Escuelita Crombie 3er Edición

Índice

Introducción

En esta unidad, se introducen los conceptos básicos de la programación orientada a objetos (OOP) en C#. La OOP es un paradigma que organiza el código en estructuras que se asemejan a objetos del mundo real, mejorando la capacidad de manejar aplicaciones de mayor complejidad. Este enfoque es ampliamente aceptado en el mundo de la programación, siendo C# un lenguaje ideal para su implementación.

Los cuatro principios básicos de OPP son:

Abstracción: modelar los atributos e interacciones pertinentes de las entidades como clases para definir una representación abstracta de un sistema.

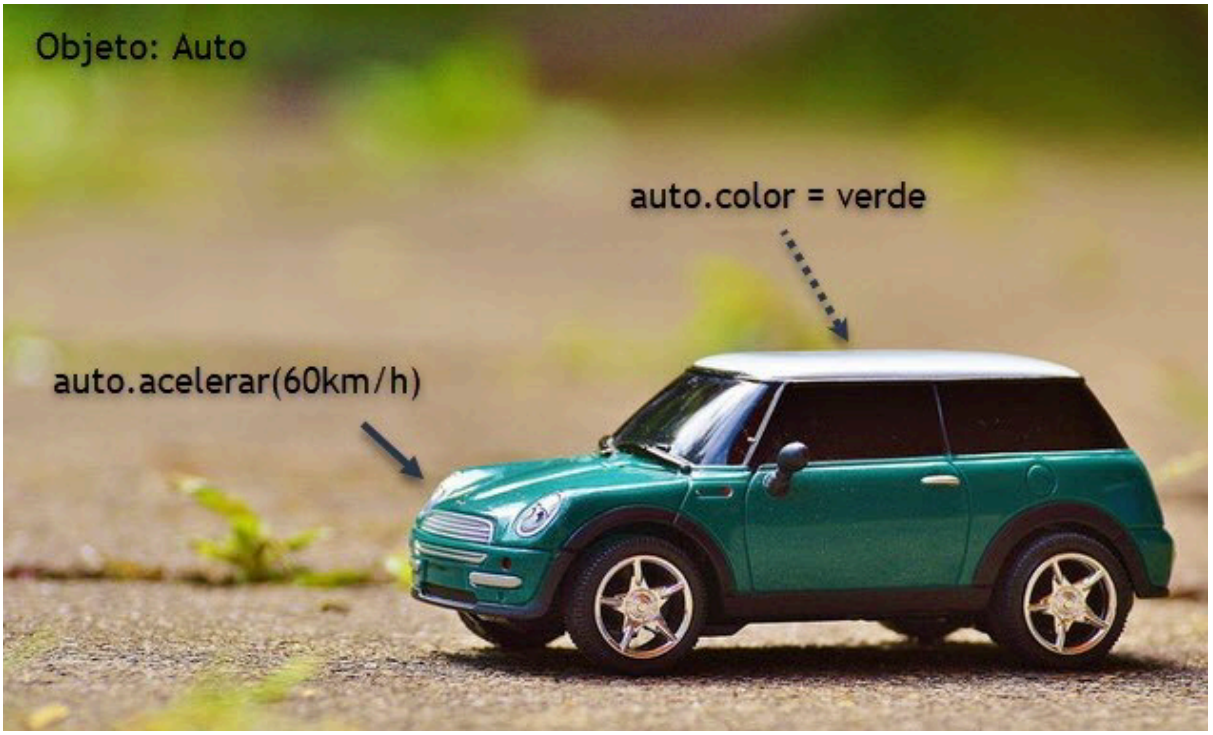
Encapsulación: ocultar el estado interno y la funcionalidad de un objeto y permitir solo el acceso a través de un conjunto público de funciones.

Herencia: capacidad de crear nuevas abstracciones basadas en abstracciones existentes.

Polimorfismo: capacidad de implementar propiedades o métodos heredados de maneras diferentes en varias abstracciones.

Programación Orientada a Objetos

La programación orientada a objetos permite estructurar el código en términos de **objetos**, los cuales tienen propiedades y métodos. Un ejemplo clásico es el de un automóvil: el auto representa un objeto con propiedades (color, marca, modelo) y métodos (acelerar, frenar). La OOP permite modelar problemas y aplicaciones con un enfoque lógico y modular.



Las Clases

Las clases son el "molde" o la estructura base para crear objetos. En una clase, se definen:

- **Propiedades**: Características de los objetos, definidas como variables (ej., color, tipo, marca).
- **Métodos**: Acciones disponibles en los objetos, definidas como funciones (ej., Acelerar, Frenar).

Ejemplo de una clase básica en C#:

```
public class Auto
{
    public int puertas;
    public string color;
    private int velocidad;

    public void Acelerar(int velocidadNueva)
    {
        this.velocidad = velocidadNueva;
    }
}
```

Los Objetos

Un objeto es una instancia de una clase, creado con la palabra clave `new`. Al crear un objeto, sus propiedades y métodos están disponibles para manipulación y consulta. La **instancia** se puede modificar y acceder mediante el uso de métodos y asignación de valores.

Los vamos a utilizar para guardar datos y realizar acciones

Ejemplo:

```
Auto miAuto = new Auto();
miAuto.color = "verde";
miAuto.Acelerar(80);
```

Estado y Mensajes de un Objeto

- **Estado**: Representa los valores actuales de las propiedades de un objeto.
- **Mensajes**: Llamadas a los métodos del objeto que permiten invocar acciones.

Encapsulamiento

El encapsulamiento es una de las claves de la OOP, que permite controlar el acceso a las propiedades y métodos de un objeto. Usando modificadores de acceso (`public`, `private`), se define qué elementos de un objeto son accesibles desde fuera de su clase. Este enfoque permite modificar la implementación interna sin afectar otras partes del programa.

Ejemplo de encapsulamiento:

```
public class Auto
{
    private int velocidad;

    public void Acelerar(int velocidadNueva)
    {
        this.velocidad = velocidadNueva;
    }
}
```

Constructores

Los constructores son métodos especiales que se llaman al crear una nueva instancia de una clase. Permiten inicializar los valores de las propiedades de un objeto. Un constructor tiene el mismo nombre que la clase y no devuelve ningún valor.

Ejemplo de un constructor en C#:

```
public class Auto
{
    private string color;
    private int año;

    public Auto(string color, int año)
    {
        this.color = color;
        this.año = año;
    }
}
```

Getters y Setters

Para mantener el encapsulamiento, se utilizan métodos *getters* y *setters*, que permiten acceder y modificar propiedades privadas de un objeto de manera controlada.

Ejemplo de Getter y Setter:

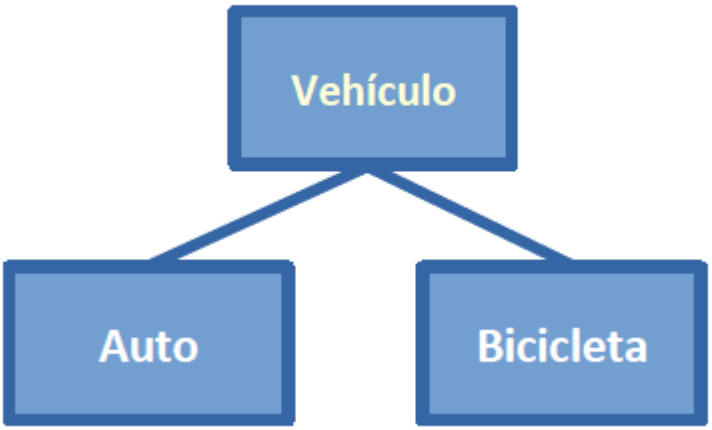
```
public class Auto
{
    private string color;

    public void SetColor(string color)
    {
        this.color = color;
    }

    public string GetColor()
    {
        return color;
    }
}
```

Resumen

Hasta aquí, hemos cubierto los conceptos de clases, objetos, encapsulamiento, constructores y *getters* y *setters*. La OOP permite crear múltiples instancias de objetos a partir de una clase, cada uno con características propias. En los próximos puntos, exploraremos cómo las clases pueden interactuar entre sí, permitiendo construir sistemas más complejos.

Herencia

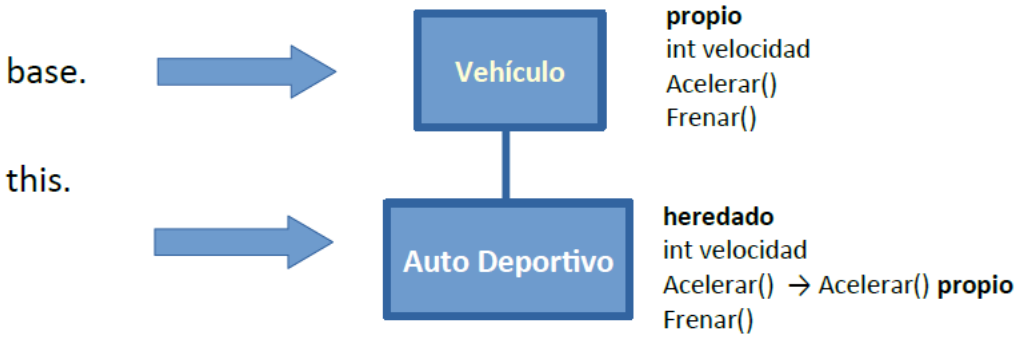
La herencia permite definir una nueva clase a partir de una clase existente, promoviendo la reutilización del código y la organización jerárquica de clases.



- **Definición**: Una clase "hija" hereda propiedades y métodos de una clase "padre", lo cual agrupa clases relacionadas y permite la especialización de características.
- **Palabras clave**:
    - `protected`: Permite el acceso a las propiedades y métodos solo en la clase heredada.
    - `virtual`: Marca un método en la clase padre para que pueda ser sobrescrito.
    - `override`: Sobrescribe un método `virtual` en una clase hija para ajustar su comportamiento.

Ejemplo de código:

```
public class Vehiculo {
    protected int velocidad;
    public virtual void Acelerar(int nuevaVelocidad) { /* ... */ }
}
public class AutoDeportivo : Vehiculo {
    public override void Acelerar(int nuevaVelocidad) {
        velocidad = nuevaVelocidad * 2;
    }
}
```



Asociación

La asociación define cómo los objetos interactúan entre sí y se clasifica en dos tipos:

- **Agregación**: Relación débil donde un objeto usa otro temporalmente. Ejemplo: Una `Persona` que usa un `Auto`.
- **Composición**: Relación fuerte donde un objeto depende de otro para existir. Ejemplo: Un `Auto` que tiene un `Motor`.

Ejemplo de Agregación:

```
public class Persona {

    private AutoDeportivo auto;

    public void AdjudicarVehiculo(AutoDeportivo auto) {

        this.auto = auto;

    }

}
```

Ejemplo de Composición:

```
public class AutoDeportivo {

    private Motor motor = new Motor(220);

    public void Acelerar(int velocidad) {

        motor.Acelerar(velocidad);

    }

}
```

Clases Estáticas

Las clases estáticas no se pueden instanciar y sus métodos pueden llamarse directamente. Son ideales para agrupar funciones auxiliares o "helpers".

Ejemplo de uso:

```
public static class Utilidades {

    public static void FormatearFecha(DateTime fecha) { /* ... */ }

}
```

Clases Abstractas

Las clases abstractas contienen métodos que pueden o no estar implementados y no pueden instanciarse. Los métodos abstractos deben implementarse en las clases hijas.

Ejemplo de implementación:

```
public abstract class Vehiculo {

    public abstract void LlenarTanque(int cantidad);

}

public class Gasolero : Vehiculo {

    public override void LlenarTanque(int cantidad) { /* ... */ }

}
```

Interfaces

Las interfaces declaran métodos y propiedades sin implementarlos. Son contratos que las clases deben seguir, permitiendo la implementación de múltiples interfaces en una sola clase.

- **Diferencias con Clases Abstractas**:

- Las interfaces no contienen código, mientras que las clases abstractas sí pueden.
- En .NET, una clase no puede heredar de múltiples clases, pero puede implementar múltiples interfaces.

Ejemplo de una interfaz y su implementación:

```
public interface IAcelerador {

    void Acelerar(int nuevaVelocidad);

}


public class Bicicleta : IAcelerador {

    public void Acelerar(int nuevaVelocidad) { /* ... */ }

}
```