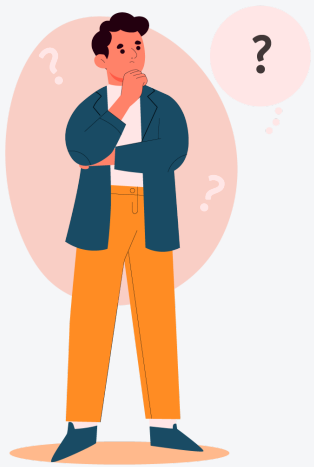
Sobreescritura de métodos

Sitio: Agencia de Habilidades para el Futuro
Curso: Desarrollo de Sistemas Orientado a Objetos 1º D
Libro: Sobreescritura de métodos

Imprimido por: Eduardo Moreno
Día: lunes, 5 de mayo de 2025, 11:13

Tabla de contenidos

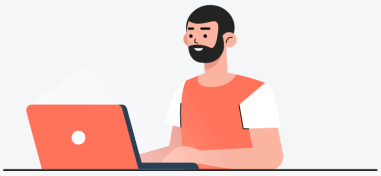
Preguntas orientadoras



- ¿Qué es la sobreescritura de métodos?
- ¿Cuál es la diferencia entre utilizar new y override en una sobreescritura de métodos?
- ¿Cómo puede una clase hija implementar su propia forma de hacer las cosas si es que no nos sirve la forma en que lo hace la clase padre?
- ¿Cómo inicializo atributos de la clase padre si necesito crear un objeto de la clase hija?
- ¿Cuál es el orden de ejecución de los constructores en una jerarquía de herencia?

Recapitulando

Antes de arrancar, **¡repasamos!**



La clase anterior aprendimos que:

- La herencia nos permite crear jerarquías de clases donde las clases derivadas heredan características y comportamientos de una clase base. Uno de los aspectos fundamentales de la herencia en C# es la capacidad de sobreescritura de métodos y constructores en las clases derivadas.
- A su vez, la sobreescritura de métodos nos permite redefinir un método heredado de la clase base en una clase derivada, lo que nos brinda la flexibilidad de personalizar su comportamiento para adaptarlo a las necesidades específicas de la clase derivada.
- Por otro lado, el manejo de constructores en la herencia nos asegura una correcta inicialización de los objetos en una jerarquía de clases. Al crear una instancia de una clase derivada, primero se llama al constructor de la clase base para realizar la inicialización necesaria en la clase base. Luego, se ejecuta el constructor de la clase derivada, donde podemos proporcionar una lógica adicional y personalizada.

Ahora sí, veamos cómo estos conceptos se ponen en juego en esta semana.

Introducción

Como vimos la herencia es un concepto fundamental en la programación orientada a objetos que nos permite crear jerarquías de clases y compartir características comunes entre ellas.

En el contexto de la herencia, la sobreescritura de métodos y el manejo de constructores son dos conceptos clave que nos permiten personalizar el comportamiento de las clases derivadas y garantizar una correcta inicialización de los objetos.

Constructores en clases derivadas

Ampliando saberes sobre los constructores

En C#, los constructores en herencia son utilizados para garantizar una correcta inicialización de los objetos en una jerarquía de clases.

Los constructores, tal como hemos podido ver en la clase anterior, permiten establecer los valores iniciales de los atributos de las clases derivadas, así como invocar los constructores de la clase base para inicializar los atributos heredados.

Repasemos la dinámica

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.

	**Te dejamos un ejemplo para ilustrar cómo se manejan los constructores en la herencia en C#:**

```
class Animal
{
    private string nombre;
    public Animal(string nombre)
permiten establecer los valores iniciize: large;">    {
        this.nombre = nombre;
        Console.WriteLine("Se creó un animal llamado: " + nombre);
    }
}
class Perro : Animal
{
    public Perro(string nombre) : base(nombre)
    {
        Console.WriteLine("Se creó un perro llamado: " + getNombre();
    }
}
class Program
{
    static void Main(string[] args)
    {
        Perro perro = new Perro("Firulais");
        // Salida:
        // Se creó un animal llamado: Firulais
        // Se creó un perro llamado: Firulais
    }
}
```

Veamos en detalle el ejemplo

Análisis del ejemplo

En el ejemplo anterior, tenemos una clase base `Animal` con un constructor que recibe un parámetro `nombre` . En la clase derivada `Perro`, utilizamos: `base(nombre)` para llamar al constructor de la clase base, pasando el parámetro `nombre` correspondiente.

Al crear una instancia de `Perro` en el método `Main`, se llamará primero al constructor de la clase base `Animal` a través de la llamada `: base(nombre)` . Luego, se ejecutará el constructor de la clase derivada `Perro`. Esto asegura que el objeto `Perro` esté correctamente inicializado tanto en la clase base como en la clase derivada.

Es importante tener en cuenta que la llamada al constructor de la clase base debe ser la primera instrucción en el bloque de código del constructor de la clase derivada. Esto garantiza que se realice la inicialización adecuada en la jerarquía de herencia antes de continuar con cualquier otra operación en la clase derivada.

El manejo de los constructores en la herencia en C# permite una inicialización coherente y controlada de los objetos en una jerarquía de clases, asegurando que los atributos tanto de la clase base como de las clases derivadas estén correctamente establecidos.

A continuación, desglosemos este tema con los métodos en las clases derivadas.

Métodos en las clases derivadas

Nuestro módulo se denomina "Sobreescritura de métodos", pero, ¿por qué? En principio porque es el eje de nuestro recorrido: la sobreescritura de los métodos es un mecanismo que nos permite redefinir un método heredado de la clase base en una clase derivada.

Un método heredado

Abordarlo de esta manera, nos brinda la flexibilidad de adaptar y modificar el comportamiento de un método para satisfacer las necesidades específicas de la clase derivada.

Al utilizar la palabra clave "override" (como lo hacemos en el método ToString), indicamos explícitamente que estamos sobrescribiendo un método.

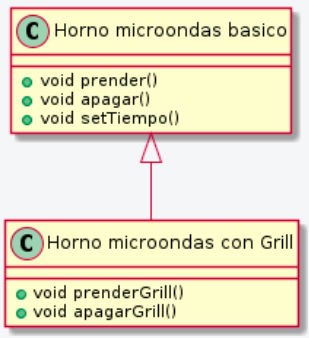
A través de la sobreescritura de métodos, podemos implementar lógica adicional, cambiar el comportamiento por completo o extender la implementación original de la clase base.

¡Algo importante! Este concepto es fundamental para el polimorfismo que veremos más adelante, donde podemos tratar los objetos de diferentes clases derivadas de manera uniforme a través de una referencia a la clase base.

Ventajas en una clase derivada

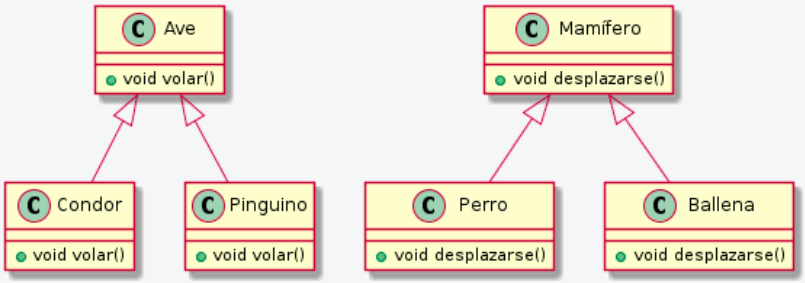
En una clase derivada (hija) se puede:

Añadir nuevos métodos/atributos propios de la clase derivada: es decir se le agregan nuevas características a la nueva clase, como por ejemplo el método ladrar() en una clase perro cuando ésta hereda métodos de Ser Vivo como respirar().



El modelo con grill de cierta marca es igual al básico, salvo que se le agrega el grill.

Modificar los métodos heredados de la clase base: el método heredado se redefine completamente, si bien se "llama" igual, su comportamiento puede ser muy distinto.

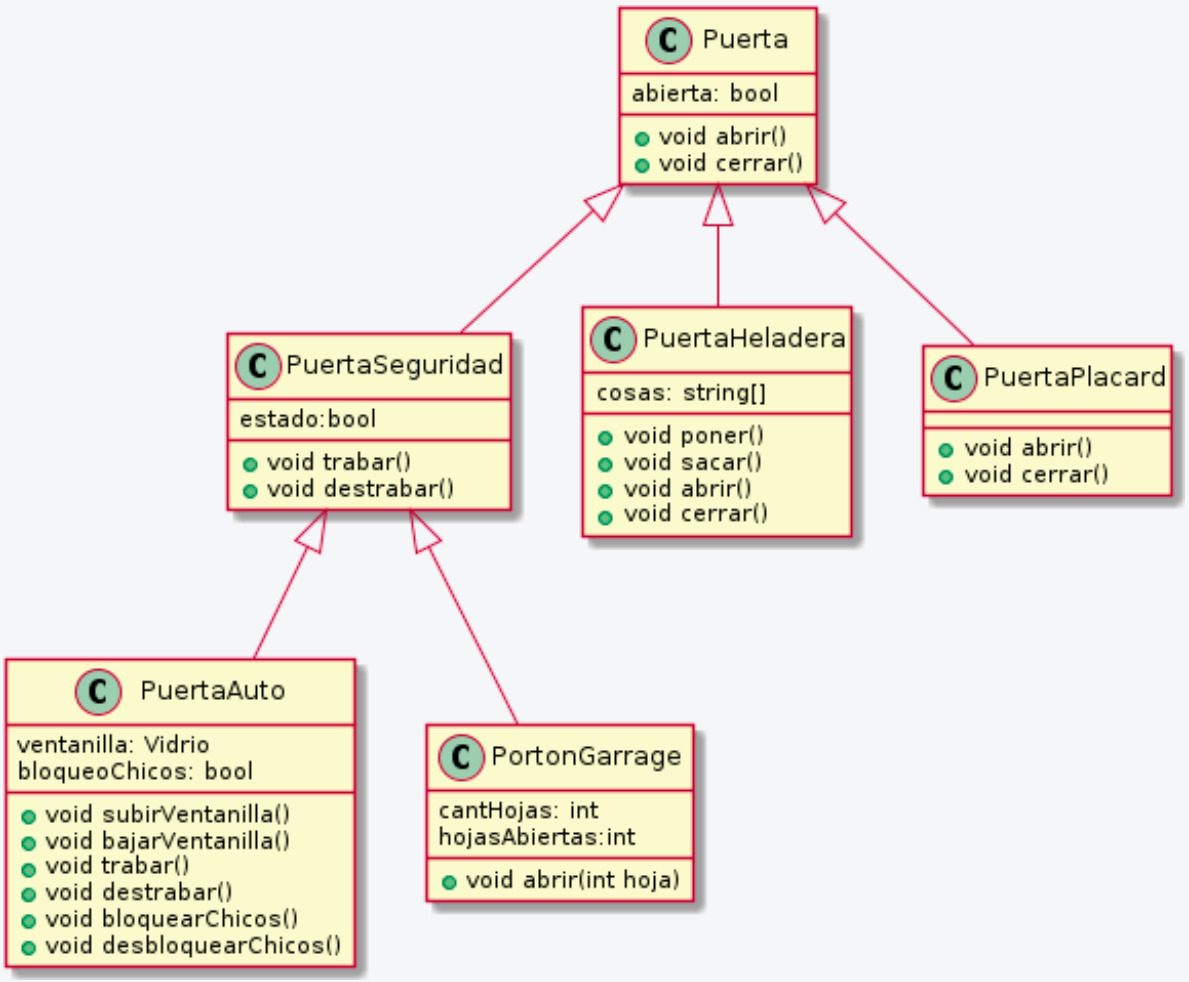


La capacidad de volar y la forma es muy distinto entre un cóndor (muy alta capacidad) y un pingüino (no vuela) a pesar que ambos son aves. La forma de desplazarse de un perro (camina, corre..) es muy distinta a la de una ballena (nada).

A continuación, abordemos desde un ejemplo de herencia simple.

Herencia simple en C#

En el siguiente ejemplo se muestra un diagrama UML donde se define una clase llamada Puerta que solo puede "abrir" y "cerrar" y desde donde varias otras clases heredan de ella sus propiedades, pero agregándole alguna característica. Puerta Seguridad representa una puerta que tiene algún tipo de cerradura mientras que PuertaHeladera que no tiene ningún tipo de seguridad pero sí puede guardar tener cosas (huevos, botellas de agua, aderezos.) en sus estantes. PuertaAuto tiene una ventanilla a la que se puede bajar y subir mientras qué Porton Garage puede tener varias hojas (se puede abrir una hoja para que pase una persona, o 2 hojas para que pase un auto).



En primer lugar se define la clase Puerta; observar que se le agregaron algunos métodos y atributos más que el diagrama, como un "nombre" para poder diferenciar una puerta de otra puerta.

```
class Puerta
{
    bool abierta;
    public void cerrar()        {...}
    public override string ToString() {...}
    public Puerta(string nom)   {...}
    protected string getNombre()        {...}
}
```

```
    string nombre;
```

```
    public void abrir() {...}
```

Para definir la clase hija PuertaSeguridad se escribe el nombre de la clase hija "dos puntos" nombre de la clase padre

```
class PuertaSeguridad : Puerta
{
...
}
```

Al agregar los dos puntos (:) y el nombre de la clase padre, la nueva clase PuertaSeguridad hereda todos los atributos y métodos de Puerta, y además agrega un nuevo atributo "estado" y varios métodos más.

```
class PuertaSeguridad : Puerta
{
    bool estado; // true si esta trabada, false si no lo esta.
    public PuertaSeguridad(string n) :base(n)   {...}
    public void trabar()        {...}
    public void destrabar()     {...}
}
```

Así, en un hipotético main se puede declarar y crear un objeto llamado p del tipo Puerta donde se observa que se llama a los métodos abrir y cerrar. Líneas más abajo se crea otro objeto del tipo PuertaSeguridad y se observa que no solo se puede llamar a los métodos abrir y cerrar definidos en la clase padre Puerta sino que también puede ejecutar los nuevos métodos trabar y destrabar.

```
static void Main(string[] args)
{
    var p = new Puerta("Comedor");
    p.abrir();
    p.cerrar();
    Console.WriteLine(p);
    var pSeg = new PuertaSeguridad("Tranquera");
    pSeg.abrir();
    pSeg.cerrar();
    pSeg.trabar();
    pSeg.destrabar();
    Console.WriteLine(pSeg);
    ...
```

```
}
```

Redefinición de métodos - new

¿Qué nos permite la sobreescritura de métodos?

La sobreescritura de métodos permite redefinir un método que se hereda para que este funcione de acuerdo a nuevas necesidades y no a lo definido en la superclase.

Cuando en un objeto se llama a un método el compilador comprueba si el método existe en nuestro objeto, si existe lo usa y si no existe en nuestro objeto entonces lo busca en la superclase. Esto ocurre así hasta que el compilador encuentra el método definido. El compilador busca el método de "abajo a arriba".

Retomemos el ejemplo de la clase PuertaSeguridad, la idea es que si la puerta está trabada (porque tiene un pasador, porque tiene puesto un cerrojo, está cerrada con llave, etc.) no pueda abrirse.

En este caso se va a redefinir el método abrir para que controle si está trabada o no.

Hay que observar que el nombre del método, los parámetros y lo que devuelve no cambian (sino sería una sobrecarga).

Por ejemplo la clase Puerta define un método abrir() que solo cambiar el valor del atributo abierta, pero luego es redefinido en la clase PuertaSeguridad donde hay un nuevo método abrir() que antes de abrir la puerta comprueba si está trabada o no.

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

namespace ConsoleApp9
{
    3 referencias
    internal class Puerta
    {
        private bool abierta;
        protected string nombre;
        1 referencia
        public Puerta (string nombre)
        {
            this.nombre = nombre;
        }
        1 referencia
        public void abrir()
        {
            abierta = true;
            Console.WriteLine(nombre + ":Abriendo");
        }
        0 referencias
        public virtual void cerrar()
        {
            abierta = false;
            Console.WriteLine(nombre + ":Cerrando");
        }
        1 referencia
        public string getNombre()
        {
            return this.nombre;
        }
    }
}
```

Clase PuertaSeguridad

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

namespace ConsoleApp9
{
    2 referencias
    internal class PuertaSeguridad : Puerta
    {
        bool estado; // true si está trabada, false si no lo esta.
        1 referencia
        public PuertaSeguridad(string nombre):base(nombre)
        {
            this.estado = true;
        }
        2 referencias
        public new void abrir()
        {
            if (!estado)
                base.abrir(); // abre efectivamente la puerta
            else
                Console.WriteLine(getNombre() + ": No se puede abrir, esta trabada");
        }
        1 referencia
        public void trabar()
        {
            estado = true;
        }

        1 referencia
        public void destrabar()
        {
            estado = false;
        }
    }
}
```

En un supuesto Main se podría comprobar que la tranquera no se abrirá si está trabada.

```
using System;

namespace ConsoleApp9
{
    0 referencias
    internal class Program
    {
        0 referencias
        static void Main(string[] args)
        {
            var pSeg = new PuertaSeguridad("Tranquera");
            Console.WriteLine(pSeg);
            pSeg.trabar();
            pSeg.abrir();


            Console.WriteLine(pSeg);
            pSeg.destrabar();
            pSeg.abrir();
            Console.WriteLine(pSeg);
        }
    }
}
```

A continuación, veamos la diferencia entre new y override

Diferencia entre new y override

La diferencia principal entre "**new**" y "**override** " se encuentra en cómo se utilizan y su implicación en la herencia en la programación orientada a objetos.

01 **new**: La palabra clave "new" se utiliza para ocultar un miembro heredado, ya sea un método, propiedad o campo, en una clase derivada. Al utilizar "new", se crea una nueva implementación del miembro en la clase derivada sin tener en cuenta el miembro heredado de la clase base. Esto se conoce como ocultamiento de miembros. La ocultación puede ser útil cuando deseas proporcionar una implementación completamente diferente o adicional para el mismo nombre de miembro en la clase derivada. Sin embargo, ten en cuenta que el miembro ocultado de la clase base no es accesible a través de la clase derivada, lo que puede generar cierta confusión.

02 **override**: La palabra clave "override" se utiliza para indicar que un método de una clase derivada está reemplazando (sobrescribiendo) un método heredado de la clase base. Al usar "override", estás proporcionando una nueva implementación del método que tiene el mismo nombre y firma en la clase derivada. Esto permite que la clase derivada utilice su propia lógica y comportamiento para ese método específico, pero aún conserva la relación de herencia y el polimorfismo. Al utilizar "override", puedes acceder al comportamiento de la clase base mediante la palabra clave "base", lo que te permite extender o modificar la implementación original.

En resumen, "new" se utiliza para ocultar un miembro heredado y proporcionar una nueva implementación independiente, mientras que "override" se utiliza para reemplazar un método heredado con una nueva implementación que mantiene la relación de herencia.

La elección entre "new" y "override" dependerá de tus necesidades específicas y del diseño de tus clases y relaciones de herencia.

Este tema lo pondremos en práctica en las próximas semanas cuando veamos **polimorfismo**.

En resumen

Recorramos algunos puntos que vimos en este libro:

Sobreescritura de métodos:

La sobreescritura de métodos en C# nos permite redefinir un método heredado de la clase base en una clase derivada.

Utilizamos la palabra clave override para indicar que un método en la clase derivada está reemplazando el método heredado de la clase base.

Al sobrescribir un método, podemos proporcionar una nueva implementación que se adapte a las necesidades específicas de la clase derivada.

La sobreescritura de métodos nos permite utilizar polimorfismo (lo veremos en la próxima semana), lo que significa que podemos tratar los objetos de diferentes clases derivadas de manera uniforme a través de una referencia a la clase base.

Sobreescritura de constructores:

Al utilizar herencia en C#, podemos invocar el constructor de la clase base desde el constructor de la clase derivada utilizando la palabra clave base .

La llamada al constructor de la clase base debe ser la primera instrucción en el bloque de código del constructor de la clase derivada.

El constructor de la clase base se ejecuta antes de ejecutar cualquier operación adicional en el constructor de la clase derivada.

Al utilizar la llamada a base, aseguramos que se realice la inicialización necesaria en la clase base antes de continuar con la lógica específica de la clase derivada.

A modo de conclusión

La sobreescritura de métodos y constructores en **herencia** en C# nos brinda la capacidad de personalizar el comportamiento de los métodos y la inicialización de objetos en una jerarquía de clases. Esto nos permite adaptar y extender la funcionalidad de las clases base en las clases derivadas, proporcionando una mayor flexibilidad y reutilización de código en nuestras aplicaciones.