Herencia

Video 41

Es la característica mas importante de la POO.

Lo que tiene el padre, pasa al hijo.

Ventaja principal: Posibilita reutilizar código de programación.

Hay establecida una jerarquía (padre-hijo) de la herencia. La jerarquía es entre clases.

La clase padre hereda de la clase abuelo, y la clase hijo, de padre, lo cual incluye lo del abuelo.

Principio de herencia “es-un”.

¿Un jefe es un empleado? Si es así, debe heredar de empleado. Si la respuesta es no, no debe heredar de empleado. ¿Un director es un empleado? Eso si, director debe heredar de empleado. Un jefe no es un director.

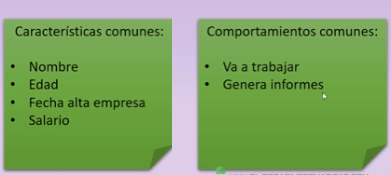


No es la clase mas importante.

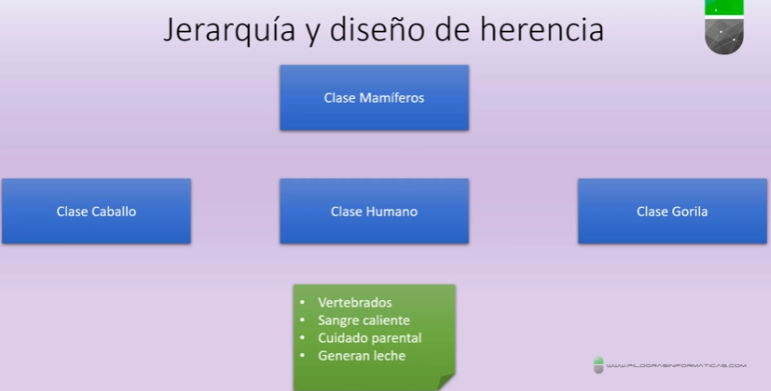


Esta es otra forma válida.

Hay que preguntarse por las características y comportamientos comunes de las 3 clases. Aquí podrían ser:



Si son comunes a todos, se escribe en la clase que está en la cúspide



Acá crea una 4 clase. Ya que de las otras no se desprende una estructura jerárquica.

Video 42

class Program

{

static void Main(string[] args)

{

Caballo Babieca = new Caballo();

Humano Juan = new Humano();

Gorila Copito = new Gorila();

Babieca.cuidarCrias(); //Acá aparecen los métodos de babieca y los de mamífero.

}

}

class Mamiferos //es la clase que está en la cúspide. SUPERCLASE. Es de la que se va a heredar. Hay además otros métodos (equals, GetType, etc), que vienen de la clase object. Esta clase siempre va a estar en la cúspide. También le dicen superclase cósmica.

{

public void respirar()

{

Console.WriteLine("Soy capaz de respirar");

}

public void cuidarCrias()

{

Console.WriteLine("Cuido de mis crías hasta que sean capaces");

}

}

class Caballo : Mamiferos //Así se la hace heredera de la clase padre.

{

public void Galopar()

{

Console.WriteLine("Soy capaz de galopar");

}

}

class Humano : Mamiferos

{

public void Religion()

{

Console.WriteLine("Soy el único gil religioso");

}

}

class Gorila:Mamiferos

{

public void trepar()

{

Console.WriteLine("Soy capaz de trepar");

}

}

Video 43

Si a una clase no le vemos el constructor, se le da un constructor por defecto. Método de igual nombre que la clase. No recibe parámetro y se encuentra vacío.

Cada constructor de una subclase llama al constructor de la clase padre, con una instrucción que no se ve que es :base(). Le dicen que inicie los objetos. Llama al constructor por defecto.

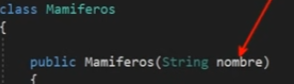
Recorrido que sigue el flujo de ejecución:

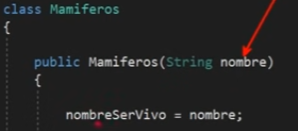


Cuando llega a esa línea llama al constructor de la clase humano y le pasa por parámetro el string Juan



Y se almacena ahí. Pero luego se pasa a :base y llama al constructor de la clase padre mamíferos. Y se almacena ahí:





Luego nombreSerVivo=nombre;

Luego, el método getNombre(); con el .Juan del ppio imprime nombreSerVivo

class Program

{

static void Main(string[] args)

{

Caballo Babieca = new Caballo("Babieca"); //Este "Babieca" viaja hasta String nombreCaballo de cuando se crea el constructor de caballo, y luego va a :base(nombreCaballo)

Humano Juan = new Humano("Juan");

Gorila Copito = new Gorila("Copito");

Juan.getNombre();

}

}

class Mamiferos

{

public Mamiferos(String nombre) //el constructor

{

nombreSerVivo = nombre; //El constructor por defecto ha sido reemplazado por este que acabamos de crear. A partir de aquí las subclases marcarán error. Porque se cambió el constructor por defecto. Tienen que llamar al nuevo constructor.

}

public void respirar()

{

Console.WriteLine("Soy capaz de respirar");

}

public void cuidarCrias()

{

Console.WriteLine("Cuido de mis crías hasta que sean capaces");

}

public void getNombre()

{

Console.WriteLine("El nombre del ser vivo es: " + nombreSerVivo);

}

private String nombreSerVivo;

}

class Caballo : Mamiferos

{

public Caballo (string nombreCaballo):base(nombreCaballo) //así se llama al nuevo constructor... Ahora marca error cuando instanciamos caballo babieca.

{

}

public void Galopar()

{

Console.WriteLine("Soy capaz de galopar");

}

}

class Humano : Mamiferos

{

public Humano (String nombreHumano) : base(nombreHumano) { }

public void Religion()

{

Console.WriteLine("Soy el único gil religioso");

}

}

class Gorila:Mamiferos

{

public Gorila (String nombreGorila): base(nombreGorila) { }

public void trepar()

{

Console.WriteLine("Soy capaz de trepar");

}

}

Video 44

Principio de sustitución

Sustitución. Se sustituye un objeto de un tipo por otro objeto de otro tipo. “Es siempre un”. Ayudaría a armar la jerarquía de herencias.

Un caballo ¿“Es siempre un humano”? No

Un humano ¿es siempre un mamífero? Si. Lo inverso no es siempre cierto.

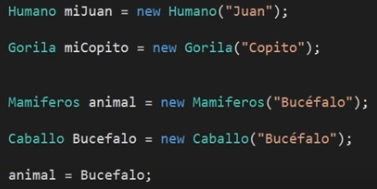
Mamíferos animal=new Caballo(); es correcto

Una variable de tipo mamíferos, contendrá un objeto de tipo caballo.

Caballo animal=new Mamiferos(); es incorrecto

Mamiferos Pepe = new Caballo("Pepe"); //primero va la clase mas general, luego la particular.

Eso es correcto. Ahora, si se declara así, Pepe solo puede acceder a las propiedades y métodos de la clase mamíferos.



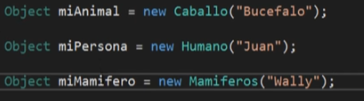
También se puede hacer así. Creo que tampoco se puede acceder a los métodos y propiedades de la clase caballo.

Object

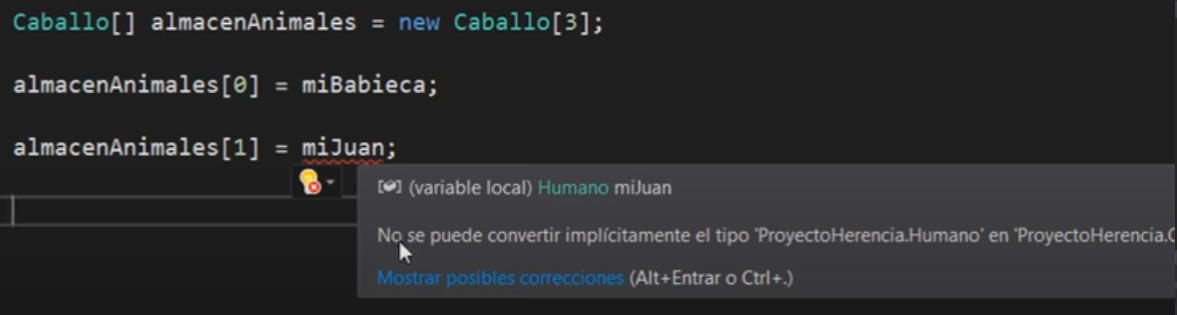
Superclase cósmica

Así como mamíferos está arriba de las otras 3 en la jerarquía, Object es la que está arriba de todas.

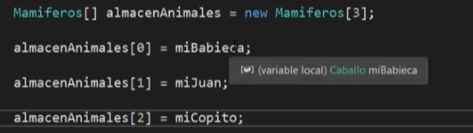
También se puede esto, con Object:



Si se crea un array de tipo Caballo, no se puede guardar ahí un objeto de tipo humano.



En este caso viene bien la sustitución:



Aquí, nuevamente, solo se podrá acceder a los métodos y propiedades de la clase mamíferos .

Video 45

Polimorfismo

Pasa el método pensar que está en la clase humano a la clase mamíferos. Yo uso religión. O sea, crea un método de igual nombre, aunque de contenido distinto en dicha clase.

Ahora sale un subrayado en verde en el método pensar en la clase humano. El método de humano invalida/oculta el método heredado de mamifero. Se podría querer que funcione así, o que estemos cometiendo un error. Si este método recibiese parámetros distintos no habría invalidación, habría sobrecarga.

Si uno le pone new, se quita la advertencia, pero todo funciona igual que si se pisara el método heredado. Se le avisa al compilador que uno sabe lo que está haciendo.

new public void Religion()

Le agrega virtual al método padre. Se le indica que todas sus subclases deberían de tener un método pensar que modifiquen el método padre.

public virtual void Religion() //En la class mamifero

Ahora ya el compilador no trata a los métodos como independientes, sino que son subclases del padre. Aunque aún el error sigue apareciendo. Hay que agregar override a los submétodos y ahí si desaparece el subrayado verde.

public override void Religion()

Ahora, tras todo lo que se hizo, de virtual en el método de la clase padre, y override en 2 de las 3 clases hijas:

Mamiferos[] almacenAnimales = new Mamiferos[3];

almacenAnimales[0] = Babieca;

almacenAnimales[1] = Juan;

almacenAnimales[2] = Copito;

for (int i = 0; i < 3; i++)

{

almacenAnimales[i].Religion();

}

/\* almacenAnimales:

\* Cuando se le pasa Babieca se comporta como mamífero. (porque no se creó un método (ni hizo override) en la clase hijo.

\* Cuando se le pasa Juan como humano.

\* Cuando se le pasa Copito como un gorila

Se comporta de distinta forma. A esto se lo llama polimorfismo.\*/



El objeto array se comporta de diferente forma. Por eso es polimorfismo.

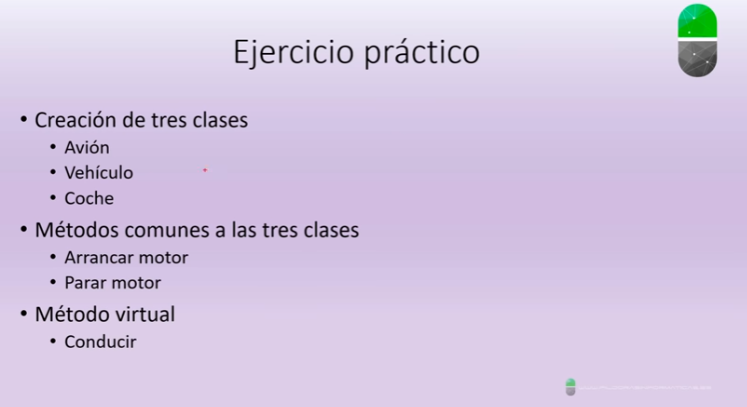
Video 46

Modificadores de acceso

* Public permite acceso desde cualquier otro lugar del programa, desde cualquiera de sus clases.

En general no se declaran variables publicas dentro de clases. Se le suele poner private. Si se necesita cambiar su valor (o acceder a él), se crea un método público que lo modifique.

* Private permite solo el acceso dentro de la clase donde fue creado, encapsula.
* Protected es una mezcla de los 2. Será accesible dentro de la propia clase y de las clases que hereden. No se suele usar mucho, aclara.



namespace Herencia\_Ejercicio

{

class Program

{

static void Main(string[] args)

{

avion LAPA = new avion();

coche Twingo = new coche();

LAPA.conducir();

Twingo.conducir();

}

public class vehiculo

{

public vehiculo()

{

}

public void arrancarMotor ()

{

Console.WriteLine("El motor ha arrancado");

}

public void pararMotor()

{

Console.WriteLine("El motor se ha detenido");

}

public virtual void conducir()

{

Console.WriteLine("Se está conduciendo el vehículo");

}

}

public class avion:vehiculo

{

public avion()

{

}

public override void conducir()

{

Console.WriteLine("Se está conduciendo el avión");

}

}

public class coche:vehiculo

{

public coche()

{

}

public override void conducir()

{

Console.WriteLine("Se está conduciendo el coche");

}

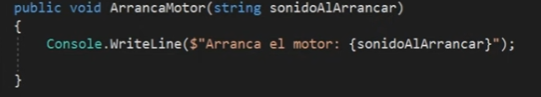
}

}

}

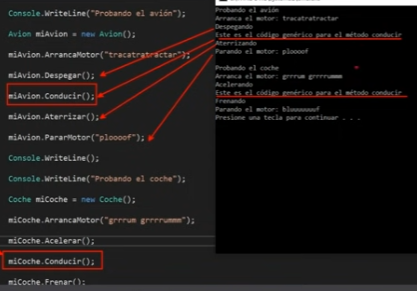
Esto es lo que hice, creo que contesta la tarea propuesta. Devuelve:



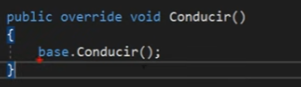


El agrega ese parámetro, dentro de vehiculo. Calculo que se le pasa cuando, en el main se llama al método.

Además, les agrega métodos propios a avión y a coche.



Ahí le falta adaptar a avión y coche a sus particularidades.



Si uno pone eso, se ejecuta el código de la clase padre. Luego de eso, se le podría agregar código para que además haga otras cosas.

Agrega luego, polimorfismo en acción. Sustituye y llama al método.

El polimorfismo se ve ya que se trata del mismo objeto, comportándose de forma diferente, en tiempo de ejecución:

