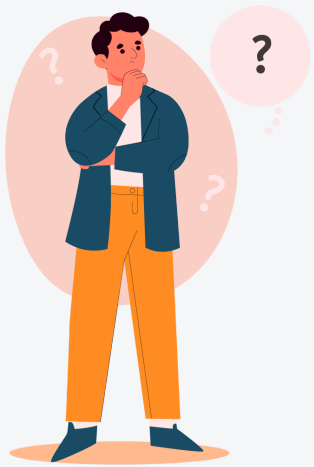
4to Pilar: Polimorfismo

Sitio: Agencia de Habilidades para el Futuro
Curso: Desarrollo de Sistemas Orientado a Objetos 1° D
Libro: 4to Pilar: Polimorfismo

Imprimido por: Eduardo Moreno
Día: miércoles, 14 de mayo de 2025, 23:09

Tabla de contenidos

Preguntas orientadoras



- Tengo la clase padre animal, y las subclases perro y gato. En la clase padre tengo un método hacerSonido().
- ¿Puedo realizar una herencia sin problemas?
- ¿Cómo se hace para que el perro haga su sonido “Guau”?
- ¿Cómo se hace para que el gato haga su sonido “Miau”?
- ¿Cómo se hace para redefinir el método hacerSonido() en las diferentes clases herederas?

Introducción

En este módulo vamos a desarrollar el concepto de polimorfismo que está estrechamente ligado al concepto de herencia.

¿Por qué es importante este pilar? En principio, consideramos que es posible implementar polimorfismo en jerarquías de clases que se dan a través de la herencia.

Pero, además, nos permite escribir un código genérico que pueda trabajar con diferentes tipos de objetos sin necesidad de conocerlos en detalle. Esto mejora la reutilización del código y facilita la extensibilidad del sistema.

	**Un ejemplo común es el uso de polimorfismo de método en lenguajes de programación como C#.**

Supongamos que tenemos una clase llamada Animal con un método "hacerSonido". Luego puedes crear subclases como Perro, Gato y Vaca que heredan de la clase Animal y cada una de ellas puede implementar su propio método "hacerSonido" de acuerdo con el sonido característico de cada animal. Luego, puedes tratar a cada objeto como un objeto de tipo Animal y llamar al método "hacerSonido", y el comportamiento específico se determinará en tiempo de ejecución según el tipo real del objeto.

El polimorfismo es la capacidad de los lenguajes orientados a objetos de tolerar diferentes tipos de datos en una misma variable, de tal manera que una referencia a una clase (atributo, parámetro o declaración local o elemento de un vector) acepta referencias de objetos de dicha clase y de sus clases derivadas (hijas, nietas,...).

A continuación, programemos el ejemplo.

Programando el ejemplo

A continuación, podrán visualizar un ejemplo de cómo implementar el concepto de hacerSonido con polimorfismo en C#:

Tenemos la clase Animal:

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

La clase Perro que hereda de Animal ya que "ES" un animal

```
// Subclase Perro
class Perro : Animal
{
    public override void HacerSonido()
    {
        Console.WriteLine("El perro ladra");
    }
}
```

La clase Gato que hereda de Animal ya que "ES" un animal

```
// Subclase Gato
class Gato : Animal
{
    public override void HacerSonido()
    {
        Console.WriteLine("El gato maulla");
    }
}
```

La clase Vaca que hereda de Animal ya que "ES" un animal

```
// Subclase Vaca
class Vaca : Animal
{
    public override void HacerSonido()
    {
        Console.WriteLine("La vaca muge");
    }
}
```

Finalmente, nuestra clase Program o Test que tiene 3 variables del tipo Animal pero que la instancia según sea el caso

Luego puede invocar al método hacerSonido ya que cada uno lo implementó "a su manera" pero en definitiva, todos los animales hacen un sonido.

```
class Program
{
    static void Main(string[] args)
    {
        Animal animal1 = new Perro();
        Animal animal2 = new Gato();
        Animal animal3 = new Vaca();

        animal1.HacerSonido(); // Salida: El perro ladra
        animal2.HacerSonido(); // Salida: El gato maulla
        animal3.HacerSonido(); // Salida: La vaca muge
    }
}
```

Para poder ampliar el concepto, pensemos este pilar dentro del sistema.

Sistema de tipos de datos

Para ampliar nuestra comprensión sobre el polimorfismo conviene primero repasar lo que significa el sistema de tipos de datos.

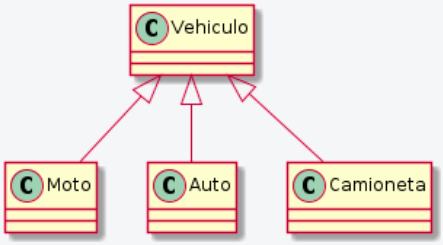
Acerca del tipo de datos

En los lenguajes de programación fuertemente tipados, cuando se define una variable, siempre hay que decir el tipo de datos que va a contener esta variable. Por ejemplo: int numero;

De esta manera, se indica que la variable declarada "numero" va a contener siempre un entero. Se pueden asignar diversos valores, pero siempre deben ser números enteros. De lo contrario el compilador dará un mensaje de error y no permitirá compilar el programa.

Esto que se acaba de describir pasa incluso con los objetos.

Por ejemplo, si se tiene una clase vehículo de la cual heredan clases hijas como auto, moto, camioneta:



Si se define una variable que hace referencia a un objeto de clase Auto, durante toda la vida de esa variable tendrá que contener un objeto de la clase Auto, no pudiendo más adelante hacer referencia a un objeto de la clase Moto o de la clase Camioneta.

//la variable miAuto apunta a un objeto de la clase Auto

```
Auto miAuto  =  new Auto("Ford Fiesta");
```

//luego puede apuntar a otro objeto diferente, pero siempre tendrá que ser de la clase Auto

```
miAuto = new Auto("Volkswagen Gol");
```

Lo que nunca se podrá hacer es guardar en esa variable, declarada como tipo Auto, otra cosa que no sea un objeto que pertenezca a la clase Auto.

Una función cuyo parámetro se haya declarado de una clase, sólo aceptará recibir objetos de esa clase. Asimismo, un array que se ha declarado con elementos de una clase determinada, sólo aceptará que se asignen objetos de esa clase declarada.

```
Auto[] misAutos = new Auto[3];
misAutos[0] = new Auto("Citroen");
```

Esa variable misAutos es un array y en ella se ha declarado que el contenido de las posiciones serán objetos de la clase Auto.

Si se define un array con elementos de una determinada clase, el compilador también aceptará que se asigne en sus posiciones objetos de una clase hija de la que fue declarada.

Si se declara una función que recibe como parámetros objetos de una determinada clase, el compilador también acepta que se le pasa en la llamada a esa función objetos de una clase derivada de aquella que fue declarada.

A continuación, trabajemos en un ejemplo concreto.

Ejemplo Sistema de tipos de datos

Se puede definir entonces un array de vehículos y gracias al polimorfismo se puede asignar en los elementos del array no solo vehículos genéricos, sino también todos los objetos de clases hijas o derivadas de la clase Vehiculo, es decir, objetos de las clases Auto, Moto, Camioneta o cualquier hija que se haya definido.

```
Vehiculo[] misVehiculos = new Vehiculo[4];
misVehiculos[0] = new Vehiculo("Generico");
misVehiculos[1] = new Auto("Fiat Palio");
misVehiculos[2] = new Moto("Yamaha YBR");
```

```
misVehiculos[3] = new Camioneta("VW Amarok");
```

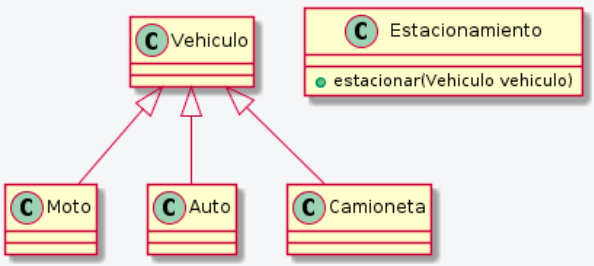
Como se mencionó, esta tolerancia del sistema de tipos se aplica también a los parámetros de métodos y funciones.

Supongamos que creamos la clase Estacionamiento y dentro de ella se define el método void estacionar(Vehiculo v). Este método se encarga de ubicar y estacionar el vehículo que recibe como parámetro.

Si no existiera el polimorfismo se tendría que crear varios métodos con diferentes parámetros, uno por cada tipo de vehículo para permitir estacionar objetos de la clase Auto, otro método que acepte objetos de la clase Moto, otro para Camioneta, etc.

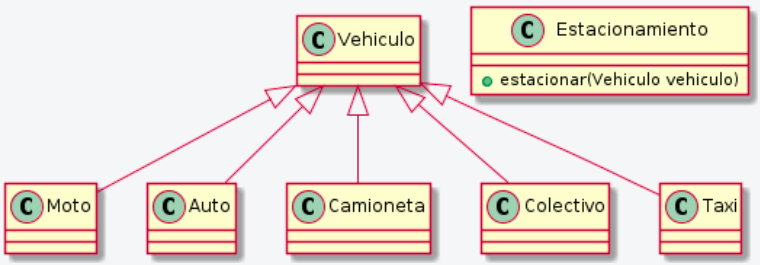
Lo ideal es que el método permita recibir todo tipo de vehículos para estacionarlos, en primer lugar, por reutilización del código, ya que es muy parecido estacionar uno u otro vehículo, pero además porque así si en el futuro se debe incorporar otro tipo de vehículos, como un colectivo, un cuatriciclo, o una nave espacial, el software será capaz de aceptarlos sin tener que modificar la clase Estacionamiento.

Entonces, cuando se declara el método estacionar() se puede definir que reciba como parámetro un objeto de la clase Vehículo y el compilador aceptará no solamente vehículos genéricos, sino todos aquellos objetos que hayamos creado que hereden de la clase vehículo, es decir, autos, motos, camionetas, etc. Esa tolerancia del sistema de tipos para aceptar una gama de objetos diferente es lo que llamamos **polimorfismo**.




```
Vehiculo[] misVehiculos = new Vehiculo[4];
misVehiculos[0] = new Vehiculo("Generico");
misVehiculos[1] = new Auto("Fiat Palio");
misVehiculos[2] = new Moto("Yamaha YBR");
misVehiculos[3] = new Camioneta("VW Amarok");
Extacionamiento parking = new Estacionamiento("Mi Estacion");
foreach(Vehiculo v in misVehiculos) {
      parking.estacionar(v);
}
```

Incluso, si en el futuro se necesita aceptar otro tipo de vehículo, no es necesario que se modifique la clase Estacionamiento ni el método "estacionar()". Simplemente agregando una clase hija a la clase Vehículo el método estacionar() lo aceptará.



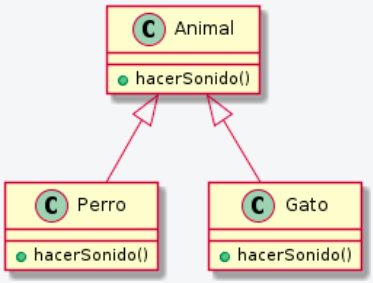

```
Vehiculo[] misVehiculos = new Vehiculo[6];
misVehiculos[0] = new Vehiculo("Generico");
misVehiculos[1] = new Auto("Fiat Palio");
misVehiculos[2] = new Moto("Yamaha YBR");
misVehiculos[3] = new Camioneta("VW Amarok");
misVehiculos[4] = new Taxi("VW Suran Taxi");
misVehiculos[5] = new Colectivo("Mercedes Benz");
Extacionamiento parking = new Estacionamiento("Mi Estacion");
foreach(Vehiculo v in misVehiculos) {
       parking.estacionar(v);
}
```

Sobreescritura y Sobrecarga

Es interesante el uso de **polimorfismo** junto con la capacidad de **sobrescribir** (override) y **sobrecargar métodos**.

Volvamos al ejemplo de la introducción, donde creamos dos clases distintas: Gato y Perro, que heredan de la superclase Animal.

La clase Animal tiene el método hacerSonido() que se implementa de forma distinta en cada una de las subclases (gatos y perros hacen sonidos distintos). Entonces, es posible llamar al método de hacer sonido a un grupo de objetos Gato y Perro por medio de una variable de referencia de clase Animal, haciendo así un uso polimórfico de dichos objetos respecto del mensaje hacerSonido().



Veamos un ejemplo de uso de virtual y override:

```
using System;
namespace animales
{
    public class Animal
    {
        protected string nombre;
        public virtual void hacerSonido()
        {
            Console.WriteLine(".        ");
        }
        public Animal(string n)
        {
            nombre = n;
        }
    }
```

```
public class Perro : Animal
{
    public Perro(string n):base(n){}
    public override  void  hacerSonido()
    {
        Console.WriteLine(nombre + ": " + "GUAU!");
```

```
    }
  }
  public class Gato : Animal
  {
    public Gato(string n):base(n){}
    public override void hacerSonido()
    {
      Console.WriteLine(nombre + ": " + "MIAU!");
    }
  }
}
```

Como todos los objetos Gato y Perro "son" objetos de la clase Animal, podemos crear dos variables de referencia de tipo Animal y las apuntamos a los objetos Gato y Perro. Luego, podemos llamar a los métodos hacerSonido().

```
using System;
namespace animales
{
    public class Program
    {
        static void Main(string[] args)
        {
            Animal mascota1 = new Perro("Odie");
            Animal mascota2 = new Gato("Garfield");
```

```
      mascota1.hacerSonido();
      mascota2.hacerSonido();
    }
```

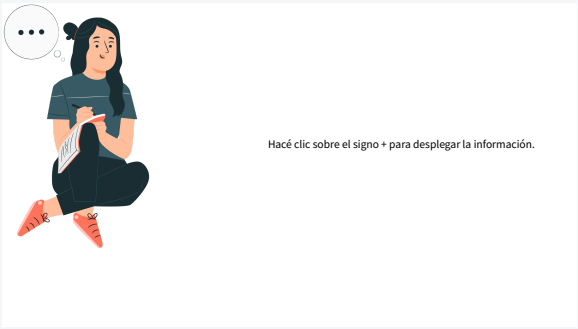
```
  }
}
```

Virtual y override

En C#, la palabra clave `virtual` se utiliza para modificar un método en una clase base para permitir que sea sobrescrito en las clases derivadas.

Cuando un método se declara como `virtual`, indica que puede ser redefinido en las clases derivadas utilizando la palabra clave `override` .

Aquí hay algunos puntos clave sobre el uso de `virtual` en C#:



En nuestro ejemplo, la clase base Animal tiene un método `HacerSonido()` marcado como `virtual` . Esto permite que las clases derivadas, como Perro y Gato, sobrescriban el método para proporcionar su propia implementación específica.

Es importante destacar que `virtual` y `override` trabajan en conjunto para lograr el polimorfismo, permitiendo que los métodos sean sobrescritos en las clases derivadas y que la implementación adecuada sea llamada en tiempo de ejecución según el tipo real del objeto.

En resumen

En resumen, el polimorfismo en C# es un concepto clave de la programación orientada a objetos que permite a los objetos de diferentes clases ser tratados de manera uniforme a través de una interfaz común. **Aquí están los puntos clave sobre el polimorfismo en C#:**

Hacé clic sobre el signo + para desplegar la información.

Powered by genially EDUCATION

Es importante recordar que `virtual` y `override` trabajan en conjunto para lograr el polimorfismo, permitiendo que los métodos sean sobrescritos en las clases derivadas y que la implementación adecuada sea llamada en tiempo de ejecución según el tipo real del objeto.