Clase 5: Programación Orientada a Objetos: Cuatro Pilares de la POO 🛠️

La **programación orientada a objetos** es un paradigma de programación que utiliza "objetos" para representar datos y métodos. Los cuatro pilares fundamentales de la POO son:

1. **Abstracción**
2. **Encapsulamiento**
3. **Herencia**
4. **Polimorfismo**

1. Abstracción 🎭

La abstracción se refiere a la capacidad de un objeto de ocultar sus detalles internos y mostrar solo las características relevantes al exterior. Permite simplificar el diseño de software al modelar clases y objetos que representan conceptos del mundo real.

Ejemplo:

- Imagina un coche. Al interactuar con un coche, no necesitas conocer cómo funciona su motor; solo necesitas saber cómo encenderlo, apagarlo y conducirlo.

2. Encapsulamiento 🔒

El encapsulamiento es el principio de restringir el acceso a algunos componentes de un objeto y solo permitir el acceso a través de métodos específicos. Esto ayuda a proteger los datos y evitar que sean modificados de manera accidental.

Ejemplo:

- En una clase `CuentaBancaria`, los atributos como el saldo son privados y solo pueden ser accedidos o modificados mediante métodos como `depositar()` o `retirar()`.

Implementación en Código:

```
class CuentaBancaria:
    def __init__(self, saldo_inicial):
        self.__saldo = saldo_inicial  # Atributo privado

    def depositar(self, cantidad):
        self.__saldo += cantidad

    def retirar(self, cantidad):
        if cantidad <= self.__saldo:
            self.__saldo -= cantidad
        else:
            print("Fondos insuficientes")

    def obtener_saldo(self):
        return self.__saldo
```

3. Herencia 🧬

La herencia permite que una clase derive de otra clase, heredando sus atributos y métodos. Esto facilita la reutilización del código y la creación de jerarquías de clases.

Ejemplo:

- Si tienes una clase base `Animal` y deseas crear clases específicas como `Perro` y `Gato`, estas clases heredan de `Animal`.

Implementación en Código:

```
class Animal:
    def hablar(self):
        raise NotImplementedError("Este método debe ser imple
mentado en la clase hija")

class Perro(Animal):
    def hablar(self):
        return "¡Guau!"

class Gato(Animal):
    def hablar(self):
        return "¡Miau!"
```

4. Polimorfismo 🌈

El polimorfismo permite que diferentes clases implementen métodos con el mismo nombre pero con comportamientos diferentes. Esto se logra a través de la herencia y la sobreescritura de métodos.

Ejemplo:

- Siguiendo con el ejemplo anterior, tanto `Perro` como `Gato` tienen el método `hablar`, pero cada uno implementa su propia versión.

Implementación en Código:

```
animales = [Perro(), Gato()]
for animal in animales:
    print(animal.hablar())  # Imprime "¡Guau!" y "¡Miau!"
```

Ejercicios Prácticos 📝

Ejercicio 1: Crear una Clase `Vehículo` 🚗

Objetivo: Implementar la clase `Vehículo` que contenga atributos como `marca`, `modelo` y `precio`. Crear clases derivadas para `Auto`, `Bicicleta` y `Camión`.

Solución:

```
class Vehículo:
    def __init__(self, marca, modelo, precio):
        self.marca = marca
        self.modelo = modelo
        self.precio = precio

class Auto(Vehículo):
    def __init__(self, marca, modelo, precio):
        super().__init__(marca, modelo, precio)

class Bicicleta(Vehículo):
    def __init__(self, marca, modelo, precio):
        super().__init__(marca, modelo, precio)

class Camión(Vehículo):
    def __init__(self, marca, modelo, precio):
        super().__init__(marca, modelo, precio)
```

Ejercicio 2: Implementar Encapsulamiento en `CuentaBancaria` 💵

Objetivo: Crear una clase `CuentaBancaria` con atributos privados y métodos para depositar y retirar dinero.

Solución:

```
class CuentaBancaria:
    def __init__(self, saldo_inicial):
        self.__saldo = saldo_inicial

    def depositar(self, cantidad):
        self.__saldo += cantidad

    def retirar(self, cantidad):
        if cantidad <= self.__saldo:
            self.__saldo -= cantidad
        else:
            print("Fondos insuficientes")

    def obtener_saldo(self):
        return self.__saldo

# Uso
cuenta = CuentaBancaria(1000)
cuenta.depositar(500)
print(cuenta.obtener_saldo())  # Saldo: 1500
cuenta.retirar(2000)  # Fondos insuficientes
```

Ejercicio 3: Demostrar Polimorfismo con `Animal` 🐾

Objetivo: Crear una lista de diferentes animales y llamar al método `hablar()`.

Solución:

```
animales = [Perro(), Gato()]
for animal in animales:
    print(animal.hablar())
```

Ejercicio 4: Consultar Disponibilidad de Vehículos 🚙

Objetivo: Crear una clase `Concesionaria` que maneje un inventario de vehículos y permita consultar su disponibilidad.

Solución:

```
class Concesionaria:
    def __init__(self):
        self.vehiculos = []

    def añadir_vehículo(self, vehículo):
        self.vehiculos.append(vehículo)

    def mostrar_vehículos_disponibles(self):
        for vehiculo in self.vehiculos:
            print(f"Marca: {vehiculo.marca}, Modelo: {vehicul
o.modelo}, Precio: {vehiculo.precio}")

# Uso
concesionaria = Concesionaria()
concesionaria.añadir_vehículo(Auto("Toyota", "Corolla", 2000
0))
concesionaria.mostrar_vehículos_disponibles()
```

Conclusión 🏁

Los cuatro pilares de la programación orientada a objetos son fundamentales para el desarrollo de software moderno. Comprender cómo aplicar estos principios te ayudará a escribir código más limpio, eficiente y fácil de mantener. ¡Continúa practicando y explorando el mundo de la POO! 🚀
