

ALGORITMOS Y PROGRAMACIÓN

Clase 6

Programación orientada a objetos: Herencia

Temario

- Herencia
- Jerarquía de clases
- Clases abstractas
- Polimorfismo



Herencia de clases

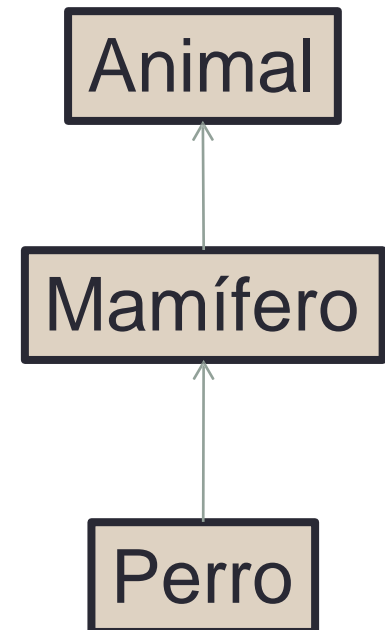
La herencia *permite crear nuevas clases que reutilizan, extienden y modifican el comportamiento definido previamente en otras clases.*

Nos permite concebir una nueva clase de objetos como un refinamiento de otra clase de objetos ya existente, conservando **en la clase base** las similitudes entre las mismas y especificando **en la clase derivada** solamente las diferencias de la nueva clase de objetos.

Herencia de clases

La clase cuyos miembros se heredan se denomina **clase base o superclase** y la clase que hereda esos miembros se denomina **clase derivada o subclase**.

Animal es la **superclase** de Mamífero y
Mamífero es **subclase** de Animal.

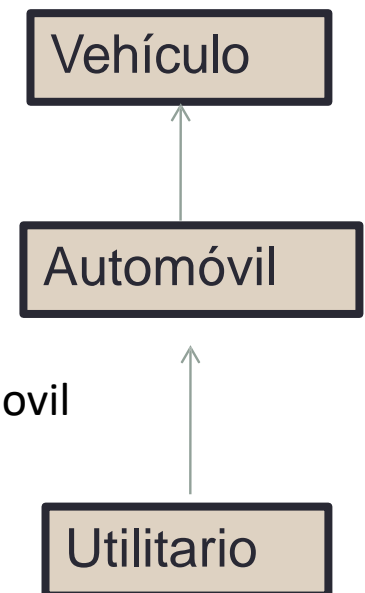


Herencia de clases

A partir de la relación **“es un”** se arma una **jerarquía de clases**, donde las subclases son más específicas y las superclases son más generales.

La relación **“es un”** entre clases: nos permite establecer que una clase **es como otra**, con la excepción de que la nueva clase incluye atributos y comportamiento extra.

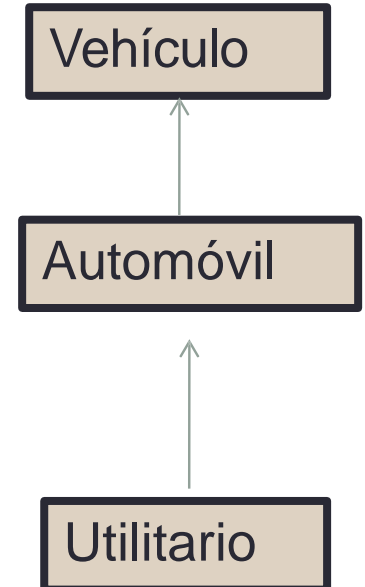
Se lee: un utilitario **ES UN** automovil



Herencia de clases

En una jerarquía de clases las clases hijas o subclases, heredan el estado y el comportamiento de las clases que se encuentran más arriba en la jerarquía (superclases).

La clase Utilitario hereda estado y comportamiento de la clase Automóvil y de la clase Vehículo.



Herencia de clases

- Supongamos tener la clase Automovil que representa la funcionalidad de un auto cualquiera.

```
public class Automovil{  
    private string marca, modelo;  
  
    public Automovil(string mar, string mod);  
    public void encenderMotor();  
    public void apagarMotor();  
    public void acelerar(double vel);  
    public void frenar();  
    public void doblar(double ang);  
    public void tocarBocina();  
    public void cargarCombustible();  
    public void encenderLuces();  
    public void imprimir ();  
    .....  
}
```

Herencia de clases

- Si necesitamos modelar un auto de policía, que además de hacer todo lo que hace cualquier auto, enciende la sirena. ¿Qué cambios habría que hacer en la clase Automovil?
- Si necesitamos modelar un vehículo utilitario que además de hacer todo lo que hace cualquier auto, carga y descarga mercadería. ¿Qué cambios habría que hacer en la clase Automovil?



Herencia de clases

- La clase Automovil tiene mucha funcionalidad implementada.
- Sería ideal poder usar toda esa funcionalidad de manera abstracta sin hacer modificación alguna.
- La herencia de clases nos permite usar el estado y comportamiento de la clase Automovil para derivar nuevas clases.

Herencia de clases

Se crean las clases MovilPolicial y Utilitario como subclases de Automovil.

```
public class MovilPolicial : Automovil {  
    public void encenderSirena();  
    public void apagarSirena();  
    .....  
}  
  
public class Utilitario : Automovil {  
    public void cargar();  
    public void descargar();  
    ....  
}
```

La clase MovilPolicial y Utilitario heredan todo el estado y comportamiento de la clase Automovil

A la clase Automovil se la denomina superclase.

A la clase hija o derivada se la llama subclase.

Herencia de clases

```
public class MovilPolicial : Automovil {
```

```
    private int numeromovil;
```

```
    public void encenderSirena();  
    public void apagarSirena();  
    .....
```

```
}
```

```
public class Utilitario : Automovil {
```

```
    private double capacidadDeCarga;
```

```
    public void cargar();  
    public void descargar();  
    .....
```

```
}
```

Cada subclase establece su propio estado "extra". El resto de los atributos los hereda.

Cada subclase agrega comportamiento para manipular sus nuevos atributos.

Modificadores de acceso

- Para que una subclase pueda acceder directamente a los miembros de su superclase, éstos deben ser declarados como públicos (`public`) o protegidos (`protected`).
- Un miembro público puede ser accedido desde cualquier otra clase.
- Un miembro protegido puede ser accedido tanto desde la propia clase que lo declara como desde cualquier subclase.
- Los miembros privados (`private`) solo pueden ser accedidos desde la propia clase.

Modificadores de acceso

Definimos los atributos marca y modelo como protected en la superclase.

```
public class Automovil {  
    protected string marca, modelo;  
    public Automovil (string marca, string modelo) {  
        this.marca = marca;  
        this.modelo = modelo;  
    }  
  
    .....  
}
```

Modificadores de acceso

```
public class MovilPolicia : Automovil {  
    private int numeromovil;  
  
    public MovilPolicia(string mar, string mod, int n) : base(mar, mod){  
        numeromovil = n;  
    }  
    public void encenderSirena();  
    public void apagarSirena();  
    .....  
}
```

En el caso de las subclases
sus miembros son private

```
public class Utilitario : Automovil {  
    private double capacidadDeCarga;  
  
    public Utilitario(string mar, string mod, double cc) : base(mar, mod){  
        capacidadDeCarga = cc;  
    }  
    public void cargar();  
    public void descargar();  
    .....  
}
```

Herencia de clases

```
public class MovilPolicial : Automovil {  
    private int numeromovil;  
  
    .....  
    public void encenderSirena();  
    public void apagarSirena();  
    public void impri()  
    {      Console.WriteLine(marca, modelo, numeromovil);  
    .....  
}  
  
public class Utilitario : Automovil {  
    private double capacidadDeCarga;  
  
    .....  
    public void cargar();  
    public void descargar();  
    .....  
}
```

**Gracias a la herencia,
las variables
declaradas como
protected en la
superclase pueden ser
usadas directamente
en la subclase**

Herencia de clases

```
public class MovilPolicial : Automovil {
    private int numeromovil;

    public MovilPolicial(string mar, string mod, int n) :
        numeromovil = n;
    }
    public void encenderSirena();
    public void apagarSirena();
    public void impri()
    {
        Console.WriteLine(marca,modelo,numerovil)
        .....
    }

    public class Utilitario : Automovil {
        private double capacidadDeCarga;

        public Utilitario(string mar, string mod, double cc) :
            capacidadDeCarga = cc;
        }
        public void cargar();
        public void descargar();
        .....
    }
```

C# obliga a definir el constructor en cada subclase.

base(mar, mod) {

Con el comando base le decimos al S.O. que antes de ejecutar el propio constructor se ejecute el constructor de la superclase que da valor a los atributos heredados (marca y modelo).

base(mar, mod) {

Herencia de clases

```
public class MovilPolicial : Automovil {  
    private int numeromovil;  
    public MovilPolicial(string mar, string mod, int n) : base(mar, mod){  
        numeromovil = n;  
    }  
    public void encenderSirena();  
    public void apagarSirena();  
    .....  
}
```

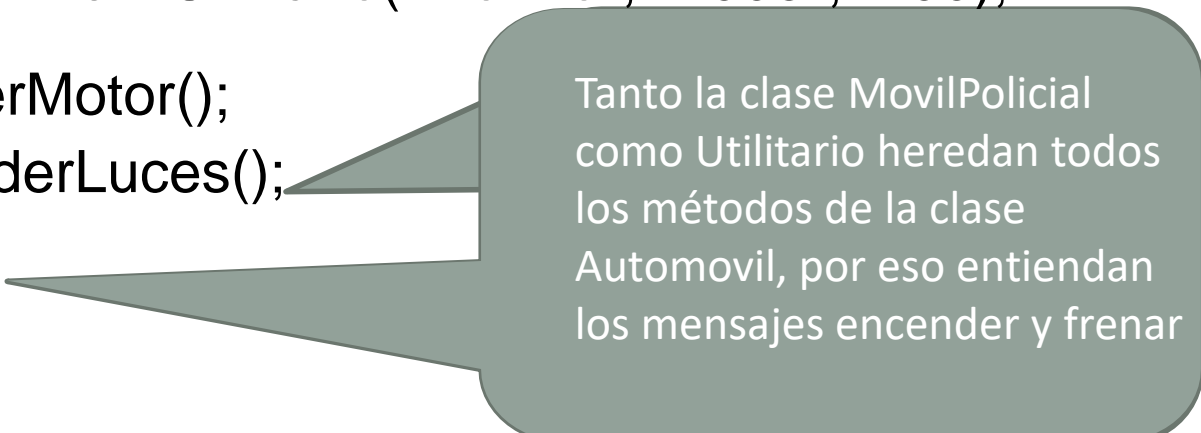
```
public class Utilitario : Automovil {  
    private double capacidadDeCarga;  
    public Utilitario(string mar, string mod, double cc) : base(mar, mod){  
        capacidadDeCarga = cc;  
    }  
    public void cargar();  
    public void descargar();  
    .....  
}
```

Los constructores de las subclases pueden tener su propio código de inicialización, que se ejecuta después de ejecutar el comando base.

Herencia de clases

Las clases MovilPolicial y Utilitario son utilizadas como cualquier otra clase. Heredan los métodos de la superclase.

```
public void Main () {  
    Automovil a = new Automovil("Siena", "2007");  
    MovilPolicial mp = new MovilPolicial("Corsa", "2012",125);  
    Utilitario u = new Utilitario("Fiorino", "2003",1200);  
  
    a.encenderMotor();  
    mp.encenderLuces();  
    u.frenar();  
}
```



Tanto la clase MovilPolicial como Utilitario heredan todos los métodos de la clase Automovil, por eso entiendan los mensajes encender y frenar

Herencia de clases

- Las clases MovilPolicial y Utilitario son utilizadas como cualquier otra clase.

```
public void Main () {  
    Automovil a = new Automovil("Siena", "2007"); MovilPolicial  
    mp = new MovilPolicial("Corsa", "2012",125);  
    Utilitario u = new Utilitario("Fiorino", "2003",1200);  
  
    a.apagarMotor();  
    mp.encenderSirenas();  
    u.cargar();  
}
```

La clase Automovil implementa el método apagarMotor.
La clase MovilPolicial define el método encenderSirenas; y la clase Utilitario implementa cargar. Estos métodos los conoce solo la clase que los declara.

Usando modificadores de acceso para definir clases

- Las **clases** pueden declararse como públicas o internas.
 - Las **clases públicas** son precedidas por el modificador de acceso **public**. Estas clases pueden ser accedidas desde todas las clases y proyectos.
 - Las **clases internas** (valor predeterminado) son precedidas por el modificador de acceso **internal** o son aquellas que no tienen definido un modificador de acceso. Solo pueden ser accedidas desde las clases comprendidas dentro del mismo proyecto, pero no desde otros.

Ejemplo:

- **Automovil** y **Utilitario** son clases internas que no pueden ser accedidas desde otro ensamblado.
- **Vehiculo** es una clase pública que puede ser accedida desde otro ensamblado

```
class Automovil {  
  
}  
  
internal class Utilitario {  
  
}  
  
public class Vehiculo {  
  
}
```

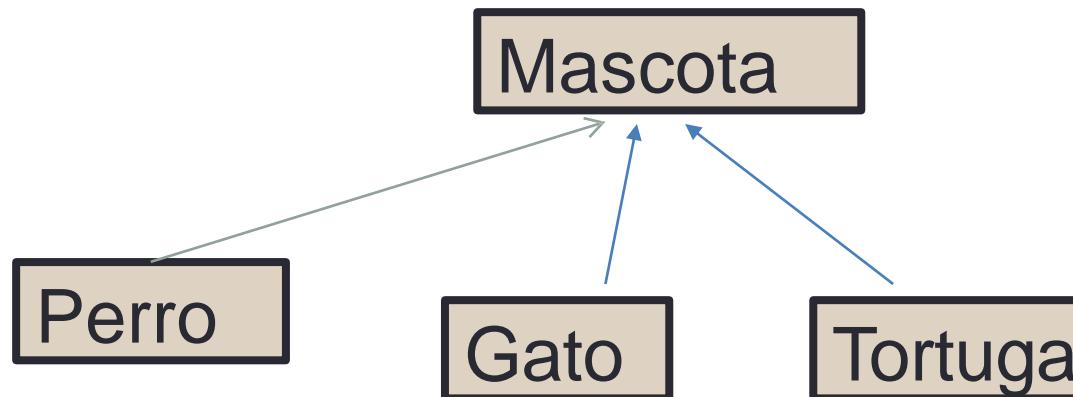
Un ensamblado es un compilado ejecutable (EXE) o una biblioteca de clases (DLL).

No se pueden definir clases con el modificador **private** o **protected** a menos que sea una clase anidada dentro de otra

Puesta en común

Dada la clase Mascota, ***arme una jerarquía de clases*** donde las subclases sean las diferentes especies de animales.

Si ahora las especies son subclases (Perro, Gato, Caballo) ¿Cuál es la superclase?



¿Cómo se implementa en C#?

Empecemos redefiniendo la clase Mascota, que es la superclase

```
public class Mascota{                                /*le sacamos el atributo especie*/
    protected string nombre;
    protected Dueño dueño;
    protected int edad;

    public Mascota(string nom){
        nombre = nom;
    }
    public Mascota(string nom, Dueño dueño, int e){
        nombre = nom;
        this.dueño = dueño;
        edad = e;
    }
    .....
}
```

¿Cómo se implementan las subclases?

```
public class Perro : Mascota{
    private string raza;
    public Perro (string nom, string ra) : base(nom){
        raza=ra;}
    public Perro (string nom, Dueño dueño, int e, string ra) : base(nom, dueño, e){
        raza=ra; }
}
```

Recordar que se deben implementar los mismos constructores definidos en la superclase.

```
public class Gato : Mascota{
    private bool siames;
    public Gato (string nom, bool esSia) : base(nom){
        siames=esSia; }
    public Gato(string nom, Dueño dueño, int e, bool esSia) : base(nom, dueño, e){
        siames=esSia;
    }
}
```

```
public class Tortuga : Mascota{
    .....}
```


Cada subclase debe definir su propio estado y comportamiento extra.

```
public class Perro : Mascota{
    private string raza;

    public Perro (string nom, string ra) : base(nom)
    {
        raza = ra;
    }
    public Perro (string nom, Dueño dueño, int e, string ra) : base (nom, dueño, e)
    {
        raza=ra; }

    public string Raza
    {   get { raza=value;}
        set {return raza;}
    }
    public void perseguirAlGato(){
        Console.WriteLine("Persiguiendo un gato");
    }
}
```

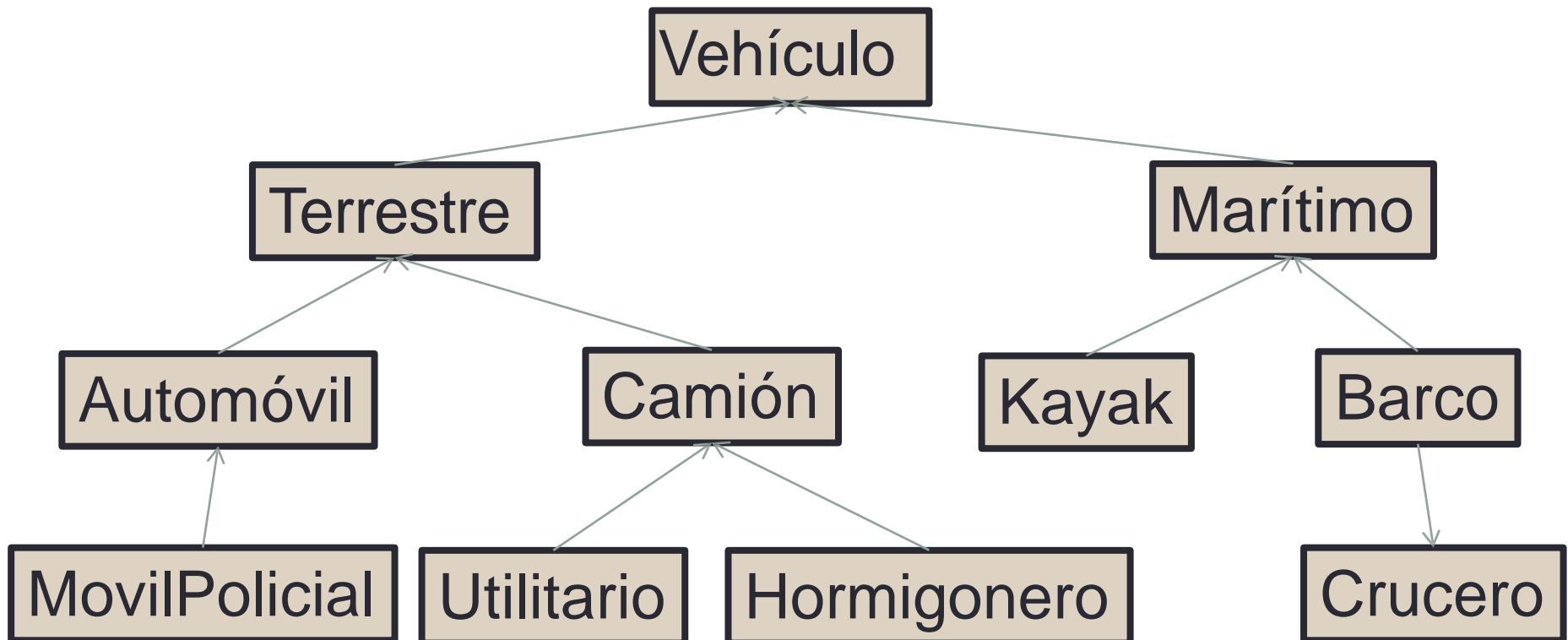
```
public class Gato : Mascota{
    private bool siames;

    public Gato (string nom, bool esSia) : base(nom)
    {
        siames = esSia;
    }
    .....

    public escaparDelPerro{
        Console.WriteLine("!!!!Que alguien me salve!!!!");
    }
}
....
....
.....
```

Jerarquía de clases

- Por lo general las superclases solo tienen estado y comportamiento en común, pero raramente se instancian.
- Siempre se trabaja con las subclases



Jerarquía de clases

La idea es que en las superclases esté todo lo común a las subclases

¿Qué tienen en común todos los vehículos?

```
public class Vehículo {  
    protected string marca, modelo;  
    public void acelerar(double velocidad);  
    public void girar(double angulo);  
    public void frenar();  
}
```

Jerarquía de clases

Un Vehículo sabe acelerar, frenar y girar, pero estas acciones son completamente diferentes en cada vehículo.

- La aceleración, el frenado y el giro en un auto no tiene nada que ver con las de un kayak o un barco.

¿Cómo se pueden especificar acciones comunes a distintos tipos de Vehículos, pero cuyas implementaciones difieran completamente?

Usando una Clase Abstracta

Una clase abstracta:

Se usa para definir una superclase que engloba estados y comportamientos comunes de sus clases derivadas

No puede ser instanciada

Declara métodos abstractos, es decir, que no tienen implementación. Sirven para especificar qué hacen los objetos pero no dicen cómo lo hacen.

Puede declarar métodos concretos, con implementación común a cualquier subclase.

Se definen con la siguiente sintaxis:

abstract modificador class Nombre

Una clase que contiene algún métodos abstracto debe ser definida como abstracta.

La clase vehículo dice que todo vehículo sabe acelerar, girar y frenar, pero "no dice" cómo hacerlo. Por eso los métodos se declaran como abstractos.

```
abstract public class Vehículo {  
    protected string modelo, marca;  
  
    abstract public void acelerar(double vel);  
    abstract public void girar(double angulo);  
    abstract public void frenar();  
  
    public void imprimir(){  
        Console.WriteLine("Marca {0}  Modelo {1} ", marca, modelo);  
    }  
}
```

**Los métodos
abstractos
NO tienen
bloque de
código.**

**Una clase abstracta no puede ser instanciada.
La sentencia `Vehiculo v=new Vehiculo();` da
ERROR.**

```
abstract public class Vehículo {  
    protected string modelo, marca;
```

```
    abstract public void acelerar(double vel);
```

```
    abstract public void girar(double angulo);
```

```
    abstract public void frenar();
```

```
        public void imprimir(){  
            Console.WriteLine("Marca {0}  Modelo {1} ", marca, modelo);  
        }
```

```
}
```

**Método concreto común a
cualquier subclase.**


```
abstract public class Vehículo {  
    protected string modelo, marca;
```

```
    abstract public void acelerar(double vel);  
    abstract public void girar(double angulo);  
    abstract public void frenar();
```

```
    public void imprimir() {  
        Console.WriteLine("Modelo: {0}, Marca: {1}",  
            modelo, marca);  
    }  
}
```

Toda subclase de Vehículo, está **OBLIGADA** a implementar estos Métodos abstractos.

Redefinición de métodos

Si una superclase declara un método como ***abstract***, ese método se debe implementar en todas las subclases no abstractas de dicha clase.

Para implementar el método en la subclase se usa ***override***.

```
public class Terrestre : Vehículo {  
    override public void acelerar(double vel) {.....  
}  
    override public void girar(double angulo){  
        Console.WriteLine("giro en cualquier ángulo"); }  
    override public void frenar(){....  
    }  
}
```

Si la clase Vehículo es abstracta, entonces la clase Terrestre debe implementar los métodos abstractos de Vehículo (salvo que Terrestre sea abstracta también)

Redefinición de métodos

En algunos casos, la reimplementación de algún método implica ejecutar un método de la superclase. Para ello se usa el comando base.

```
public class Utilitario : Camion {  
    override public void cargar () { ..... }  
    override public void descargar () { ..... }  
  
    override public void girar(double angulo){  
        base.girar(angulo);  
        Console.WriteLine("uso las 4 ruedas para girar");  
    }  
}
```

Binding dinámico

- Las subclases agregan nuevas variables y métodos, cambiando en algunos casos el comportamiento de los métodos heredados.
- El método a ejecutar se establece en tiempo de ejecución: el S.O. evalúa en ese momento de qué clase es la instancia que recibe el mensaje.
- En POO la ligadura o asociación en tiempo de ejecución entre un objeto y el método a ejecutar se conoce como **binding dinámico**.

Binding dinámico

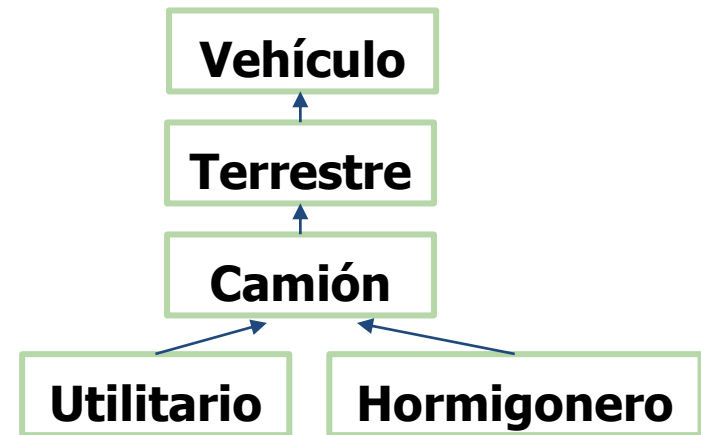
```
Camion [ ] camiones = new Camion [2];  
camiones[0] = new Utilitario();  
camiones[1] = new Hormigonero();
```

```
foreach(Camion c in camiones)  
    c.girar(45);
```

¿Qué método girar() se ejecuta?

Depende del objeto receptor del mensaje...primero ejecuta el girar de Utilitario y luego el de Hormigonero....

Recordemos la Jerarquía



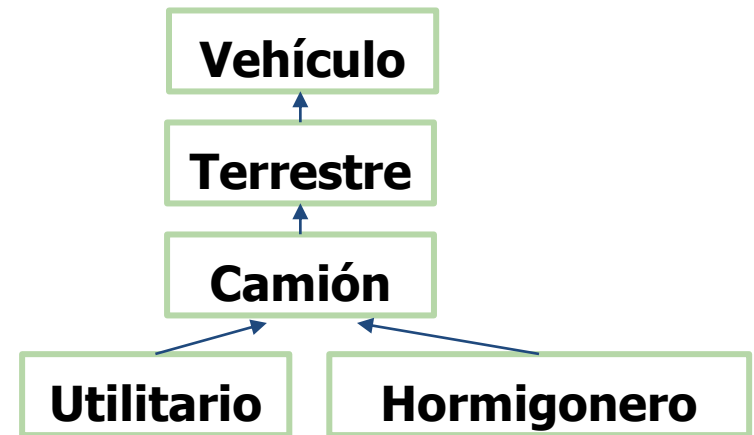
```
abstract public class Vehículo {  
    protected string modelo, marca;  
  
    abstract public void acelerar(double vel);  
    abstract public void girar(double angulo);  
    abstract public void frenar();  
  
    public void imprimir() {  
        .....;  
    }  
}
```

Binding dinámico

```
Camion [ ] camiones = new Camion [2];  
camiones[0] = new Utilitario();  
camiones[1] = new Hormigonero();
```

```
foreach(Camion c in camiones)  
    c.girar(45);
```

Recordemos la Jerarquía



Cuando un objeto recibe un mensaje, el S.O. por binding dinámico irá a buscar el método asociado a la clase de dicho objeto. Si NO encuentra el método buscado va a salir a buscarlo por árbol jerárquico. Sube un nivel hacia arriba y se fija si en esa superclase lo encuentra. Si no está allí, repite el proceso. Este mecanismo se denomina **look up**.

Binding dinámico

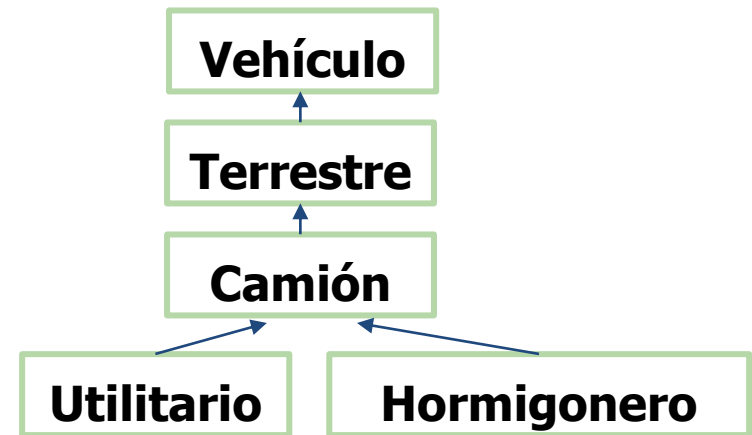
```
Camion [ ] camiones = new Camion [2];  
camiones[0] = new Utilitario();  
camiones[1] = new Hormigonero();
```

```
foreach(Camion c in camiones)  
    c.girar(45);
```

```
public class Utilitario : Camion {  
    override public void cargar () { ..... }  
    override public void descargar () { ..... }  
    override public void girar(double angulo){  
        base.girar(angulo);  
        Console.WriteLine("uso las 4 ruedas para girar");  
    }  
}
```

La instancia de Utilitario ejecuta su propio método.

Recordemos la Jerarquía



```
public class Terrestre : Vehículo {  
    override public void acelerar(double vel) {...}  
    override public void girar(double angulo){  
        Console.WriteLine("giro en cualquier ángulo");  
    }  
    override public void frenar(){... }  
}
```

La instancia de Hormigonero usa el método heredado de Vehículo.

Polimorfismo

Cuando se redefinen los métodos de una clase abstracta, los métodos se convierten en **polimórficos**: tienen el mismo nombre pero ejecutan acciones diferentes, ya que en cada clase la funcionalidad del método puede ser diferente.

Las clases Terrestre, Camión, Utilitario y Hormigonero tienen todos el método girar(), pero no hacen lo mismo todos ellos.

Polimorfismo

- El polimorfismo es la capacidad que tienen los objetos de diferentes clases de responder al mismo mensaje pero de diferente manera.
- Dependiendo del objeto receptor se ejecuta el método que corresponda, gracias al binding dinámico.

Polimorfismo

```
public class Automovil {  
    public void tocarLaBocina() { }  
}  
public class Bicicleta {  
    public void tocarLaBocina() { }  
}  
public class Barco {  
    public void tocarLaBocina() { }  
}
```

Diferentes clases pueden definir el mismo método. Aunque la implementación de este sea completamente diferente en cada clase. El polimorfismo puede aparecer aunque no se haya usado herencia.

```
Barco b=new Barco();  
Bicicleta bici=new Bicicleta();  
b.tocarLaBocina();  
bici.tocarLaBocina();
```

Aquí se aplica binding dinámico

Polimorfismo

El concepto de polimorfismo se aplica con mucha frecuencia cuando se programan jerarquías de clases y se usa herencia, ya que se pueden anular o redefinir los métodos heredados de la superclase.

C# proporciona una opción para anular el método de la clase base, agregando la palabra clave "virtual" al método de la superclase y utilizando la palabra clave "override" en el método de clase derivada.

Polimorfismo

```
class Animal // Clase Base (Padre)
{
    public virtual void animalSonido() {
        Console.WriteLine("El animal hace un sonido");
    }
}

class Gato : Animal // Clase Derivada (Hija)
{
    public override void animalSonido() {
        Console.WriteLine("El gato dice: miau miau");
    }
}

class Perro : Animal // Clase Derivada (Hija)
{
    public override void animalSonido() {
        Console.WriteLine("El perro dice: guau guau");
    }
}
```

```
class Program {
    static void Main(string[] args) {
        Animal un_animal = new Animal();
        Animal un_gato = new Gato();
        Animal un_perro = new Perro();

        un_animal.animalSonido();
        un_gato.animalSonido();
        un_perro.animalSonido();

        Console.ReadLine();
    }
}
```

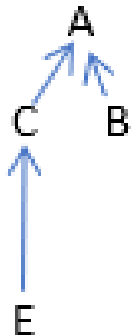
¿Qué imprime?

```
El animal hace un sonido
El gato dice: miau miau
El perro dice: guau guau
```

En este caso, el método de las clases derivadas anularon al método de la clase base.

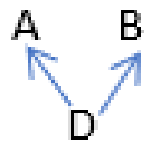
Tipos de Herencia

Herencia simple



cada subclase tiene una única superclase

Herencia múltiple



una subclase puede heredar de varias superclases. D hereda de A y de B.

Tipos de Herencia

Total: la subclase hereda TODOS los atributos y comportamiento de la superclase

Parcial: la subclase hereda algunos atributos y métodos de la superclase. Se restringen los atributos a heredar y se redefinen los métodos heredados que queremos cambiar su funcionalidad (con override).

C# tiene herencia simple y total: hereda todos los atributos de la superclase y todos los métodos, excepto el constructor. Si desea anular algún método heredado, lo define como polimórfico y le pone override.

Puesta en común Ejercicio 2 – TP 6

Cree una jerarquía de clases como la que se indica en el esquema y defina en todas ellas el método polimórfico `ToString()` que escribe en la consola la jerarquía desde la clase A hasta aquella a la que pertenece el objeto en forma invertida.

```
Clase A
| Clase B
| Clase C
| Clase D
```

Por ejemplo, si `obj` es una instancia de la clase D,

```
obj.ToString() deberá imprimir: Clase D   Clase C   Clase B
Clase A
```

```
public class classA
{
    public classA()
    {
    }
    public virtual void toString()
    {
        Console.WriteLine("Clase A");
    }
}
```

```
public class classB:classA
{
    public classB()
    {
    }
    public override void toString()
    {
        Console.WriteLine("Clase B");
        base.toString();
    }
}
```

```
public class classC:classB
{
    public classB()
    {
    }
    public override void toString()
    {
        Console.WriteLine("Clase C");
        base.toString();
    }
}
```



```
public static void Main(string[] args)
{
    claseA var1=new claseA();
    claseB var2=new claseB();
    claseC var3=new claseC();

    var2.toString();
    var3.toString();

    Console.ReadKey(true);
}
```

Puesta en común

Dentro de la jerarquía de Mascota:

¿Hay algún método que pueda ser polimórfico?

```
public class Mascota{  
    protected string nombre;  
    protected Dueño dueño;  
    protected int edad;  
  
    public Mascota(string nom){  
        nombre = nom;  
    }  
    public Mascota(string nom, Dueño dueño, int e){  
        nombre = nom;  
        this.dueño = dueño;  
        edad = e;  
    }  
    public virtual void hablarConElDueño(); {  
        Console.WriteLine("estoy hablando");}  
}
```

El método hablarConElDueño puede ser polimórfico.
Primera opción:
En la clase Mascota lo declaramos como virtual y así puede ser redefinido en la subclase.

```
abstract class Mascota{
    protected string nombre;
    protected Dueño dueño;
    protected int edad;

    public Mascota(string nom){
        nombre = nom;
    }
    public Mascota(string nom, Dueño dueño, int e){
        nombre = nom;
        this.dueño = dueño;
        edad = e;
    }
    public abstract void hablarConElDueño();
}
```

Segunda opción:
En la clase Mascota lo
declaramos como abstracto, ya
que esta clase dice que toda
Mascota debería hablar con su
dueño pero no como debe
hacerlo.

```
abstract class Mascota{  
    protected String nombre;  
    protected Dueño dueño;  
    protected int edad;  
  
    public Mascota(string nom){  
        nombre = nom;  
    }  
    public Mascota(string nom, Dueño dueño, int e){  
        nombre = nom;  
        this.dueño = dueño;  
        edad = e;  
    }  
    public abstract void hablarConElDueño();  
}
```

Al tener un método abstracto
debemos declarar la clase como
abstracta

¿Cómo se implementa el método hablarConElDueño en cada una de las subclases?

```
public class Perro : Mascota{
    override public void hablarConElDueño(){
        Console.WriteLine("Estoy ladrando");
    }
}

public class Gato : Mascota{
    override public void hablarConElDueño(){
        Console.WriteLine("Estoy maullando");
    }
}

public class Tortuga : Mascota{
    override public void hablarConElDueño(){
        Console.WriteLine("Soy un animal que no habla");
    }
}

}
```

