Clases y métodos abstractos

Sitio: Agencia de Habilidades para el Futuro
Curso: Desarrollo de Sistemas Orientado a Objetos 1° D
Libro: Clases y métodos abstractos

Imprimido por: Eduardo Moreno
Día: lunes, 19 de mayo de 2025, 00:46

Tabla de contenidos

1. Preguntas orientadoras

2. Introducción

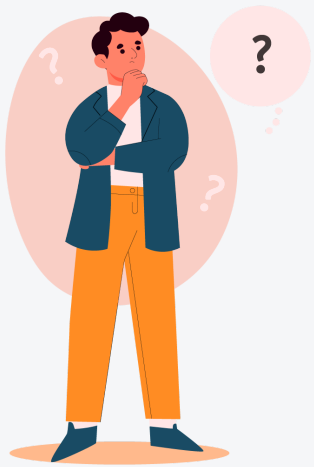
3. Encabezado de Clase Abstracta
3.1. Implementando métodos abstractos
3.2. Analizando métodos abstractos
3.3. Codificando clases y métodos
3.4. Continuamos codificando
3.5. Probemos nuestro sistema

4. Especialización

5. Recordar!!!!

6. En resumen

Preguntas orientadoras



- ¿Cómo se restringe la creación de una clase base para que solo se pueda heredar y no instanciar?
- ¿Cómo modelo una clase dejando solamente alguno métodos implementados y obligo a que las clases herederas los implementen?
- ¿Cómo se define una clase abstracta en C# y cuál es la diferencia con una clase normal?
- ¿Cuál es la importancia de los métodos abstractos en una clase abstracta?
- ¿Cómo se define un método abstracto en C# y qué restricciones deben cumplir?
- ¿Qué ocurre si una clase hereda de una clase abstracta pero no implementa todos los métodos abstractos?

Introducción

En este módulo vamos a definir y diferenciar una clase abstracta.

Una clase abstracta es una clase que no puede tener instancias, es decir, que no se pueden crear objetos a partir de ella. Pueden servir de base para definir una subclase que sí puede instanciarse siempre que ésta subclase no sea también abstracta.

Las clases abstractas generalmente contienen métodos abstractos pero también pueden contener métodos comunes (implementados). Basta que se declare al menos un método como abstracto, para que la clase que lo contiene sea abstracta.

Sin embargo, una clase puede ser abstracta aún teniendo todos sus métodos implementados.

¿A qué nos referimos con "método abstracto"? Un método abstracto es un método que solamente se declara, no tiene implementación. Y esta implementación se hará luego en las clases derivadas para que cada clase lo "haga" a "su manera" (polimorfismo)

El método abstracto debe estar definido en una clase (abstracta) y nunca puede ser privado.

Para que quede definida la clase se utiliza la palabra clave 'abstract' antes de la definición de la clase o método

Comencemos con la lectura del módulo ...

Declarando una Clase Abstracta y métodos abstractos

Visualicemos el encabezado de una clase abstracta

Observá las palabras abstract en la declaración de la clase y de los métodos:

```
abstract class Animal
{
    ...
```

```
   public abstract void hacerSonido(); // solo se declara, no se implementa

   public abstract void caminar(); // solo se declara, no se implementa

   ...

}
```

Notemos que este código no permitirá instanciar un Animal y que los métodos **hacerSonido** y **caminar** los deberá implementar cada una de las clases derivadas que hereden de la clase Animal.

A continuación, definamos el método.

Implementando métodos abstractos

Las clases hijas pueden o no implementar el método. Si lo implementan, entonces esta nueva clase hija puede instanciar objetos con el operador new normalmente (ver las clases Perro y Gato del código que sigue).

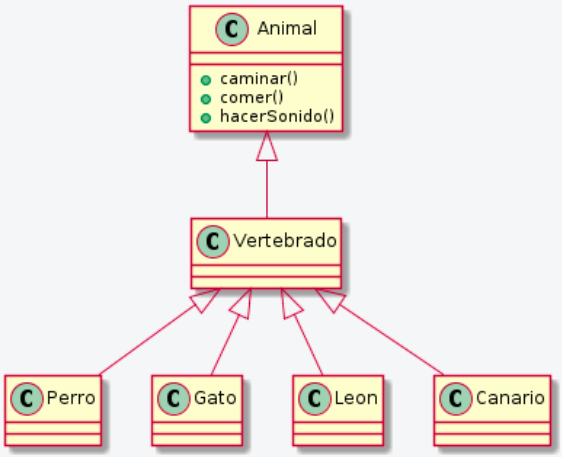
Si las clases hijas no implementan o desarrollan todos los métodos abstractos de la superclase, entonces esta nueva clase también será abstracta y no podrán instanciarse objetos (ver la clase Vertebrado del código que sigue).

Cuando una clase derivada se crea, primero se llama al constructor de la clase base antes de ejecutar su propio constructor. Esto asegura que se realice la inicialización necesaria en la clase base antes de continuar con cualquier operación adicional en la clase derivada.

Para manejar los constructores en la **herencia** en C#, se utiliza la palabra clave `base` para llamar al constructor de la clase base. La llamada al constructor de la clase base se realiza en el bloque de código del constructor de la clase derivada, como la primera instrucción que se ejecuta.

Analicemos un ejemplo

Se define una clase Animal que tiene los métodos caminar y comer. Esta clase es la clase base para Vertebrados e Invertebrados. Luego heredan otras clases, como Perro, Gato, León y Canario. Todas ellas van a heredar de nuestra clase Animal con sus respectivos métodos y se tendrá la posibilidad de crear objetos de cada uno de estos animales, pero no de la superclase Animal ni de Vertebrados e Invertebrados.

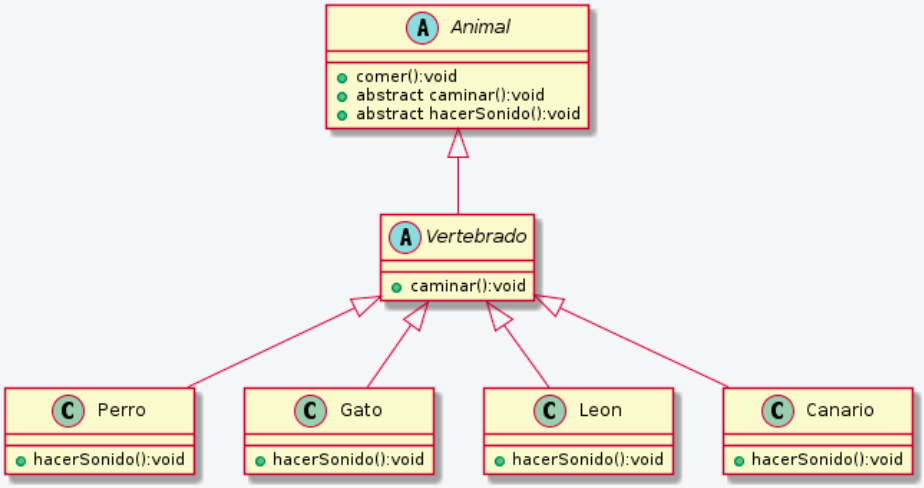


Cada animal puede comportarse de manera similar y realizar las mismas acciones, como caminar, comer, pero el sonido emitido no será igual en todos ellos.

Analizando métodos abstractos

No tendría sentido implementar el método hacerSonido() en la clase base ya que no hay un sonido asociado a la clase Animal, ni tampoco a la clase Vertebrado. Por lo tanto este método será abstracto y será compartido por las clases que hereden de nuestra clase Animal, que será abstracta. Es decir, la clase Animal no podrá ser instanciada (no se puede hacer new Animal()), de hecho, no hay ningún elemento en el mundo real que sea una instancia de Animal sin pertenecer a ninguno de los grupos derivados, todos los animales que existen son instancias de un tipo particular de animal.

Las clases abstractas, entonces, se utilizan en aquellos casos en los que existe un conjunto de acciones comunes a todas las subclases pero que su implementación depende de cada subclase particular.



No se podrán crear instancias de la clase Animal ni de Vertebrados, solo se pueden instanciar cualquiera de las subclases.

Codificando clases y métodos

Si codificamos la clase Animal nos queda de la siguiente manera:

```
using System;
using System.Collections.Generic;
using System.Text;

namespace Veterinaria
{
    6 referencias
    abstract class Animal
    {
        private string nombre;
        1 referencia
        public Animal(string n)
        {
            nombre = n;
        }

        5 referencias
        public string Nombre
        {
            get { return nombre; }
            private set { nombre = value; }
        }

        /* ...
        5 referencias
        public abstract void hacerSonido();

        /* ...
        2 referencias
        public abstract void caminar();
        1 referencia
        public void comer()
        {
            Console.WriteLine(nombre + ": comiendo... ");
        }
    }
}
```

El método hacerSonido solo se declara, no se implementa ya que cada animal lo hará a su manera
El método caminar solo se declara, no se implementa. La implementación la haremos en Vertebrados ya que si tuviésemos una clase Invertebrados, estos caminan de otra forma.

La clase Vertebrado nos queda:

```
using System;
using System.Collections.Generic;
using System.Text;

namespace Veterinaria
{
    9 referencias
    abstract class Vertebrado : Animal
    {
        4 referencias
        public Vertebrado(string n) : base(n) { }

        2 referencias
        public override void caminar()
        {
            Console.WriteLine(Nombre + ": caminando... ");
        }
    }
}
```

Continuemos...

Continuamos codificando...

Para avanzar con nuestro código de ejemplo, ahora vamos a programar cada uno de los animales de la veterinaria.

Iniciamos con Perro:

```
using System;
using System.Collections.Generic;
using System.Text;

namespace Veterinaria
{
    4 referencias
    class Perro : Vertebrado
    {
        3 referencias
        public Perro(string n) : base(n) { }
        2 referencias
        public override void hacerSonido()
        {
            Console.WriteLine(Nombre + ": GUAU!");
        }
    }
}
```

A continuación, programamos la clase Gato:

```
using System;
using System.Collections.Generic;
using System.Text;

namespace Veterinaria
{
    2 referencias
    class Gato : Vertebrado
    {
        1 referencia
        public Gato(string n) : base(n) { }
        2 referencias
        public override void hacerSonido()
        {
            Console.WriteLine(Nombre + ": MIAU");
        }
    }
}
```

Agregamos Canario:

```
using System;
using System.Collections.Generic;
using System.Text;

namespace Veterinaria
{
    2 referencias
    class Canario : Vertebrado
    {
        1 referencia
        public Canario(string n) : base(n) { }
        2 referencias
        public override void hacerSonido()
        {
            Console.WriteLine(Nombre + ": PIO-PIO");
        }
    }
}
```

Y otra clase Leon:

```
using System;
using System.Collections.Generic;
using System.Text;

namespace Veterinaria
{
    2 referencias
    class Leon : Vertebrado
    {
        1 referencia
        public Leon(string n) : base(n) { }
        2 referencias
        public override void hacerSonido()
        {
            Console.WriteLine(Nombre + ": GRRRRRR");
        }
    }
}
```

Veamos a continuación como probar nuestro sistema.

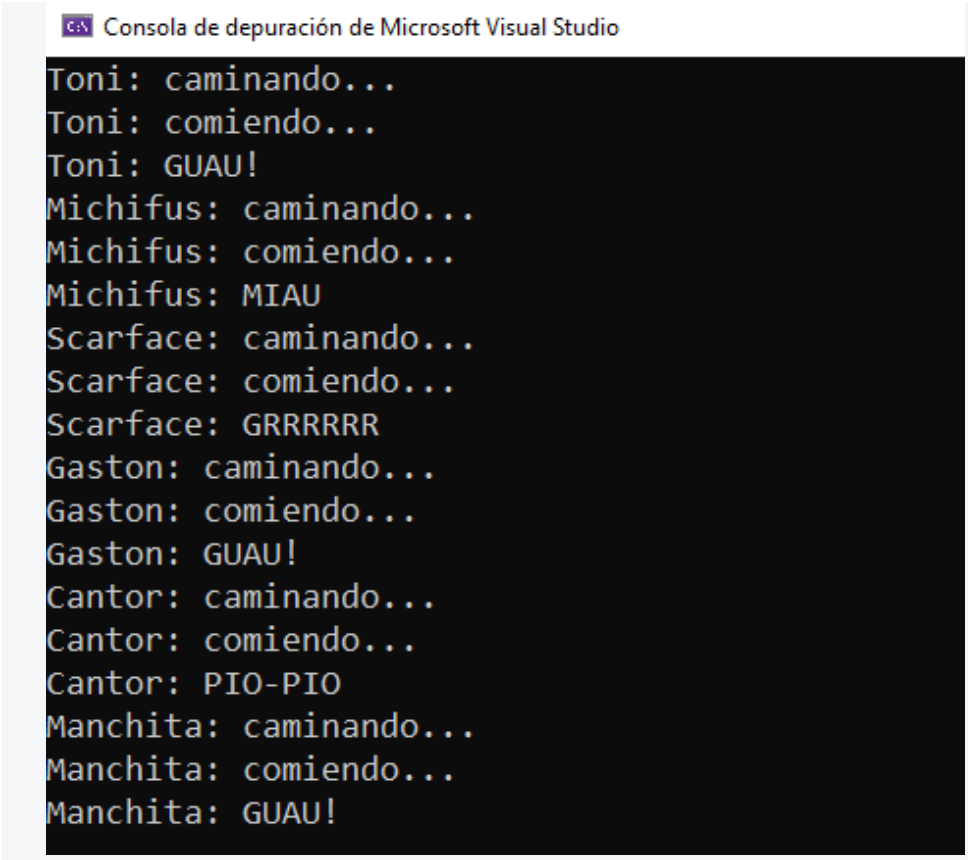
Probemos nuestro sistema

Para finalizar programaremos la clase Test para probar nuestro sistema:

```
namespace Veterinaria
{
    0 referencias
    internal class Test
    {
        0 referencias
        static void Main(string[] args)
        {
            {
                List<Animal> animales = new List<Animal>();
                animales.Add(new Perro("Toni"));
                animales.Add(new Gato("Michifus"));
                animales.Add(new Leon("Scarface"));
                animales.Add(new Perro("Gaston"));
                animales.Add(new Canario("Cantor"));
                animales.Add(new Perro("Manchita"));

                //Recorremos cada uno de los animales de la veterinaria
                foreach (Animal animal in animales)
                {
                    animal.caminar();
                    animal.comer();
                    animal.hacerSonido();
                }
            }
        }
    }
}
```

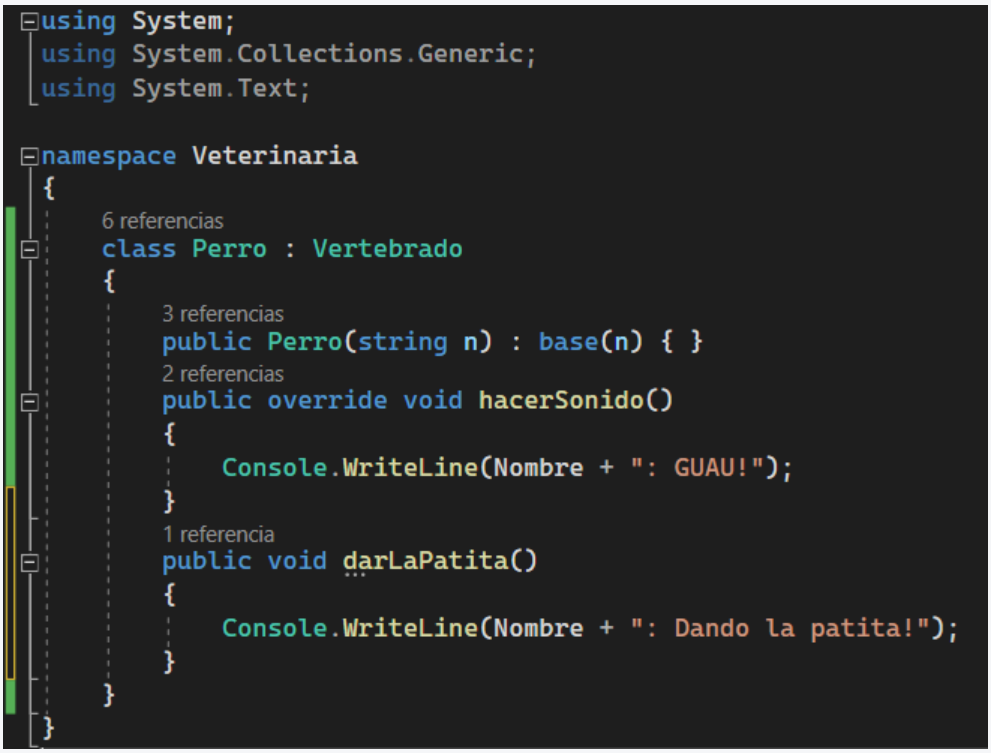
Luego, al ejecutar nuestro programa observamos que tenemos una salida como la siguiente:


Consola de depuración de Microsoft Visual Studio
Toni: caminando...
Toni: comiendo...
Toni: GUAU!
Michifus: caminando...
Michifus: comiendo...
Michifus: MIAU
Scarface: caminando...
Scarface: comiendo...
Scarface: GRRRRRR
Gaston: caminando...
Gaston: comiendo...
Gaston: GUAU!
Cantor: caminando...
Cantor: comiendo...
Cantor: PIO-PIO
Manchita: caminando...
Manchita: comiendo...
Manchita: GUAU!

Especialización

Siguiendo con el ejemplo de la veterinaria, supongamos que el perro sabe dar la patita y ningún otro animal lo puede hacer.

Es decir, este método estará implementado solamente en la clase Perro, quedando el código de la siguiente manera:



```
using System;
using System.Collections.Generic;
using System.Text;

namespace Veterinaria
{
    6 referencias
    class Perro : Vertebrado
    {
        3 referencias
        public Perro(string n) : base(n) { }
        2 referencias
        public override void hacerSonido()
        {
            Console.WriteLine(Nombre + ": GUAU!");
        }
        1 referencia
        public void darLaPatita()
        {
            Console.WriteLine(Nombre + ": Dando la patita!");
        }
    }
}
```

Ahora bien, al momento de recorrer todos los animales de la veterinaria, si quisiéramos además pedirles que nos den la patita, los Animales no saben hacerlo, con lo cual tenemos que pedírselo sólo al Animal que haya sido instanciado como Perro. Esto se averigua preguntando `if (animal.GetType() == typeof(Perro))`

Si nos dice que si, entonces a ese animal hay que decirle que se comporte como Perro. A esto se lo denomina realizar un casteo o castear una variable. Es decir que una variable "se comporte" como si fuese de otro tipo de dato.

Nuestro ejemplo queda de la siguiente manera:

```
namespace Veterinaria
{
    0 referencias
    internal class Test
    {
        0 referencias
        static void Main(string[] args)
        {
            {
                List<Animal> animales = new List<Animal>();
                animales.Add(new Perro("Toni"));
                animales.Add(new Gato("Michifus"));
                animales.Add(new Leon("Scarface"));
                animales.Add(new Perro("Gaston"));
                animales.Add(new Canario("Cantor"));
                animales.Add(new Perro("Manchita"));

                //Recorremos cada uno de los animales de la veterinaria
                foreach (Animal animal in animales)
                {
                    animal.caminar();
                    animal.comer();
                    animal.hacerSonido();
                    if (animal.GetType() == typeof(Perro))
                        ((Perro)animal).darLaPatita();
                }
            }
        }
    }
}
```

Y la salida al momento de ejecutar nuestro programa:

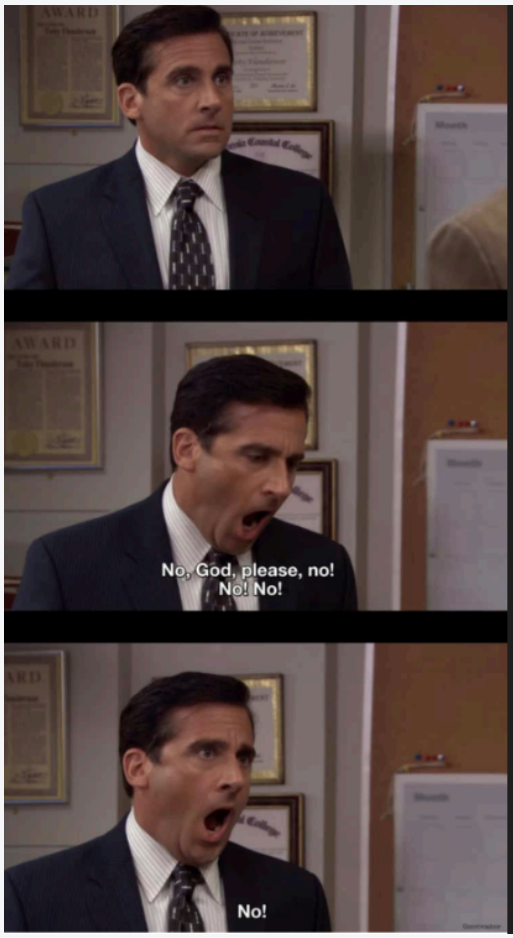
```
Consola de depuración de Microsoft Visual Studio
Toni: caminando...
Toni: comiendo...
Toni: GUAU!
Toni: Dando la patita!
Michifus: caminando...
Michifus: comiendo...
Michifus: MIAU
Scarface: caminando...
Scarface: comiendo...
Scarface: GRRRRRR
Gaston: caminando...
Gaston: comiendo...
Gaston: GUAU!
Gaston: Dando la patita!
Cantor: caminando...
Cantor: comiendo...
Cantor: PIO-PIO
Manchita: caminando...
Manchita: comiendo...
Manchita: GUAU!
Manchita: Dando la patita!
```

Observemos que solamente da la patita aquellos animales que son perros.

Solo utilizaremos **GetType** cuando necesitemos ejecutar un método específico de una clase derivada.

Nunca lo haremos para métodos que fueron o bien creados como abstractos en las clases padre, o bien para métodos concretos que fueron implementados en la clase padre.

Recordar!




```
foreach (Animal animal in animales)
{
    switch (animal.GetType().Name)
    {
        case "Perro":
            ((Perro)animal).caminar();
            ((Perro)animal).comer();
            ((Perro)animal).hacerSonido();
            break;
        case "Gato":
            ((Gato)animal).caminar();
            ((Gato)animal).comer();
            ((Gato)animal).hacerSonido();
            break;
        case "Leon":
            ((Leon)animal).caminar();
            ((Leon)animal).comer();
            ((Leon)animal).hacerSonido();
            break;
        case "Canario":
            ((Canario)animal).caminar();
            ((Canario)animal).comer();
            ((Canario)animal).hacerSonido();
            break;
```

En resumen

Como ya hemos señalado, el polimorfismo es uno de los conceptos centrales en los lenguajes de POO. Describe la capacidad de usar diferentes clases con la misma interfaz. Cada una de estas clases puede proporcionar su propia implementación de la interfaz.

Varios lenguajes, entre ellos C#, admiten dos tipos de polimorfismo:

Polimorfismo estático	Polimorfismo dinámico
Puede sobrecargar un método con diferentes conjuntos de parámetros. Esto se llama polimorfismo estático porque el compilador une estáticamente la llamada al método a un método específico (visto en la semana 3).	Dentro de una jerarquía de herencia, una subclase puede anular un método de su superclase. Si instancias la subclase, siempre llamará al método anulado, incluso si lanzas la subclase a su superclase. Eso se llama polimorfismo dinámico (visto en la semana 10).

En esta semana aprendimos **métodos y clases abstractas en C#.** Aquí están los puntos claves para recordar:

- Una clase abstracta es una clase que no se puede instanciar directamente.
- Se utiliza como base para otras clases, que pueden proporcionar implementaciones para los métodos abstractos de la clase abstracta.
- Las clases abstractas se utilizan para proporcionar una base común para un grupo de clases relacionadas, y para evitar la duplicación de código.
- Para crear una clase abstracta, se utiliza la palabra clave `abstract` en la declaración de la clase.

Algunos ejemplos de cuándo utilizar clases abstractas:

- Cuando se desea crear una jerarquía de clases, donde algunas clases son más generales que otras.
- Cuando se desea proporcionar una implementación común para un grupo de métodos.
- Cuando se desea evitar la duplicación de código.

Las clases abstractas son una herramienta poderosa que puede ayudar a hacer que el código sea más organizado y fácil de mantener.