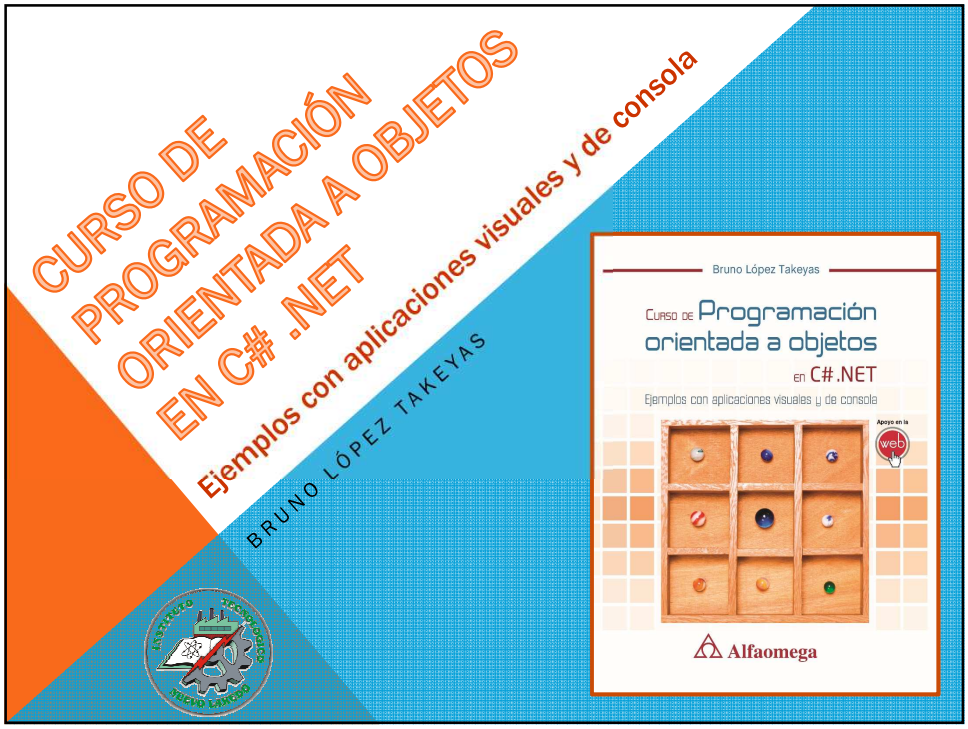
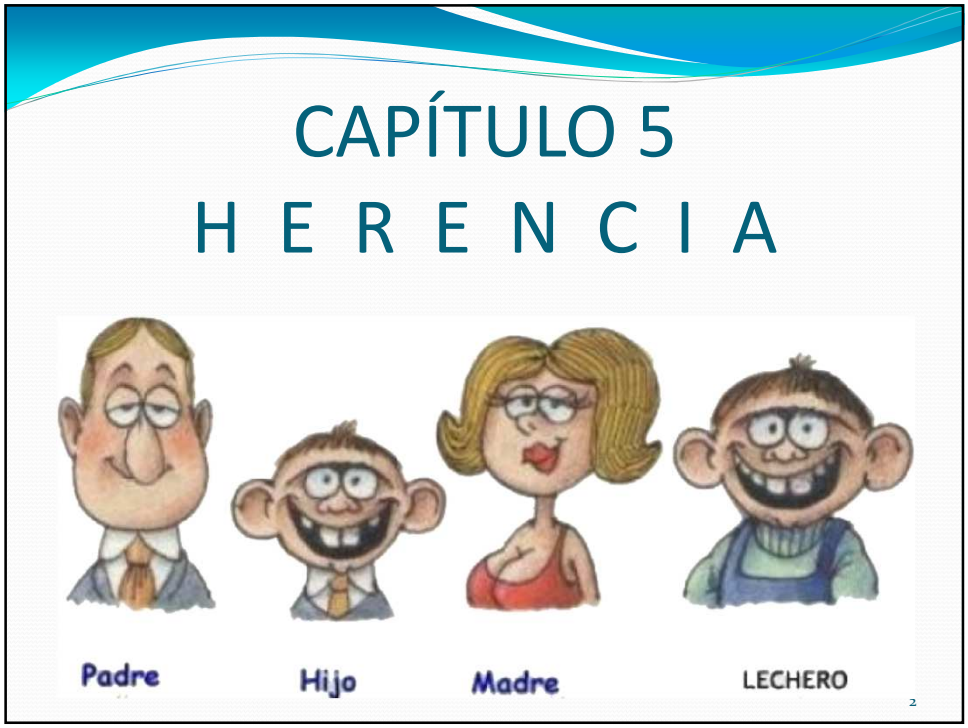

CURSO DE PROGRAMACIÓN ORIENTADA A OBJETOS EN C# .NET
Ejemplos con aplicaciones visuales y de consola
BRUNO LÓPEZ TAKEYAS
Bruno López Takeyas
Curso de Programación orientada a objetos en C#.NET
Ejemplos con aplicaciones visuales y de consola
Apoyo en la web
Alfaomega


CAPÍTULO 5
H E R E N C I A
Padre
Hijo
Madre
LECHERO
2

# Preguntas detonadoras

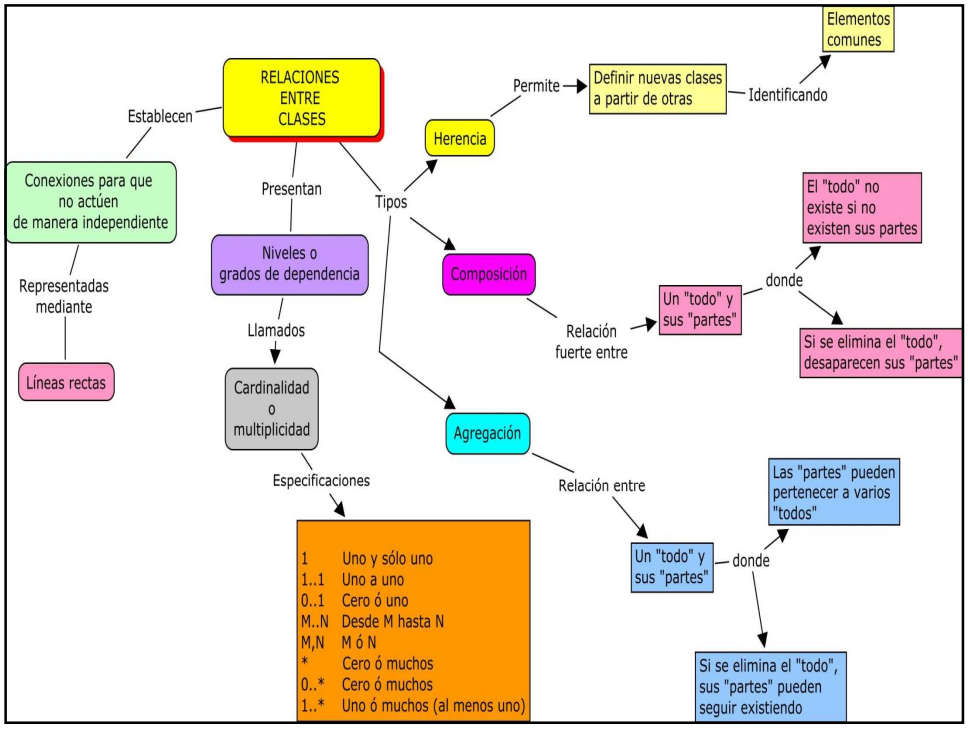
- ¿Qué ventajas ofrece la herencia a un programador?
- ¿Cuál es la diferencia entre herencia simple y herencia múltiple?
- Si una clase recibe herencia de una clase y varias interfaces, ¿se considera herencia múltiple?
- Si una clase transmite (hereda) sus componentes a dos o más clases, ¿se considera 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?

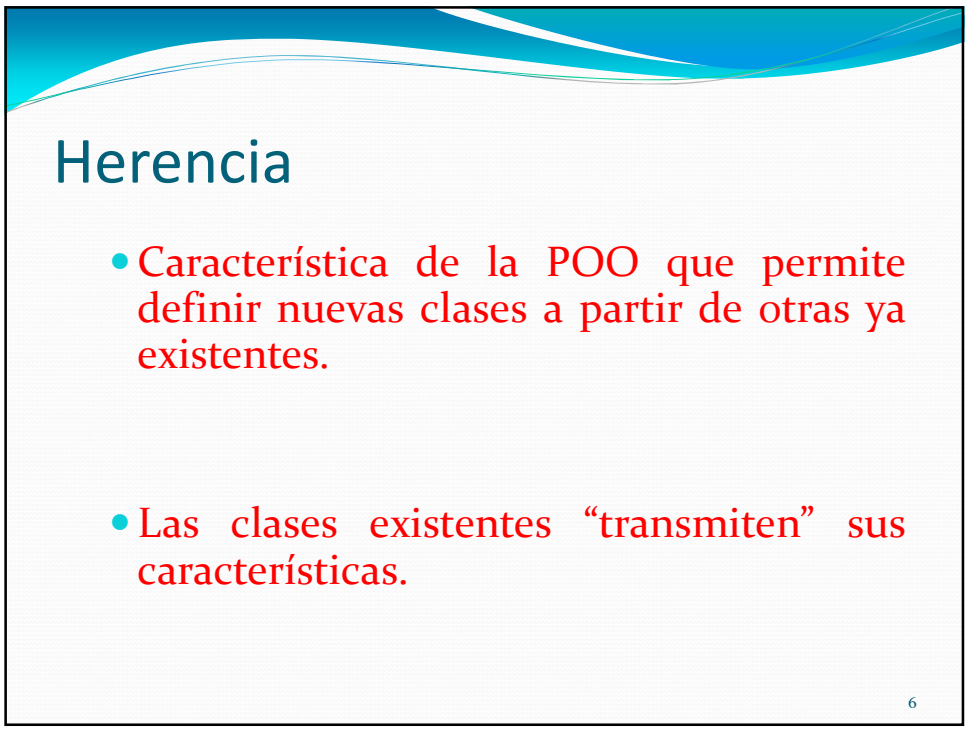
3

# Relaciones entre clases:

## Herencia, Composición y Agregación

4


RELACIONES ENTRE CLASES
Establecen
Conexiones para que no actúen de manera independiente
Representadas mediante
Líneas rectas
Presentan
Niveles o grados de dependencia
Llamados
Cardinalidad o multiplicidad
Especificaciones
1 Uno y sólo uno
1..1 Uno a uno
0..1 Cero ó uno
M..N Desde M hasta N
M,N M ó N
* Cero ó muchos
0..* Cero ó muchos
1..* Uno ó muchos (al menos uno)
Tipos
Herencia
Permite
Definir nuevas clases a partir de otras
Identificando
Elementos comunes
Composición
Relación fuerte entre
Un "todo" y sus "partes"
donde
El "todo" no existe si no existen sus partes
Si se elimina el "todo", desaparecen sus "partes"
Agregación
Relación entre
Un "todo" y sus "partes"
donde
Las "partes" pueden pertenecer a varios "todos"
Si se elimina el "todo", sus "partes" pueden seguir existiendo


Herencia
Característica de la POO que permite definir nuevas clases a partir de otras ya existentes.
Las clases existentes "transmiten" sus características.
6

# 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 |
| --- |
| CaballosDeFuerza: int<br>CantidadDePuertas: int |
| Arrancar() : void<br>Detener() : void<br>Acelerar(int cuanto): void |

| PalaMecanica |
| --- |
| CaballosDeFuerza: int<br>PesoMaximoDeLevante: int |
| Arrancar() : void<br>Detener() : void<br>MoverPala(string direccion) : void |

8 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)
    {
    }
}
```

Iguales

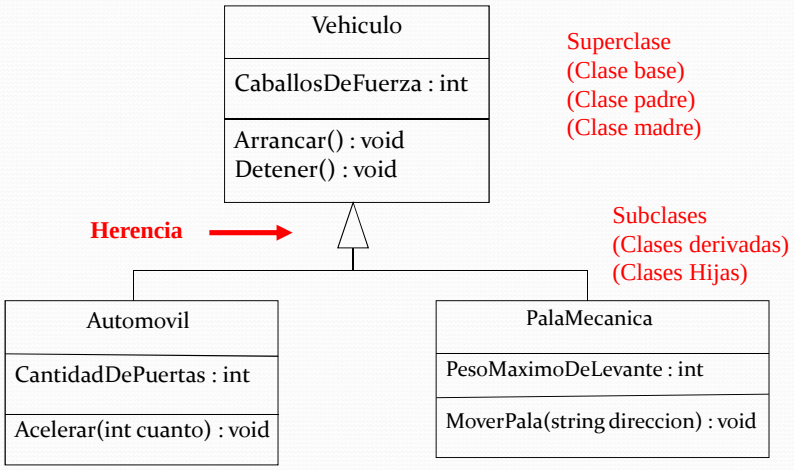
Iguales

Iguales

```
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#

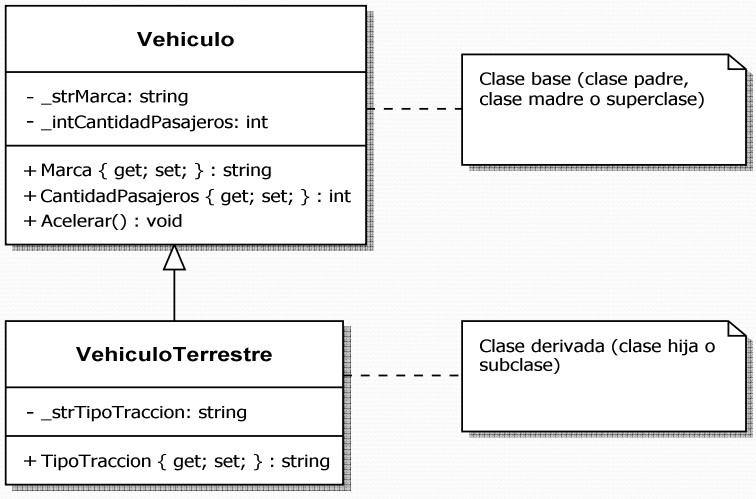
```
class Vehiculo
{
    private int cf;
    public int CaballosDeFuerza
    {
        get { return cf;  }
        set { cf = value; }
    }
    public void Arrancar()
    {
    }
    public void Detener()
    {
    }
}
```

```
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)
    {
    }
}
```

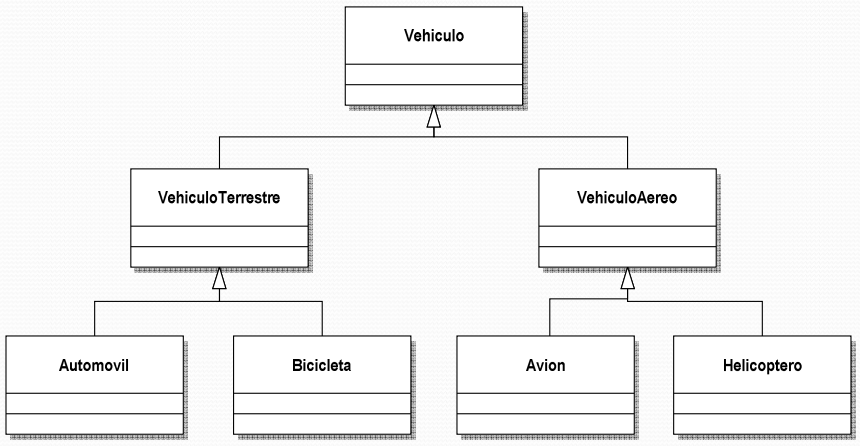
11

## Ejemplo de herencia




12

## Ejemplo de herencia con varios niveles




13

## Tipos de herencia

Herencia

- Simple
- Múltiple

14

# Herencia

**Aparato_Electrodomestico**

(**Propiedades:**
Interruptor,
CableElectrico
**Métodos:**
Encender, Apagar)

Herendan características de
Aparato_Electrodomestico
e incorporan las suyas propias.

**Lavadora**
• **Propiedades:**
(Interruptor, CableElectrico,
***PerillaDeCiclosDeLavado,***
***CapacidadDeCarga***)
• **Métodos:**
(Encender, Apagar,
***LlenarConAgua, TirarAgua***)

**Televisión**
• **Propiedades:**
(Interruptor, CableElectrico,
***BotonDeCanales,***
***BotonDeVolumen***)
• **Métodos:**
(Encender, Apagar,
***CambiarVolumen,***
***CambiarCanal***)

15

# Herencia múltiple


*En C# .NET no se puede implementar herencia múltiple*

16

# Herencia en C#

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

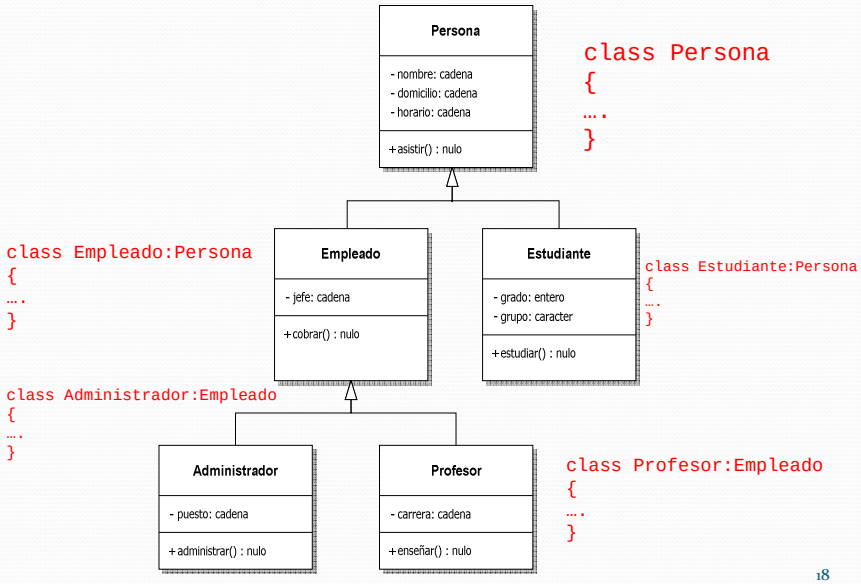
```
class A
{
}
class B : A
{
}
```

Indica que B "Hereda de" A

- Todos los objetos heredan de `System.Object`

17

# Otro ejemplo de herencia




```
class Persona
{
….
}
```

```
class Empleado:Persona
{
….
}
```

```
class Estudiante:Persona
{
….
}
```

```
class Administrador:Empleado
{
….
}
```

```
class Profesor:Empleado
{
….
}
```

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

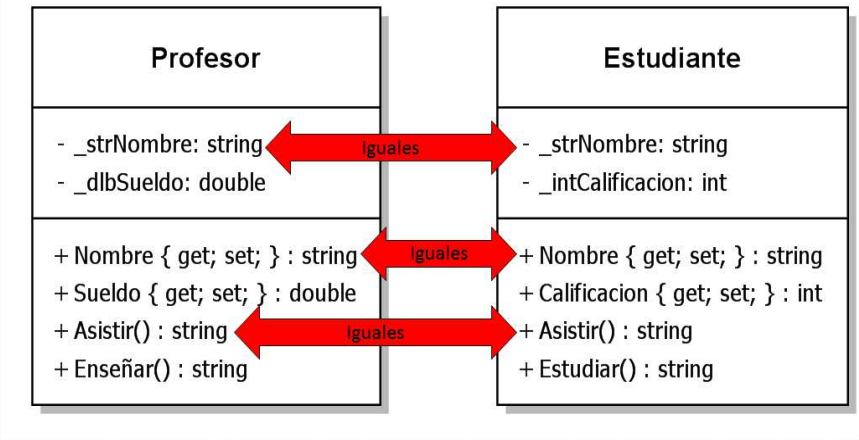
- Una escuela desea modelar los datos y las actividades de sus profesores y de sus estudiantes.

| PROFESOR | |
|---|---|
| **Datos** | Actividades |
| **Nombre (cadena)**<br>**Sueldo (numérico real)** | Asistir a la escuela<br>Enseñar |

| ESTUDIANTE | |
|---|---|
| **Datos** | Actividades |
| **Nombre (cadena)**<br>**Calificación (numérico entero)** | Asistir a la escuela<br>Estudiar |

20

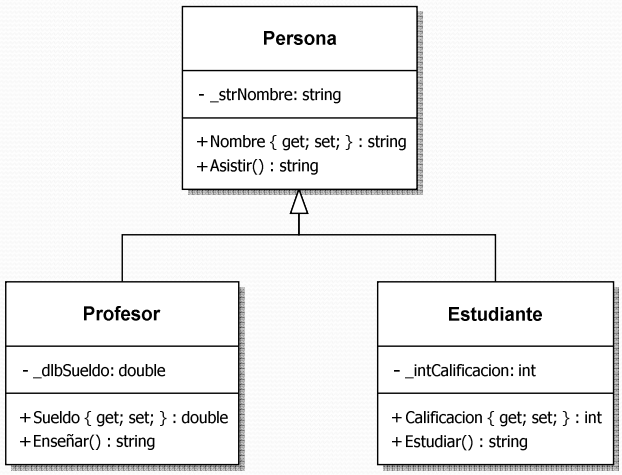
# Mal diseño (no recomendable)




21

# Diseño con herencia

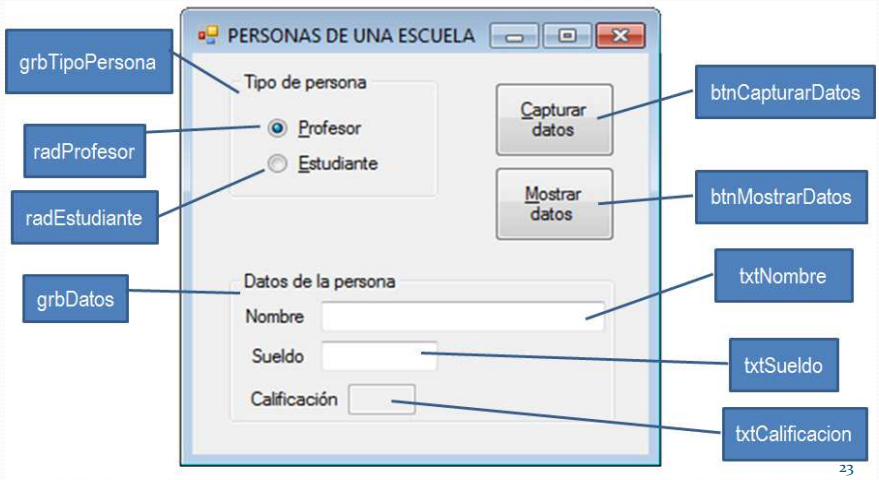
- Prog. 5.1.- HerenciaConsola




22

# Diseño con herencia

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

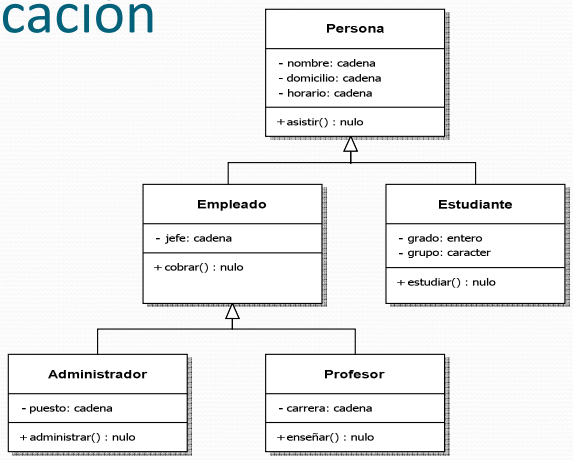



23

# Salida del Prog. 5.2.- HerenciaFormas


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*

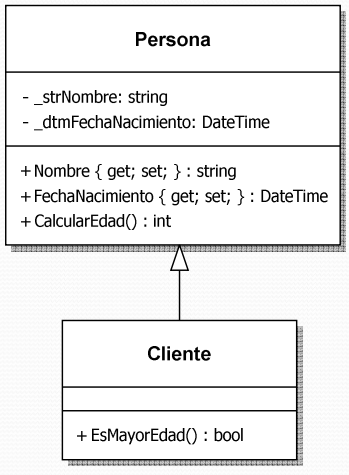
26

## 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`.

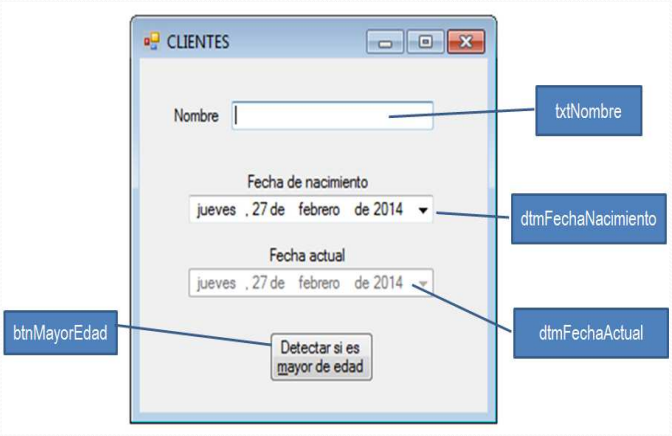
27

## Prog. 5.3.- InvocandoMetodoClaseBase



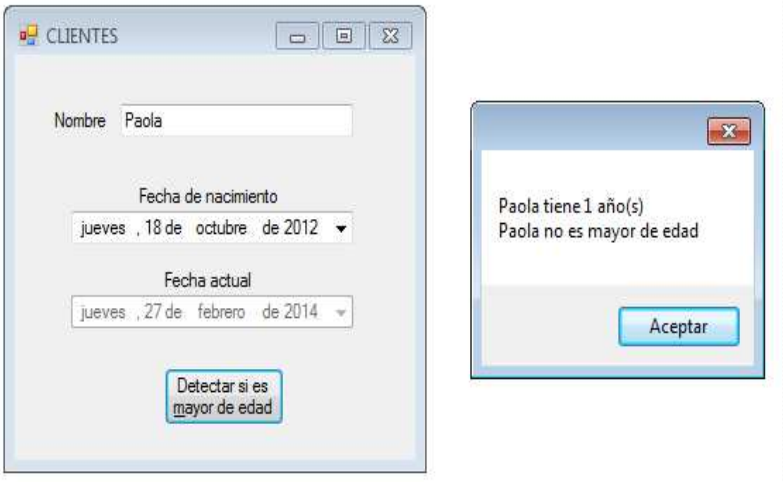

28

# 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

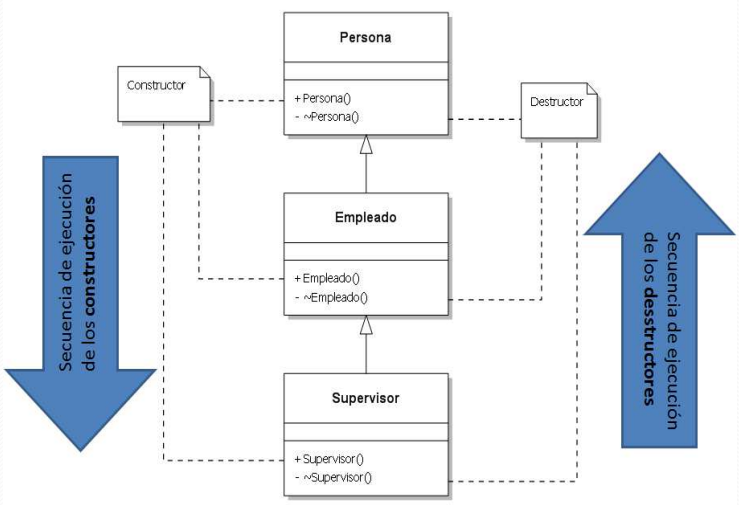
```
// La clase Cliente hereda de la clase Persona
    class Cliente:Persona  // Un cliente "es una" persona
    {
        // Método público para determinar si es mayor de edad
        public bool EsMayorEdad()
        {
            // Variable local
            int intEdad;

            // 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.

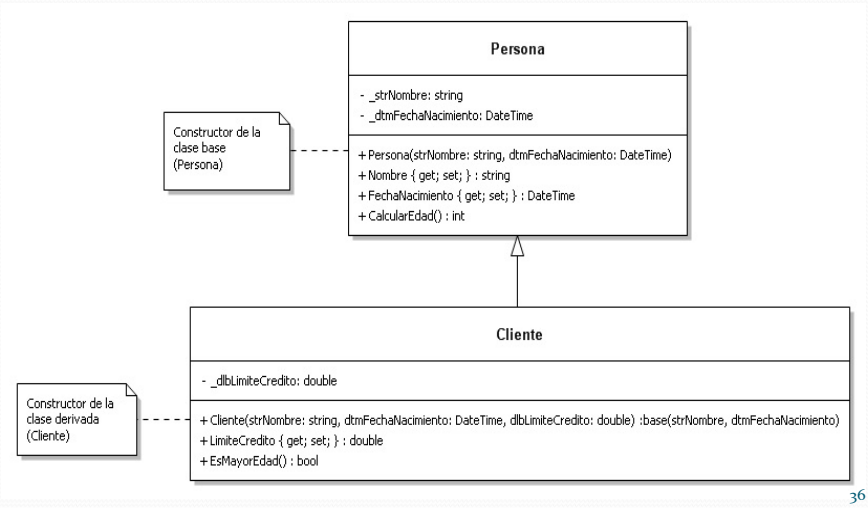
34

## 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`.

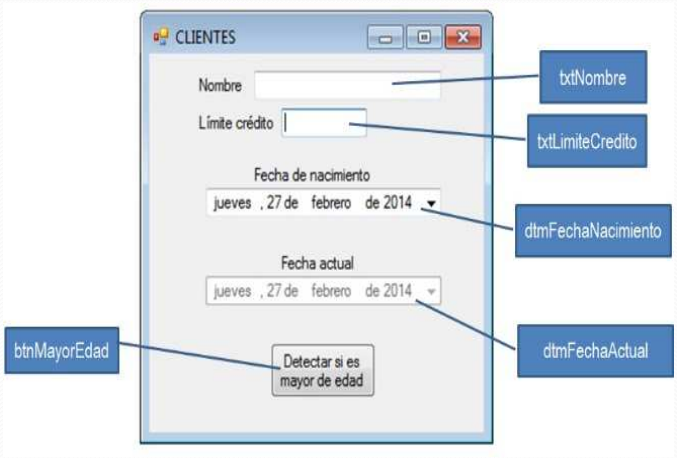
35

## Prog. 5.4.- InvocarConstructorClaseBase



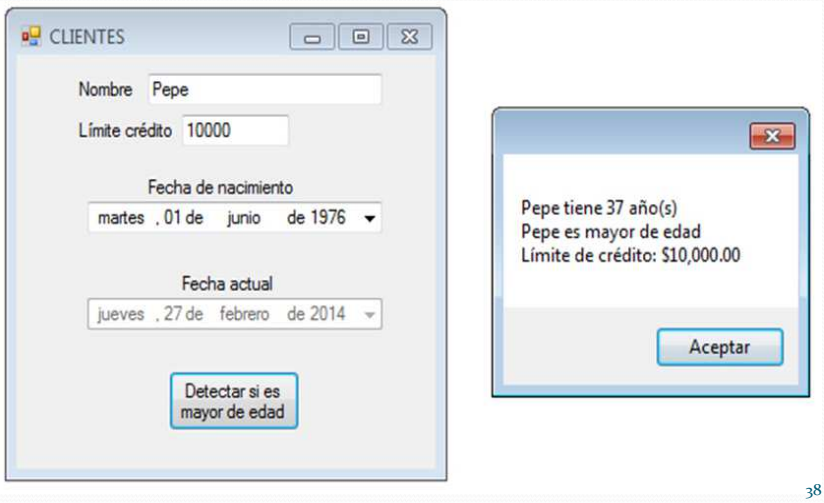

36

# 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

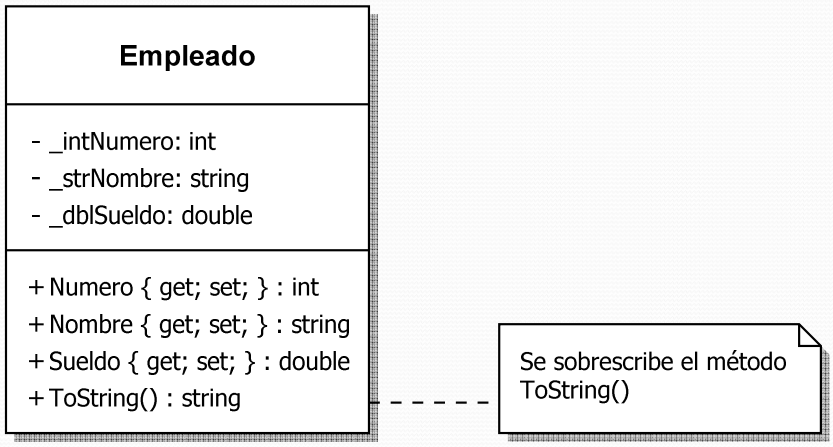
## Evitando la herencia: Clases selladas

- 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 _nombre;
    public string Nombre
    {   get { return _nombre;  }
        set { _nombre = value; }
    }
}
sealed class Empleado : Persona
{   private string _departamento;
    public string Departamento
    {   get { return _departamento; }
        set { _departamento = value; }
    }
}
class EmpleadoTiempoParcial:Empleado
{   private int _horasAsignadas;
    public int HorasAsignadas
    {   get { return _horasAsignadas; }
        set { _horasAsignadas = value; }
    }
}
```

# Ejemplo




41

# 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"));
   }

}
```

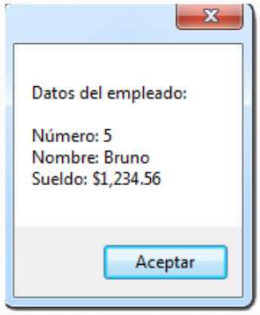
42

# Salida

```
Empleado miEmpleado = new Empleado();

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

MessageBox.Show(miEmpleado);
```



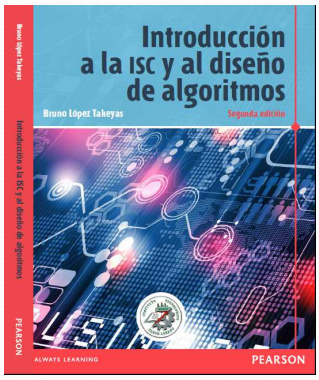

*En versiones de Microsoft Visual Studio anteriores a la 2015 es necesario codificar:*

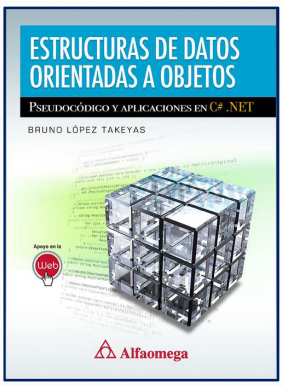
```
MessageBox.Show(miEmpleado.ToString());
```

43

# Otros títulos del autor

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




takeyas@itnuevolaredo.edu.mx

Bruno López Takeyas