Calcular() // Devuelve un valor de tipo numérico real  
static string Comparar() // Devuelve un valor de tipo cadena
```

```
static double CalcularArea()  
{  
    return(Math.PI * Math.Pow(Radio,2));  
}
```

9

Llamadas a los métodos

```
class Program  
{  
    static void Main(string[] args)  
    {  
        Método(); // Se invoca (llamada)  
    }  
  
    static void Método( )  
    {  
        . . . // Codificación  
    }  
}
```

10

Llamadas de procedimientos

```
class Program
{
    static void Main(string[] args)
    {
        Procedimiento(); // Llamada
    }

    static void Procedimiento()
    {
        Console.WriteLine("Tec Laredo");
        return(); // Fin del Procedimiento
    }
}
```

El
Procedimiento
NO devuelve
valor

11

Métodos que reciben parámetros

- Entre los paréntesis se especifican una o mas variables (separadas por comas) con sus respectivos tipos de datos.
- Esas variables estarán accesibles dentro del método.

```
public void CambiarEstado( string nuevoestado )
{
    estado = nuevoestado;
}
```



- Al momento de invocar el método, se deben incluir esos valores en la llamada:

```
miCarro.CambiarEstado("Apagado");
```



12

Parámetros recibidos por los métodos

Parámetros

Por valor – Se envía una copia del valor de la variable

Por referencia – Se envía la dirección de la variable

13

Envío de parámetros por valor

```
class Program
{
    static void Main(string[] args)
    {
        int x=10;
        Metodo( x ); // Se envia el valor de x
        Console.WriteLine("x="+x.ToString()); // x=10
    }
    static void Metodo(int y)
    {
        y+=5;
        Console.WriteLine("y="+y.ToString()); // y=15
    }
}
```

14

Envío de parámetros por referencia

```
static void Main(string[] args)
{
    int x = 10;
    Metodo(ref x); // Se envia la referencia de x
    Console.WriteLine("x=" + x); // x=15
    Console.ReadKey();
}
static void Metodo(ref int y)
{
    y += 5;
    Console.WriteLine("\n\ny=" + y); // y=15
}
```

15

Métodos que retornan valores

- El “Tipo de dato” del método NO es “void”.
- Dentro del método debe haber una sentencia “return” con algún valor del tipo de dato del método.
- Ejemplo (Al declararlo):

```
public string ConsultarEstado()
{
    return estado;
}
```



- Al llamar al método (desde el programa):

```
string estado_actual = miCarro.ConsultarEstado();
```



16

Llamadas de métodos que retornan valor (funciones)

```
static void Main(string[ ] args)
{
    double Radio = 10, Area;
    Area = Funcion(Radio);
    Console.WriteLine( "Area=" + Area );
    Console.ReadKey( );
}

static double Funcion(double r)
{
    return (Math.PI * r * r);
}
```

Variable receptor

Parámetro enviado a la función

Valor devuelto por la Función

17

Invocando al método (en el programa)

```
Carro miCarro = new Carro();
miCarro.Encender();
```

Nombre del objeto

Nombre del método

Parámetros

18

Invocando métodos

```
class Arbol
{
    int Altura;
    public void Podar( )
    {
        Console.WriteLine("Podando ...");
    }
}

Arbol Pino = new Arbol(); // Se crea el objeto Pino
Pino.Podar(); //Se invoca el método Podar() del
               objeto Pino
```

19

Ámbito de las variables

- El **ámbito de una variable** define dónde puede usarse esa variable
- Una variable local declarada en un bloque de programa, sólamente puede ser usada en ese bloque
- El **ámbito de una variable** también aplica a los métodos y a los ciclos

Ámbito de las variables

Ámbito de variables

Locales – Se declaran y utilizan dentro de un contexto específico. No se puede hacer referencia a ellas fuera de la sección de código donde se declara.

Globales – Se declaran fuera del cuerpo de cualquier método

21

Ámbito de variables en un ciclo for

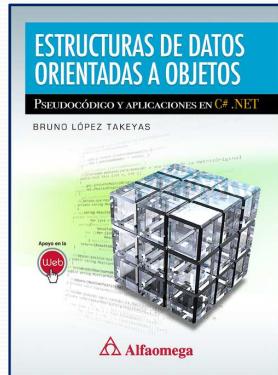
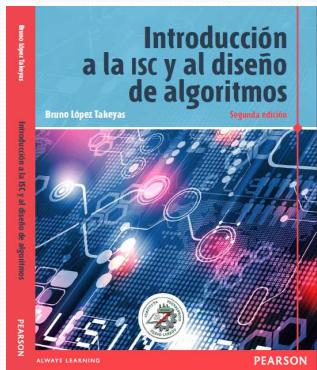
```
for(int x = 1; x<=10; x++)  
{  
    Console.WriteLine(x);  
}  
  
Console.WriteLine();
```

Microsoft Visual Studio
Errores al generar. ¿Desea continuar y ejecutar la última versión generada correctamente?
 No volver a mostrar este cuadro de diálogo
Sí No

Lista de errores
1 error 2 advertencias 0 mensajes
Descripción
3. El nombre 'x' no existe en el contexto actual
Lista de errores Resultados

Otros títulos del autor

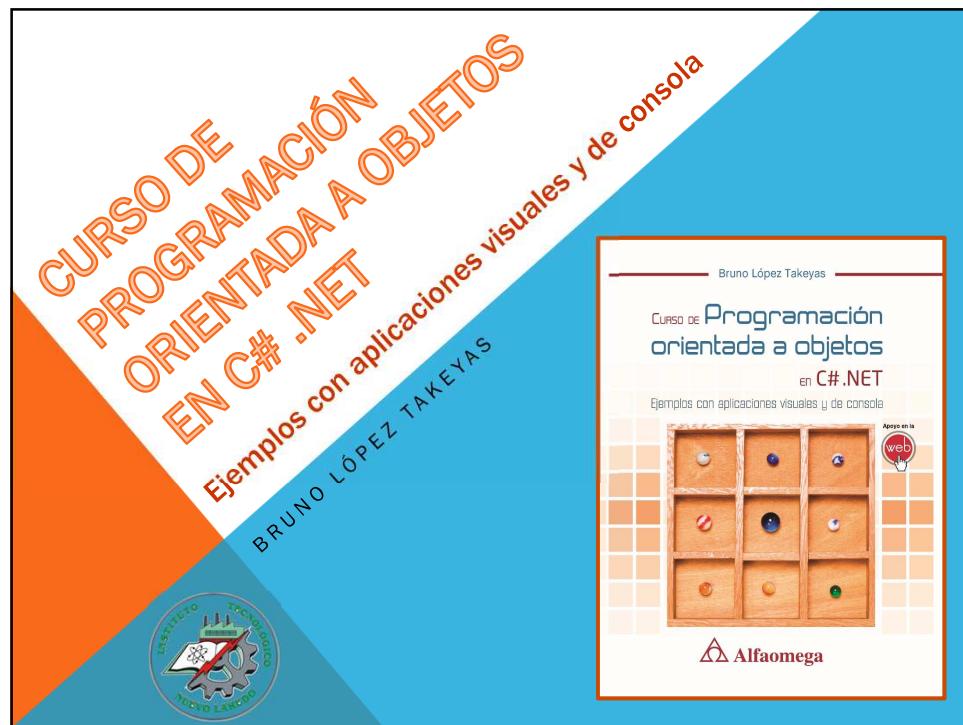
<http://www.itnuevolaredo.edu.mx/Takeyas/Libro>



bruno.lt@nlaredo.tecnm.mx



Bruno López Takeyas



Preguntas detonadoras



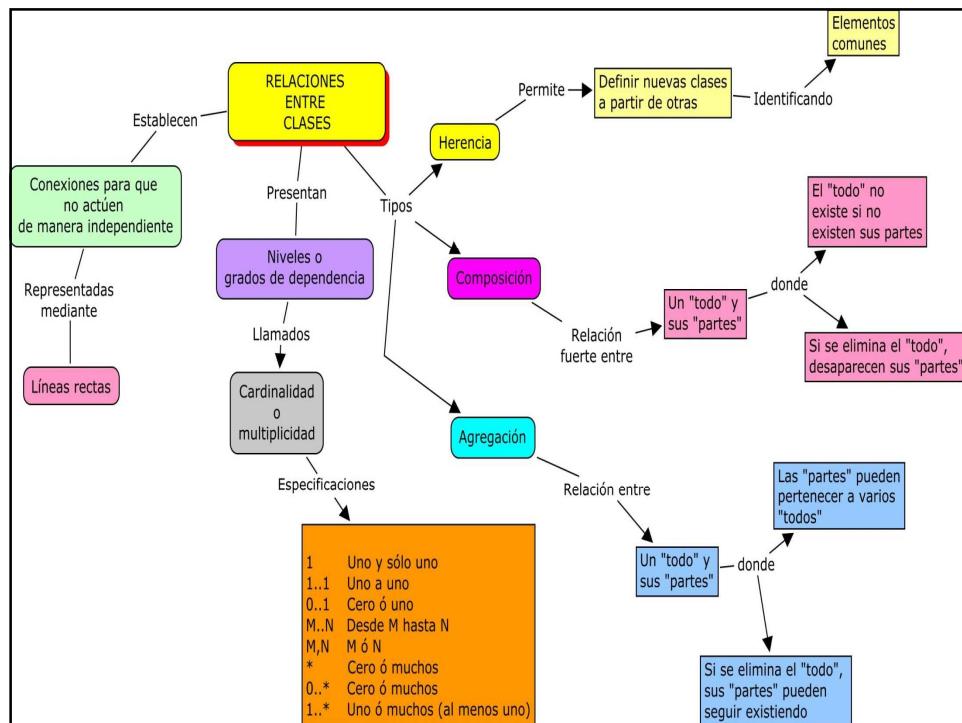
- ¿Qué ventajas ofrece la herencia a un programador?**
- ¿Cuál es la diferencia entre herencia simple y herencia múltiple?**
- Si una clase recibe herencia de una clase y varias interfaces, ¿se considera herencia múltiple?**
- Si una clase transmite (hereda) sus componentes a dos o más clases, ¿se considera herencia múltiple?**
- ¿Se pueden diseñar aplicaciones con herencia múltiple en C# .NET?**
- Si una clase abstracta no puede generar objetos, ¿entonces para qué sirve?**
- ¿Se puede modificar la implementación de un método heredado?**
- Si un miembro abstracto no tiene implementación, ¿entonces para qué sirve?**
- En una clase abstracta, ¿todos sus miembros son abstractos?**
- ¿Cuál es la ventaja de sobrescribir el método ToString()?**
- ¿Para qué sirve una clase sellada (sealed)?**
- ¿En qué se parece una interfase a una clase abstracta? ¿En qué difieren?**

3

Relaciones entre clases:

Herencia, Composición y Agregación

4



Herencia

- Característica de la POO que permite definir nuevas clases a partir de otras ya existentes.
- Las clases existentes “transmiten” sus características.

Herencia (cont.)

- Puede usarse para:
 - Relaciones del tipo “es un”
 - Ejemplo: Un Gerente “es un” Empleado con características propias adicionales.
- Objetivo: Reutilización de código.

7

Ejercicio

- Se deben modelar dos clases con las siguientes características:

Automovil	PalaMecanica
CaballosDeFuerza: int CantidadDePuertas: int	CaballosDeFuerza: int PesoMaximoDeLevante: int
Arrancar() : void Detener() : void Acelerar(int cuanto): void	Arrancar() : void Detener() : void MoverPala(string direccion) : void

8

Mal diseño (no recomendable)

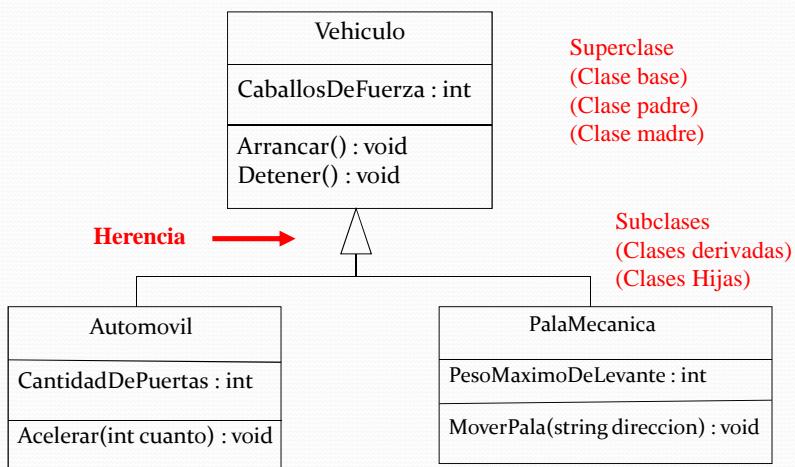
- Modelarlas de manera independiente.

```
class Automovil
{
    private int cf, cp;
    public int CaballosDeFuerza
    {
        get { return cf; }
        set { cf = value; }
    }
    public int CantidadDePuertas
    {
        get { return cp; }
        set { cp = value; }
    }
    public void Arrancar()
    {
    }
    public void Detener()
    {
    }
    public void Acelerar(int cuanto)
    {
    }
}

class PalaMecanica
{
    private int cf, pml;
    public int CaballosDeFuerza
    {
        get { return cf; }
        set { cf = value; }
    }
    public int PesoMaximoDeLevante
    {
        get { return pml; }
        set { pml = value; }
    }
    public void Arrancar()
    {
    }
    public void Detener()
    {
    }
    public void MoverPala(string direccion)
    {
    }
}
```

9

Diseño usando herencia (recomendado)



10

Definición de las clases usando herencia en C#

```

class Vehiculo
{
    private int cf;
    public int CaballosDeFuerza
    {
        get { return cf; }
        set { cf = value; }
    }
    public void Arrancar()
    {
    }
    public void Detener()
    {
    }
}

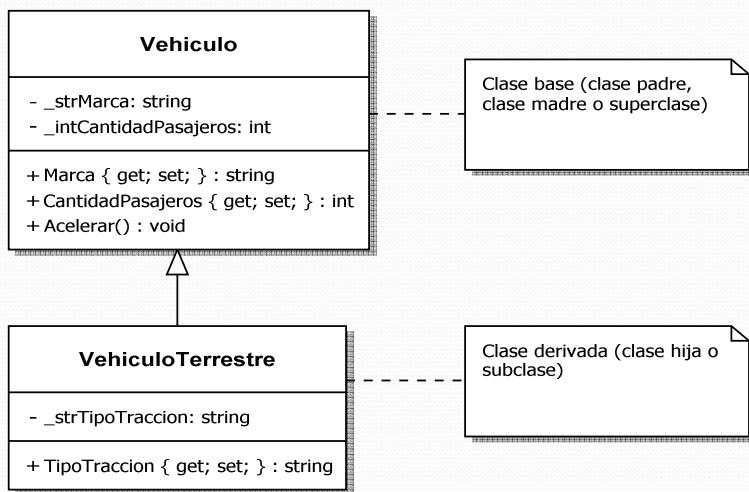
class Automovil : Vehiculo
{
    private int cp;
    public int CantidadDePuertas
    {
        get { return cp; }
        set { cp = value; }
    }
    public void Acelerar(int cuanto)
    {
    }
}

class PalaMecanica : Vehiculo
{
    private int pml;
    public int PesoMaximoDeLevante
    {
        get { return pml; }
        set { pml = value; }
    }
    public void MoverPala(string direccion)
    {
    }
}

```

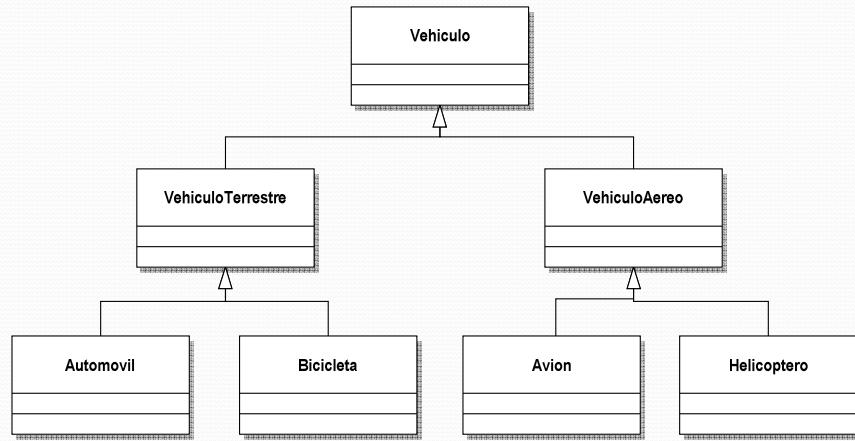
11

Ejemplo de herencia



12

Ejemplo de herencia con varios niveles



13

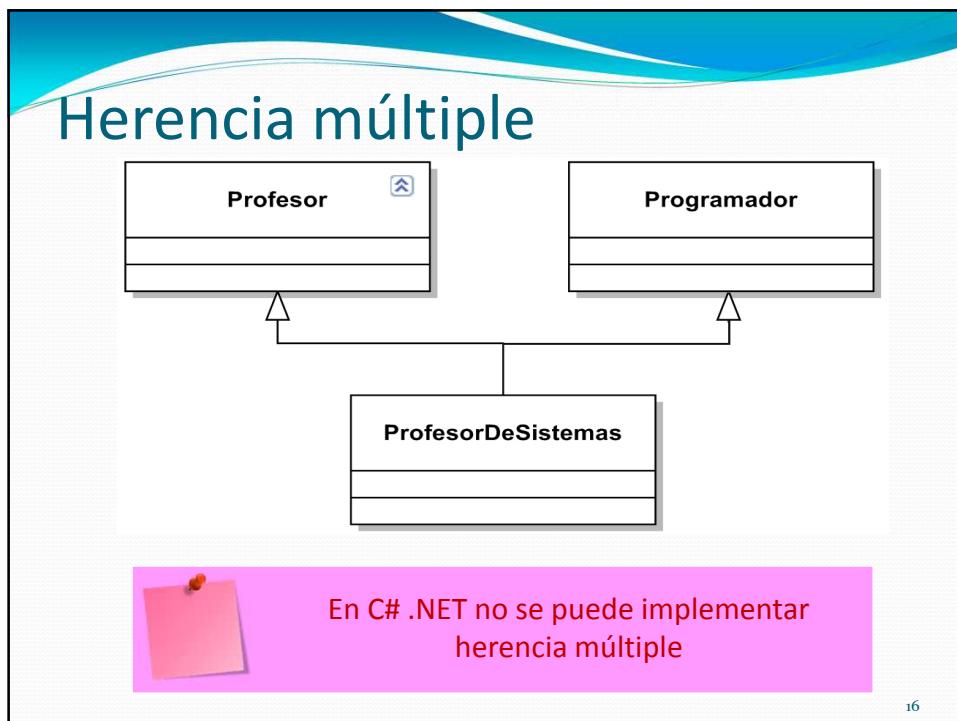
Tipos de herencia

Herencia

Simple

Múltiple

14



Herencia en C#

- En C# solo se permite Herencia simple.
- Ejemplo de Herencia en C#

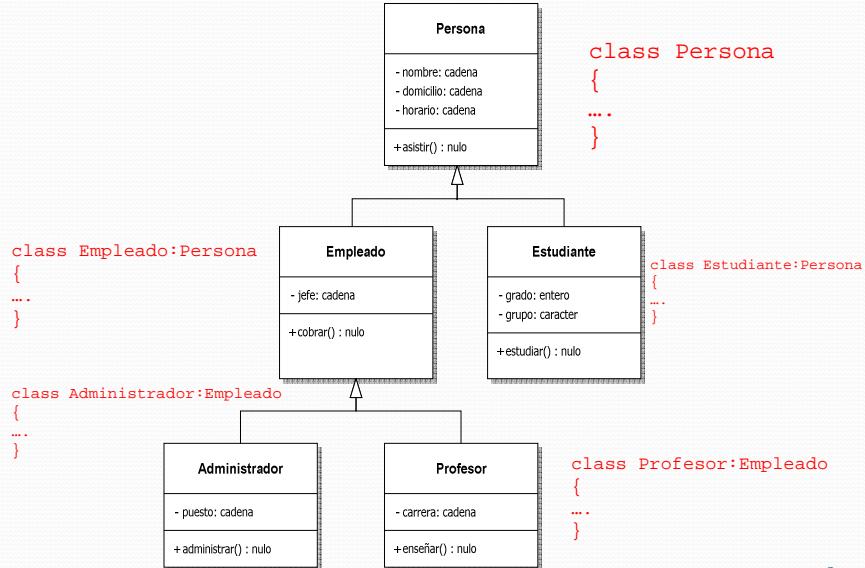
```
class A
{
}
class B : A
{
}
```

Indica que B “Hereda de” A

- Todos los objetos heredan de `System.Object`

17

Otro ejemplo de herencia



18

Uso de la Herencia

- **Beneficios:**

- Permite escribir menos código.
- Mejora la reusabilidad de los componentes.
- Facilita el mantenimiento del sistema completo.

- **Útil para un buen diseño del programa.**

- **Un diseño pobre sin herencia implementaría las clases involucradas de manera independiente.**

19

Ejercicio

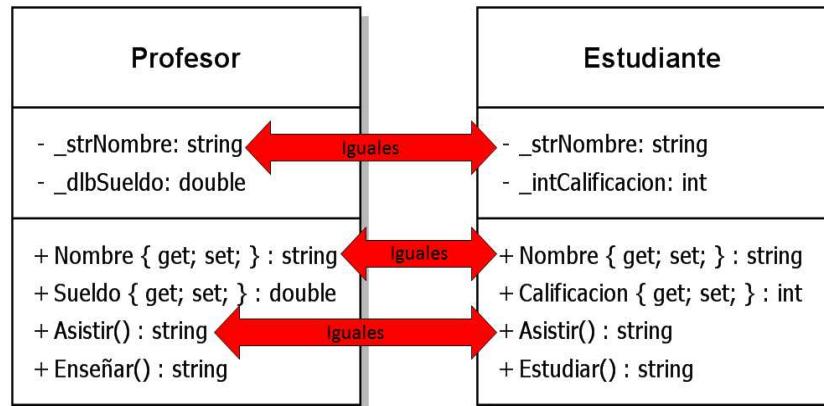
- Una escuela desea modelar los datos y las actividades de sus profesores y de sus estudiantes.

P R O F E S O R	
Datos	Actividades
Nombre (cadena)	Asistir a la escuela
Sueldo (numérico real)	Enseñar

E S T U D I A N T E	
Datos	Actividades
Nombre (cadena)	Asistir a la escuela
Calificación (numérico entero)	Estudiar

20

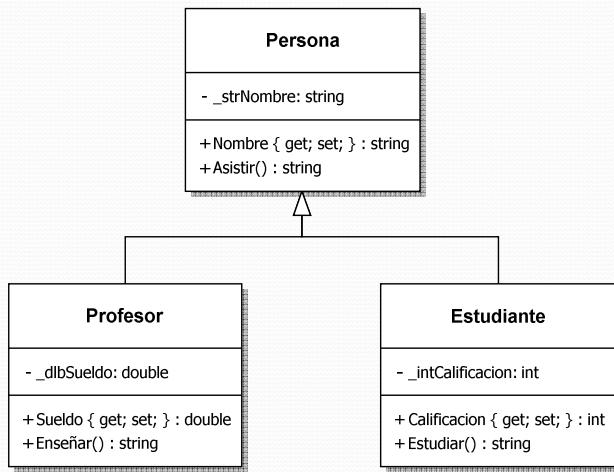
Mal diseño (no recomendable)



21

Diseño con herencia

- Prog. 5.1.- HerenciaConsola



22

Diseño con herencia

- El Prog. 5.2.- HerenciaForms utiliza el mismo diseño de herencia del proyecto de consola



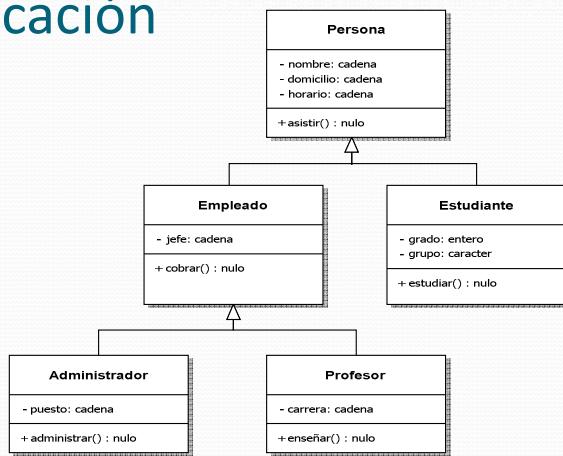
23

Salida del Prog. 5.2.- HerenciaForms



24

La herencia como estrategia de clasificación



Una clase base hereda todos sus componentes no privados y la clase derivada no puede elegir qué elementos recibir

25

Invocando un método de la clase base

- Una subclase puede llamar los métodos de su superclase con la palabra reservada “base”.

Se usa la palabra reservada “base” para invocar un método de una clase base desde una clase derivada, por lo tanto esta palabra NO puede usarse como el nombre de una variable

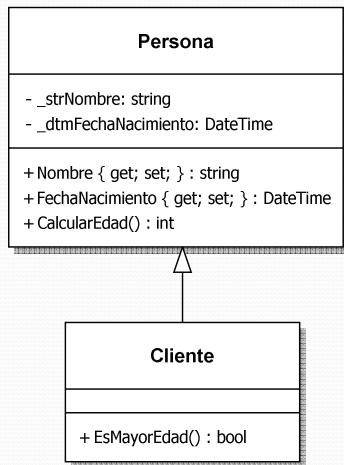
26

Ejercicio para invocar un método de una clase base desde una clase derivada

- Se desea determinar si un cliente es mayor de edad tomando como referencia su fecha de nacimiento. Para ello, se diseña un modelo orientado a objetos de una clase base **Persona** que define los datos nombre y la fecha de nacimiento de un individuo (con sus respectivas propiedades) y un método para determinar su edad (**CalcularEdad()**), que son heredados a una clase derivada identificada como **Cliente**.

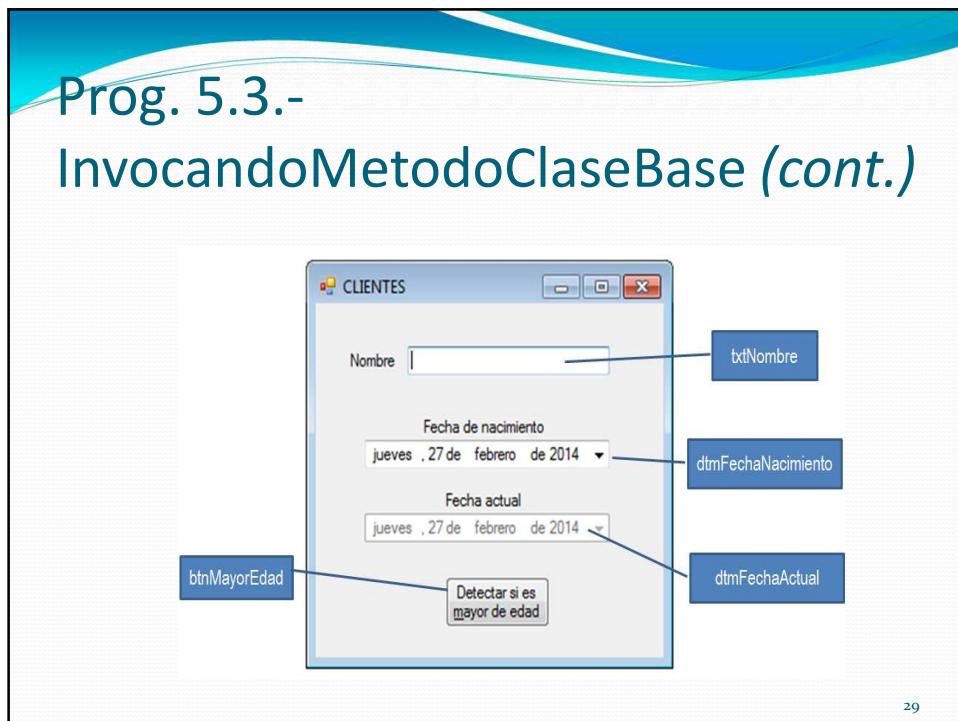
27

Prog. 5.3.- InvocandoMetodoClaseBase



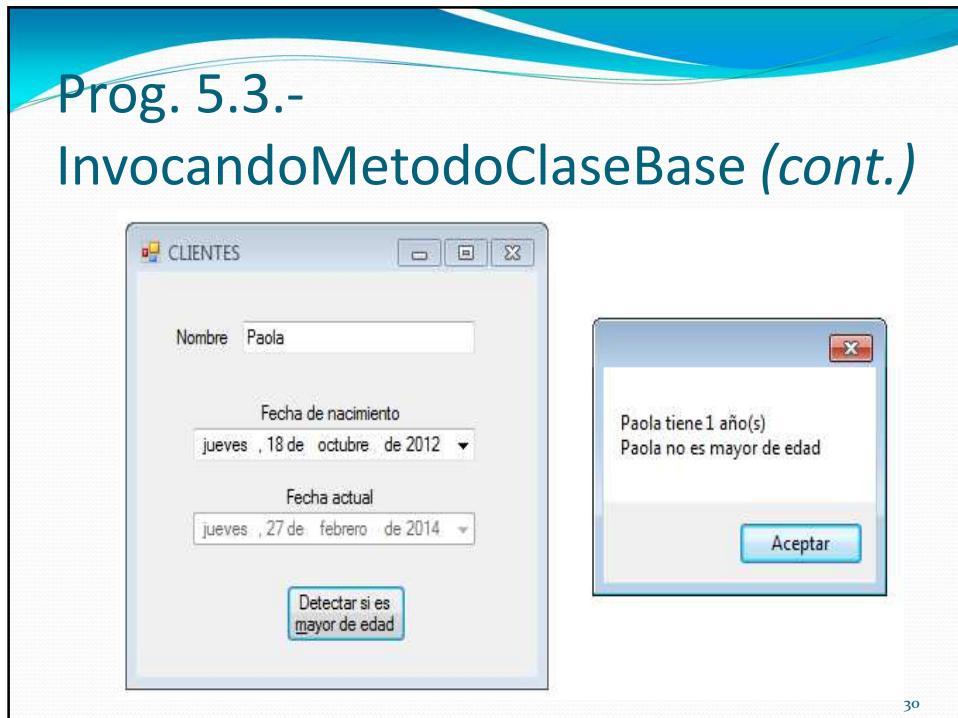
28

Prog. 5.3.- InvocandoMetodoClaseBase (cont.)



29

Prog. 5.3.- InvocandoMetodoClaseBase (cont.)



30

Codificación de la clase base

```
class Persona
{
    // Atributos privados
    private string _strNombre;
    private DateTime _dtmFechaNacimiento;

    // Propiedades públicas
    public string Nombre
    {
        get { return _strNombre; }
        set { _strNombre = value; }
    }

    public DateTime FechaNacimiento
    {
        get { return _dtmFechaNacimiento; }
        set { _dtmFechaNacimiento = value; }
    }

    // Método público para calcular la edad
    public int CalcularEdad()
    {
        int intEdad;
        TimeSpan intervalo;
        intervalo = DateTime.Now - this.FechaNacimiento;
        intEdad = (int)(intervalo.Days / 365.25);
        return (intEdad);
    }
}
```

31

Codificación de la clase derivada

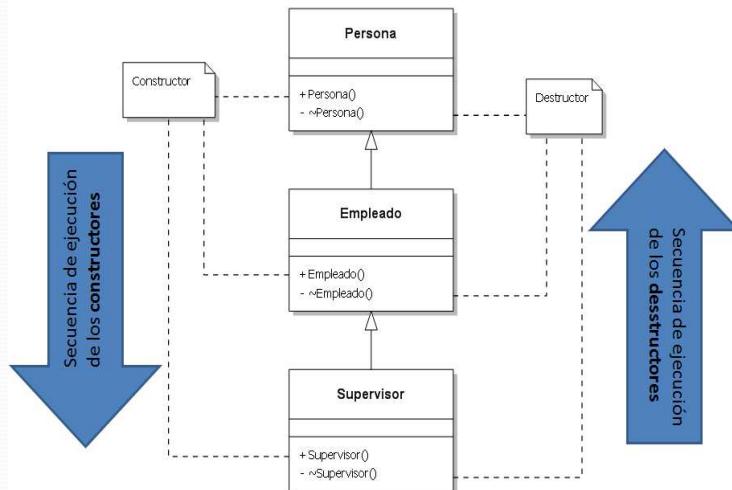
```
// La clase Cliente hereda de la clase Persona
class Cliente:Persona // Un cliente "es una" persona
{
    // Método público para determinar si es mayor de edad
    public bool EsMayorEdad()
    {
        // Variable local
        int intEdad;

        // Invoca el método CalcularEdad() de la clase base
        intEdad = base.CalcularEdad();

        if (intEdad >= 18)
            return (true);
        else
            return (false);
    }
}
```

32

Secuencia de ejecución de los constructores y destructores en la herencia



33

Invocando los constructores de la clase base

- También se puede invocar un constructor de la clase base desde el constructor de la clase derivada.
- Basta con definir el constructor de la clase derivada y colocar al final de su definición `:base(parámetros)`.
- Se puede invocar el constructor *default* (sin parámetros) o cualquier sobrecarga del constructor.

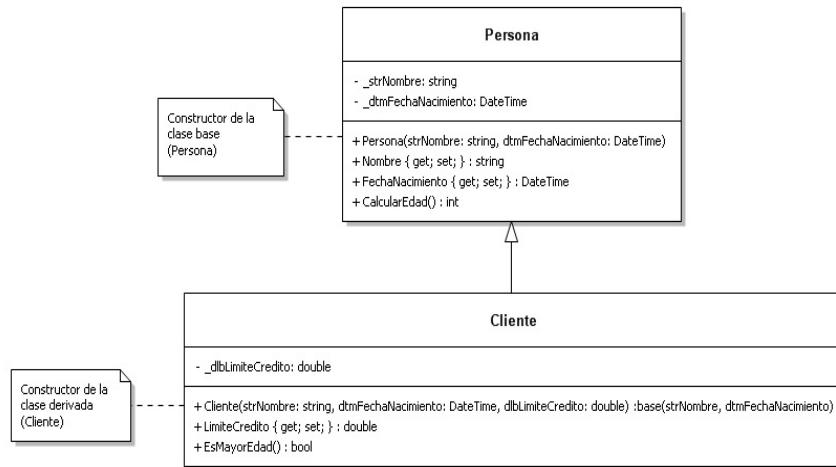
34

Ejercicio para invocar constructores de una clase base desde una clase derivada

- Una clase derivada llamada **Cliente** invoca el constructor de su clase base denominada **Persona**.

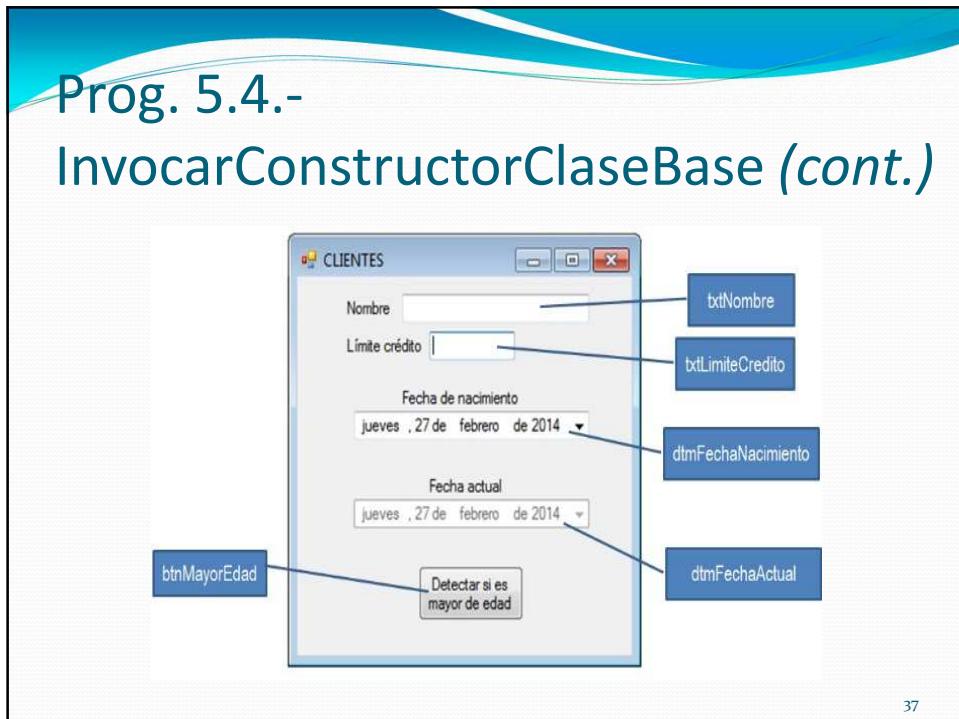
35

Prog. 5.4.- InvokerConstructorClaseBase



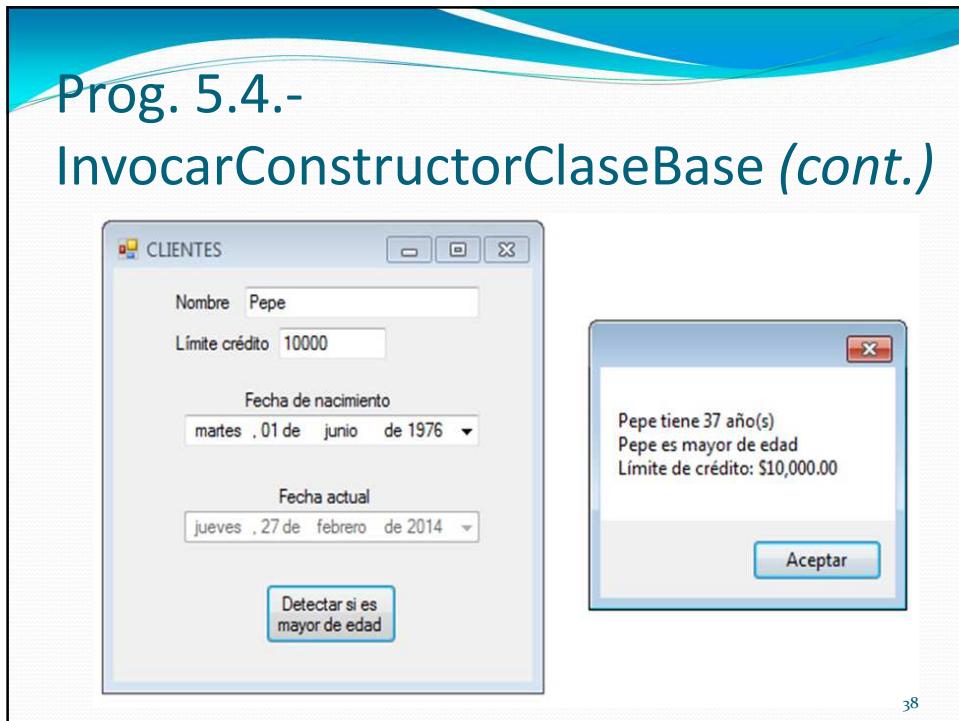
36

Prog. 5.4.- InvocarConstructorClaseBase (cont.)



37

Prog. 5.4.- InvocarConstructorClaseBase (cont.)



38

Sobrescritura del método ToString()

- El método `ToString()` está incluido en el framework .NET y se utiliza para convertir un dato a su representación de cadena (`string`).
- Todas las clases automáticamente heredan de la clase `System.Object`.
- Por lo tanto, el método `ToString()` puede ser sobrescrito (`override`) para ampliar su comportamiento y definir nuevas formas de desplegar datos.

39

Ejemplo

Empleado

- `_intNumero: int`
- `_strNombre: string`
- `_dblSueldo: double`

+ `Numero { get; set; } : int`
+ `Nombre { get; set; } : string`
+ `Sueldo { get; set; } : double`
+ `ToString() : string`

Se sobrescribe el método
`ToString()`

40

Implementación

```
class Empleado
{
    ...
    ...
    // Sobrescribir el método ToString()
    public override string ToString()
    {
        return ("Datos del empleado:\n\nNúmero: " + this.Numero + "\nNombre: "
+ this.Nombre + "\nSueldo: " + this.Sueldo.ToString("C"));
    }
}
```

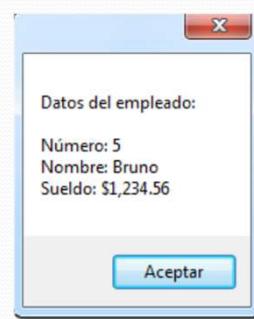
41

Salida

```
Empleado miEmpleado = new Empleado();
```

```
miEmpleado.Numero = 5;
miEmpleado.Nombre = "Bruno";
miEmpleado.Sueldo = 1234.56;
```

```
MessageBox.Show(miEmpleado);
```



En versiones de Microsoft Visual Studio anteriores a la 2015 es necesario codificar:

```
MessageBox.Show(miEmpleado.ToString());
```

42

Evitando la herencia: Clases selladas

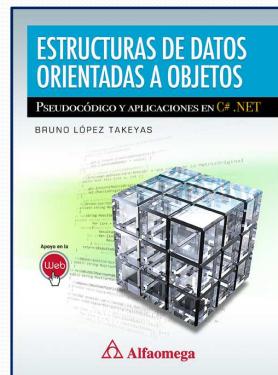
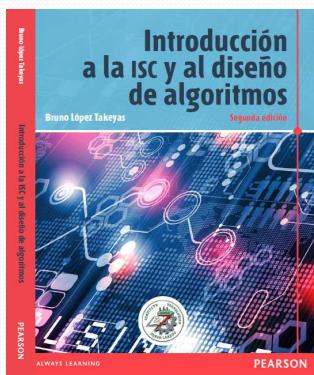
- Las clases selladas (**sealed**) pueden ser instanciadas pero NO heredadas.
- Se utiliza la palabra “**sealed**” para indicarlo.
- Usar “**sealed**” simultáneamente con “**abstract**” produce un error.

ERROR!

```
class Persona
{
    private string _strNombre;
    public string Nombre
    {
        get { return _strNombre; }
        set { _strNombre = value; }
    }
}
sealed class Empleado : Persona
{
    private string _strDepartamento;
    public string Departamento
    {
        get { return _strDepartamento; }
        set { _strDepartamento = value; }
    }
}
class EmpleadoTiempoParcial:Empleado
{
    private int _intHorasAsignadas;
    public int HorasAsignadas
    {
        get { return _intHorasAsignadas; }
        set { _intHorasAsignadas = value; }
    }
}
```

Otros títulos del autor

<http://www.itnuevolaredo.edu.mx/Takeyas/Libro>



bruno.lt@nlaredo.tecnm.mx



Bruno López Takeyas

IMPLEMENTACIÓN DE MÉTODOS EN C# .NET

Bruno López Takeyas

Instituto Tecnológico de Nuevo Laredo

Reforma Sur 2007, C.P. 88070, Nuevo Laredo, Tamps. México

<http://www.itnuevolaredo.edu.mx/takeyas>

E-mail: takeyas@itnuevolaredo.edu.mx

Resumen: El presente documento tiene como objetivo crear conciencia en los programadores en C# .NET sobre el aprovechamiento de los recursos que proporciona el lenguaje para mejorar el estilo de programación y facilitar la depuración y corrección de los programas. Para este efecto se mencionan temas de suma importancia: Uso de métodos (procedimientos y funciones), manejo de variables (locales y globales), el envío de parámetros o argumentos (por valor y por referencia) y la recepción de valores; de tal forma que sean de utilidad al implementar programas orientados a objetos.

1. Introducción

En los primeros paradigmas de programación de computadoras, las instrucciones se escribían y ejecutaban de manera secuencial o lineal; es decir, se codificaban las sentencias una después de la otra y seguían este patrón durante su ejecución. Sin embargo, este estilo provocaba programas muy extensos, poco legibles, mal organizados y por ende, complicados de depurar o corregir; a esto se le añade que en muchas ocasiones había necesidad de ejecutar un conjunto de instrucciones en varias ocasiones, lo cual provocaba escribirlo repetidamente en la codificación, ocasionando duplicidad de código y por ende más trabajo para el programador, ya que debía escribir varias veces el mismo código en el programa, revisarlo y provocando que los programas ocuparan más memoria y se tornaran difíciles de depurar.

Con el surgimiento del paradigma de la programación estructurada, se introduce la idea de organizar un programa de computadora en módulos, los cuales permiten organizarlo e identificar claramente la operación de los mismos. Cada módulo está identificado con un nombre, contiene

un conjunto de instrucciones que solamente se escriben una vez, pero pueden ser invocados las veces que sean necesarias; de tal forma, que ofrece al programador la facilidad de organizar sus programas de forma clara, precisa y fácil de depurar. En este paradigma de programación, estos módulos fueron conocidos con el nombre de subrutinas o subprogramas, los que actualmente en el paradigma de programación orientado a objetos se conocen con el nombre de métodos.

2. Definición de método

En la actualidad se conoce con el nombre de método a un conjunto de instrucciones que realiza una tarea específica y bien definida. Los métodos solamente se escriben una vez pero pueden ser invocados en múltiples ocasiones durante la ejecución de un programa. Esto le brinda al programador las siguientes ventajas:

- Facilita la separación de actividades en módulos debidamente identificados.
- Organiza de manera legible y fácil de entender a los programas.
- Facilita al programador la escritura de código.
- Facilita la depuración, corrección y mantenimiento de los programas.

Los métodos se clasifican en procedimientos y funciones (Fig. 1).

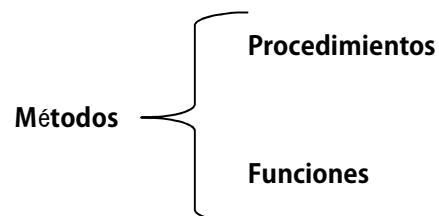


Fig. 1.- Tipos de métodos

En el paradigma orientado a objetos, los métodos representan las acciones realizadas por los objetos; por lo que se recomienda que se utilicen verbos para nombrarlos e identificarlos.

3. Procedimientos

Un procedimiento es un método que se compone de un conjunto de instrucciones para realizar un proceso, sin embargo, no devuelve algún resultado como producto de su operación; simplemente ejecuta las instrucciones que contiene sin informar el resultado obtenido. En C#, los procedimientos se identifican por su declaración de tipo `void`.

Por ejemplo, el siguiente método se encarga de imprimir en pantalla los datos de una persona; sin embargo, no devuelve valor alguno. Nótese que el método se declara de tipo `void` indicando que no devuelve valor; por lo tanto se trata de un procedimiento.

```
static void Imprimir()
{
    Console.WriteLine(Nombre);
    Console.WriteLine(Edad);
    Console.WriteLine(Sueldo);
}
```

La palabra `static` indica que el método es un miembro estático del programa; es decir, solamente se crea una vez cuando se ejecuta el programa y existe mientras se ejecute la aplicación (por el momento no se explica a detalle este concepto, el cual se trata en cursos de Programación Orientada a Objetos).

4. Funciones

Las funciones son métodos que ejecutan un conjunto de instrucciones e informan del resultado obtenido; es decir, devuelven el dato resultante de la ejecución. En C#, una función utiliza la sentencia `return()` para devolver el valor correspondiente. Enseguida se muestran algunos ejemplos de declaraciones de funciones:

```
static int Sumar() // Devuelve un valor
de tipo numérico entero
```

```
static double Calcular() // Devuelve un
valor de tipo numérico real
```

```
static string Comparar() // Devuelve un
valor de tipo cadena
```

Por ejemplo, el siguiente método calcula el área de una circunferencia aplicando la fórmula $A = \pi Radio^2$ y devuelve el resultado. Nótese que el método se declara de tipo `double`, indicando que devuelve un valor numérico real, por lo tanto se trata de una función.

```
static double CalcularArea()
{
    return(Math.PI*Math.Pow(Radio,2));
}
```

4.1.- Limitación de `return()`

Una limitante de una función es que la sentencia `return()` sólo devuelve un valor; esto restringe a que una función solamente pueda devolver un dato. Si se desea que la función devuelva más de un valor, entonces debe usarse otro mecanismo (por ejemplo el envío de parámetros por referencia ó el uso de parámetros de salida `out` en C# .NET).

5. Ámbito de las variables: Variables locales y globales

En el contexto de programación, se conoce como el ámbito a la disponibilidad que ofrece una variable dependiendo del lugar donde se declare. Las variables que se declaran dentro de un método o un bloque de sentencias se llaman variables locales mientras que las variables globales se conocen a través del programa entero y se pueden usar en cualquier segmento de código.

El valor de una variable local solamente se puede acceder dentro del segmento de código donde fue declarada dicha variable y no puede utilizarse en otra sección; en cambio una variable global puede accederse en cualquier parte del programa, manteniendo disponible su valor en todo momento.

Se pueden declarar variables globales declarándolas fuera de cualquier método (antes de `Main()`) y cualquier método puede acceder a ellas sin tener en cuenta en qué segmento de código esté dicha declaración.

6. Envío de parámetros a los métodos

Un método (procedimiento o función) puede recibir datos para realizar algunas acciones, los cuales se denominan parámetros o argumentos.

Existen dos formas de enviar datos a un método: por valor y por referencia (Fig. 2).

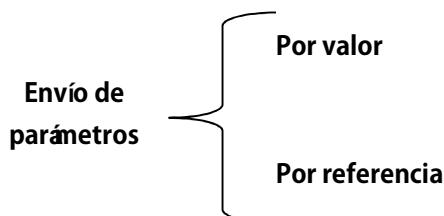


Fig. 2.- Tipos de envíos de parámetros

Cuando se invoca un método al que se le envía un parámetro por valor, se le manda una copia del valor de una variable o expresión, el cual es recibido por un parámetro declarado del mismo tipo de dato que el valor enviado. En cambio, si se le envía un parámetro por referencia, se le manda la referencia (dirección de memoria) de una variable.

Se pueden enviar varios datos a un método, sin embargo es necesario precisar que las variables receptoras deben estar declaradas en el orden indicado por el envío, también considerando la correspondencia con el tipo de dato.

Cuando se invoca un método se colocan entre paréntesis los parámetros enviados.

Es importante mencionar que los parámetros que reciben los datos enviados al método se consideran variables locales (independientemente si se hace por valor o por referencia) e incluso pueden tener el mismo nombre que la variable origen, sin embargo se trata de copias de los valores originales, por lo tanto, son variables diferentes.

```
Procesar(Nombre, Edad, Sueldo);
```

La declaración del método se muestra enseguida:

```
void Procesar(string N, int E, double S)  
{  
    .....  
}
```

En este caso, el parámetro **N** recibe una copia del valor de la variable **Nombre**, **E** recibe copia del valor de **Edad** y **S** recibe copia del valor de **Sueldo** (Fig. 3). Obsérvese que durante la llamada del método, se envían una cadena, un entero y un número real respectivamente, los cuales son recibidos en ese orden por los correspondientes parámetros; esto es, debe respetarse el orden de los tipos de datos enviados en la declaración de los parámetros.

Memoria RAM

Dirección	Valor	Variable
FA31:B278	“Pepe”	Nombre
...		
FA31:C45C	18	Edad
...		
FA31:D2A8	1500.50	Sueldo
...		
FA31:E6A1	“Pepe”	N
...		
FA31:E9A2	18	E
...		
FA31:F3A8	1500.50	S

Fig. 3.- Almacenamiento en memoria de los parámetros enviados por valor.

6.1.- Envío de parámetros por valor

Cuando se envía un parámetro por valor, se hace una copia del valor de la variable enviada en el parámetro recibido, el cual actúa como una variable local, que lo recibe, lo manipula dentro del método y que desaparece y pierde su valor al terminar la ejecución de ese segmento de código.

Por ejemplo, al invocar (llamar) el método identificado con el nombre **Procesar**, se colocan entre paréntesis los parámetros **Nombre**, **Edad** y **Sueldo** separados por comas:

Ahora se considera el siguiente ejemplo: Declarar una variable global denominada “x” y se inicializa con el valor 5 ($x=5$). Dentro del programa principal se declara y se inicializa una variable local llamada “y” con el valor 13 ($y=13$). Cuando se hace la llamada a un **Metodo(y)** y se envía la variable “y”, se hace por valor, es decir, se envía una copia del valor de la variable (13) que lo recibe otra variable local “a”. En ese momento, se transfiere el control de la ejecución del programa hacia el método, activando su variable

local “a” y desactivando momentáneamente la variable local del programa principal “y” (la variable “x” permanece activa ya que se trata de una variable global). Dentro del método, se modifica el valor de su variable local “a”, se imprime (16) y se duplica el valor de la variable global “x” y se imprime (10). Cuando el método termina, el sistema desactiva su variable local “a”, regresa el control de la ejecución del programa al lugar donde se hizo la llamada, activa y recupera el valor de su propia variable local ($y=13$) y continúa con su operación. Al imprimir los valores, nos percatamos que la variable “x” modificó su valor por tratarse de una variable global que puede ser accedida en cualquier parte del programa, sin embargo, la variable “y” mantiene su valor original (no fué alterado), ya que por tratarse de una variable local, fue desactivada al llamar al Método() y reactivada con su valor original al retornar. Enseguida se muestra el código en C# de este ejemplo (Prog. 1):

```
class Program
{
    static int x = 5; // Variable global

    static void Main(string[] args)
    {
        int y = 13; // Variable local

        Console.WriteLine("\nx=" + x);

        // Llamada al método y envío por valor
        Metodo(y);

        Console.WriteLine("\ny=" + y);

        Console.ReadKey();
    }

    // El parámetro "a" recibe el valor de "y"
    static void Metodo(int a)
    {
        a = a + 3;
        Console.WriteLine("\na=" + a);

        x = x * 2;
    }
}
```

Prog. 1.- Envío de parámetros por valor

Al ejecutar el programa anterior se produce la salida mostrada en la Fig. 4.

```
x=5
a=16
x=10
y=13
```

Fig.4.- Salida del programa de envío de parámetros por valor.

El Prog. 1 completo puede descargarse de:

[http://www.itnuevolaredo.edu.mx/takeyas
/libroED/Prog7-1.rar](http://www.itnuevolaredo.edu.mx/takeyas/libroED/Prog7-1.rar)

Para monitorear los valores de las variables e identificar su ámbito, se recomienda ejecutar paso a paso por instrucciones este programa en Microsoft Visual C# 2010 ó 2012 oprimiendo repetidamente la tecla F11.

6.2.- Envío de parámetros por referencia

Todas las variables se almacenan en celdas de la memoria RAM (*Random Access Memory* por sus siglas en inglés), las cuales están identificadas por una dirección expresada en números hexadecimales (p. ejem FA4D:32CE).

Cuando se envía un parámetro por referencia, no se hace una copia de su valor y quien lo recibe, no contiene directamente el dato, sino una referencia (dirección de memoria) a él.

El envío de parámetros por referencia permite a un método cambiar el valor del parámetro y mantener vigente dicho cambio. Cuando una variable es enviada por referencia, el método recibe la referencia de la variable original y esto implica que los cambios realizados a esa variable dentro del método, afectan la variable original.

Para ilustrarlo mejor, consideremos el ejemplo de la sección anterior, pero ahora enviando la variable “y” por referencia: Se declara una variable global denominada “x” y se inicializa con el valor 5 ($x=5$). Dentro del programa principal se declara y se inicializa la variable “y” con el valor 13 ($y=13$), la cual se considera variable local al ser declarada dentro de un método. Cuando se hace la llamada del Método(ref y) y se envía la variable “y”, se hace por referencia, es decir, se envía la dirección de memoria (no el valor) de la variable “y”, que lo recibe

la variable local “a”. Dentro del método, se modifica el valor de la variable “a”, se imprime (16) y se duplica el valor de la variable global “x” (10) y se imprime. Cuando el método termina, el sistema desactiva su variable local “a”, regresa el control de la ejecución del programa al lugar donde se hizo la llamada, activa y recupera los valores de su propia variable local ($y=13$) y continúa con su operación. Al imprimir los valores, nos percatamos que la variable “x” modificó su valor por tratarse de una variable global que puede ser accedida en cualquier parte del programa y la variable “y”, aunque se trata de una variable local, también modificó su valor, ya que siendo la variable “a” una referencia de la variable “y”, entonces al modificar “a” también se modifica “y”. Esto se debe a que la variable “a” no recibe una copia del valor de la variable “y” sino la dirección de memoria donde está almacenado dicho valor (FA31:C45C). A este concepto se le conocía con el nombre de apuntador en lenguajes como Pascal, C y C++, ya que la variable “a” apunta al valor almacenado en la variable “y” (Fig. 5).

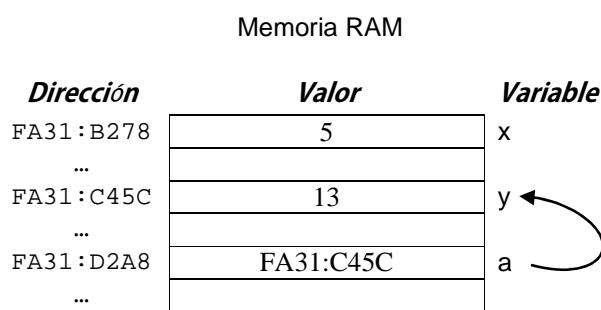


Fig. 5.- Almacenamiento en memoria de los parámetros enviados por referencia.

El siguiente código en C# ilustra este ejemplo (Prog. 2):

```
class Program
{
    static int x = 5; // Variable global

    static void Main(string[] args)
    {
        int y = 13; // Variable local

        Console.WriteLine("\nx=" + x);
```

```
// Llamada al método y envío por referencia
Metodo(ref y);

Console.WriteLine("\nx=" + x);

Console.WriteLine("\ny=" + y);

Console.ReadKey();
}

// El parámetro "a" recibe la ref. de "y"
static void Metodo(ref int a)
{
    a = a + 3;
    Console.WriteLine("\na=" + a);

    x = x * 2;
}
```

Prog. 2.- Envío de parámetros por referencia.

Al ejecutar el programa anterior se produce la salida mostrada en la Fig. 6.

```
x=5
a=16
x=10
y=16
```

Fig. 6.- Salida del programa de envío de parámetros por referencia.

El Prog. 2 completo puede descargarse de:

<http://www.itnuevolaredo.edu.mx/takeyas/libroED/Prog7-2.rar>

Para monitorear los valores de las variables e identificar su ámbito, se recomienda ejecutar paso a paso por instrucciones este programa en Microsoft Visual C# 2010 ó 2012 oprimiendo repetidamente la tecla F11.

6.3.- Parámetros de salida out en C# .NET

Un parámetro de salida en C# (`out`) es muy similar a un parámetro por referencia (`ref`), excepto que los parámetros por referencia deben ser inicializados antes de enviarse; sin embargo, el método debe asignarle un valor antes de devolverlo. Para utilizar un parámetro de salida, basta con anteponer la palabra `out` tanto en el parámetro enviado como en su declaración en el método.

Este tipo de parámetros son útiles cuando se requiere que una función devuelva más de un dato, ya que por definición, existe una restricción de que una función solamente devuelve un valor.

Para ilustrar mejor el uso de un parámetro de salida, tomaremos como ejemplo el programa anterior (Prog. 2) al que se le agrega una variable booleana como parámetro de salida para determinar si el parámetro enviado por referencia es par ó impar. Este código en C# muestra este ejemplo (Prog. 3):

```
class Program
{
    static int x = 5; // Variable global

    static void Main(string[] args)
    {
        int y = 13; // Variable local
        bool esImpar;

        Console.WriteLine("\nx=" + x);

        // Llamada al método y envío por referencia
        Metodo(ref y, out esImpar);

        Console.WriteLine("\nx=" + x);

        Console.WriteLine("\ny=" + y);

        if (esImpar)
            Console.WriteLine("\ny es un número
impar");
        else
            Console.WriteLine("\ny es un número
par");

        Console.ReadKey();
    }

    // El parámetro "a" recibe la referencia de
    // "y" y el parámetro de salida sirve para
    // determinar si el parámetro enviado es Impar
    static void Metodo(ref int a, out bool Impar)
    {
        a = a + 3;
        Console.WriteLine("\na=" + a);
    }
}
```

```
x = x * 2;

if (a % 2 != 0)
    Impar = true;
else
    Impar = false;
}
```

Prog.3.- Parámetro de salida en C#.

Al ejecutar el programa anterior se produce la salida mostrada en la Fig. 7.

```
x=5
a=16
x=10
y=16
y es un número par
```

Fig. 7.- Salida del programa de uso de parámetro de salida.

El Prog. 3 completo puede descargarse de:

<http://www.itnuevolaredo.edu.mx/takeyas/libroED/Prog7-3.rar>

Para monitorear los valores de las variables e identificar su ámbito, se recomienda ejecutar paso a paso por instrucciones este programa en Microsoft Visual C# 2010 ó 2012 oprimiendo repetidamente la tecla F11.

7. Recibiendo un valor de una función

Una vez que se invoca una función es necesario utilizar la variable capaz de recibir el valor devuelto por ésta, la cual debe ser del mismo tipo de la función. Por ejemplo, si se invoca el siguiente método

```
x = Procesar(a, b);
```

la variable “x” recibe el valor calculado por la función `Procesar()`, que acepta los parámetros “a” y “b” respectivamente.

Para ilustrarlo mejor, se considera el siguiente ejemplo: Se desea implementar una función que calcule el área de una circunferencia; para ello, la función recibe como parámetro el valor del radio, aplica la fórmula correspondiente y devuelve el resultado calculado. El programa principal solicita al usuario capturar el valor del radio de una circunferencia y almacenarlo en la variable local llamada “Radio”, la cual es enviada como parámetro por valor a la función `CalcularArea()`, que recibe el parámetro en la variable local llamada “r” aplica la fórmula $Area = \pi r^2$ y devuelve el resultado calculado al programa principal quien lo recibe en la variable “Area”. El siguiente código en C# ilustra este ejemplo (Prog. 4):

```
class Program
{
    static void Main(string[] args)
    {
        double Radio, Area;
        Console.WriteLine("Teclee el valor del radio:");
        Radio = double.Parse(Console.ReadLine());

        // La variable Area recibe el valor devuelto
        // por la función
        Area = CalcularArea(Radio);

        Console.WriteLine("Área = " + Area);
        Console.ReadKey();
    }

    static double CalcularArea(double r)
    {
        return (Math.PI * Math.Pow(r, 2));
    }
}
```

Prog. 4.- Función para calcular el área de una circunferencia

Al ejecutar el programa anterior se produce la salida mostrada en la Fig. 8.

```
Teclee el valor del radio: 2.3
Área = 16.61902513749
```

Fig. 8.- Salida del programa de uso de una función.

El Prog. 4 completo puede descargarse de:

http://www.itnuevolaredo.edu.mx/takeyas/_repasoFP/Prog4.rar

Para monitorear los valores de las variables e identificar su ámbito, se recomienda ejecutar paso a paso por instrucciones este programa en Microsoft Visual C# 2010 ó 2012 oprimiendo repetidamente la tecla F11.

8. Aplicaciones prácticas del uso de métodos

Existen numerosas aplicaciones prácticas del uso de métodos para el desarrollo de sistemas computacionales. La implementación de métodos ayuda a organizar mejor el diseño de programas y facilitan su mantenimiento.

El Prog. 5 muestra una aplicación típica de un método al que se le envían parámetros: el ordenamiento de los datos de un arreglo. El programa principal solicita al usuario que capture la cantidad de celdas de un arreglo unidimensional y lo almacena en la variable local llamada “Tamaño”, crea el arreglo local y lo llena con números enteros generados de manera aleatoria. Después invoca al procedimiento llamado `Ordenar()` y le envía como parámetros el “Arreglo” completo y su “Tamaño” (los cuales son recibidos por el arreglo “A” y la variable “T” respectivamente) para implementar un algoritmo que ordena de manera ascendente (de menor a mayor) los datos almacenados en el arreglo (en este momento no es importante explicar el método de ordenamiento). Obsérvese que se trata de un procedimiento, puesto que no devuelve valor; sin embargo, no es necesario devolver valor alguno, ya que cuando se envía un arreglo como parámetro a un método, automáticamente se hace por referencia; es decir, no se requiere implementar una función que devuelva el arreglo con los datos ordenados; sino que al hacer los cambios en el arreglo recibido dentro del método (arreglo “A”), éstos se reflejan de manera inmediata en el arreglo original (Arreglo) que fue generado en el programa principal. El siguiente código en C# muestra este ejemplo:

```
class Program
{
    static void Main(string[] args)
    {
```

```

// Declaración del tamaño del arreglo
int Tamaño;

// Declaración del arreglo
int [] Arreglo;

// Generar un número aleatorio
Random NumeroAleatorio = new
Random(int.MaxValue);

Console.WriteLine("Teclee el tamaño del arreglo:");
Tamaño = int.Parse(Console.ReadLine());

// Creación del arreglo
Arreglo = new int[Tamaño];

// Llena el Arreglo con números generados
aleatoriamente
Console.WriteLine("\nARREGLO
DESORDENADO\n");

for (int i = 0; i < Tamaño; i++)
{
    // Genera núm. aleatorio y lo almacena en
    la celda "i" del Arreglo
    Arreglo[i] = NumeroAleatorio.Next();

    // Imprime el núm. generado
    Console.WriteLine("{0:N0} ",Arreglo[i]);
}

Console.Write("\nOprima cualquier tecla para
ordenar el arreglo ...");
Console.ReadKey();

// Invoca el procedimiento "Ordenar" y le
envía el "Arreglo" y su "Tamaño"
Ordenar(Arreglo, Tamaño);

// Limpia la pantalla
Console.Clear();

// Imprime el arreglo
Console.WriteLine("ARREGLO ORDENADO\n");

for(int i=0; i<Tamaño; i++)
    Console.WriteLine("{0:N0} ", Arreglo[i]);

Console.Write("\nOprima cualquier tecla para
salir ...");
Console.ReadKey();
}

// Procedimiento para ordenar un arreglo
// recibido como parámetro
static void Ordenar(int[] A, int T)
{

```

```

for(int p=0; p<T-1; p++)
    for(int e=p+1; e<T; e++)
        if (A[e] < A[p])
        {
            int temp=A[p];
            A[p] = A[e];
            A[e] = temp;
        }
}

```

Prog. 5.- Procedimiento para ordenar los datos de un arreglo

Al ejecutar el programa anterior se produce la salida mostrada en la Fig. 9.

Teclee el tamaño del arreglo: 6

ARREGLO DESORDENADO

```

1,559,595,546
1,755,192,844
1,649,316,172
1,198,642,031
442,452,829
1,200,195,955

```

Oprima cualquier tecla para ordenar el arreglo ...

ARREGLO ORDENADO

```

442,452,829
1,198,642,031
1,200,195,955
1,559,595,546
1,649,316,172
1,755,192,844

```

Oprima cualquier tecla para salir

Fig. 9. Salida del programa de ordenamiento de un arreglo

El Prog. 5 completo puede descargarse de:

<http://www.itnuevolaredo.edu.mx/takeyas/repassoFP/Prog5.rar>

9. Conclusiones

El uso de métodos produce sistemas organizados en módulos que le facilitan al programador su legibilidad, depuración y mantenimiento, dando como consecuencia menor esfuerzo de diseño y programación; razón por la que es muy importante concientizar a estudiantes que vayan a cursar la materia de Programación Orientada a Objetos (POO) sobre la importancia de dominar el uso de métodos (procedimientos, funciones), envío de parámetros y recepción del valor de una función, ya que son un componente fundamental de los objetos porque representan sus acciones realizadas. Desde el inicio del curso de POO, se realiza diseño orientado a objetos en el que los métodos son la parte medular del comportamiento de los objetos, el cual, posteriormente se transforma en sistemas codificados en un lenguaje de programación como C# .NET.

10. Bibliografía

- Archer, Tom. “**A fondo C#**”. Editorial McGraw Hill. 2001.
- Ceballos, Francisco Javier. “**Enciclopedia de Microsoft Visual C#**”. Editorial Alfa Omega. 2010.
- Ceballos, Francisco Javier. “**Microsoft C#. Curso de Programación**”. Editorial Alfaomega. 2008.
- Ceballos, Francisco Javier. “**Microsoft C#. Lenguaje y aplicaciones**”. Editorial Alfaomega. 2008.
- Deitel & Deitel. “**Programming in C#**”. Editorial Prentice Hall.
- Ferguson, Jeff. “**La Biblia de C#**”. Editorial Anaya.2003.
- López Takeyas, Bruno. “**Curso de Programación en C#**”. Filminas. Instituto Tecnológico de Nuevo Laredo. Consultado el 10 de diciembre de 2012 de <http://www.itnuevolaredo.edu.mx/takeyas/Apuntes/POO/index.htm>
- López Takeyas, Bruno. “**Estructuras de datos orientadas a objetos. Pseudocódigo y aplicaciones en C# .NET**”. Editorial Alfaomega. México. 2012.
- Miles, Rob. “**C# Development**”. Department or Computer Science. University of Hull. 2008-2009.
- Roque Hernández, Ramón. “**POO (Ingeniería – Tecnológico de Nuevo Laredo)**”. Filminas. Instituto Tecnológico de Nuevo Laredo. Consultado el 10 de diciembre de 2012 de <http://ramonroque.com/Materias/pooTec.htm>.

El autor



Bruno López Takeyas se tituló de Ingeniero en Sistemas Computacionales en el Instituto Tecnológico de Nuevo Laredo en 1993. Obtuvo el grado de Maestro en Ciencias de la Administración con especialidad en Sistemas en la Universidad Autónoma de Nuevo León en marzo del 2000.

Desde 1994 es profesor del Depto. de Sistemas y Computación del Instituto Tecnológico de Nuevo Laredo. También es profesor invitado de varias universidades públicas y privadas en sus programas de nivel maestría.

Ha impartido varias conferencias relacionadas con sistemas computacionales, las más recientes en el Instituto Tecnológico de Cancún, Instituto Tecnológico de Piedras Negras, Universidad Autónoma de Tamaulipas y para la Universidad Técnica de Machala, Ecuador. Es autor de los libros “Introducción a la ISC y al diseño de algoritmos” y “Estructuras de datos orientadas a objetos. Pseudocódigo y aplicaciones en C# .NET”.

Contacto:

Email: takeyas@itnuevolaredo.edu.mx

Web: <http://www.itnuevolaredo.edu.mx/takeyas>