

poo

APUNTES



1 de enero de 2025

madrid-españa

WALTER ISAMEL SAGASTEGUI LESCANO

INDEX

[1. ENCAPSULAMIENTO 3](#_Toc203476699)

[2. HERENCIA 5](#_Toc203476700)

[3. POLIMORFISMO 9](#_Toc203476701)

[4. ABSTRACCION 10](#_Toc203476702)

[5. EJEMPLO QUE INTEGRA LOS 4 PILARE DE LA POO 12](#_Toc203476703)

# ENCAPSULAMIENTO

El **encapsulamiento** es uno de los **cuatro pilares de la Programación Orientada a Objetos (POO)**, junto con la herencia, la abstracción y el polimorfismo. Su propósito principal es **ocultar los detalles internos de un objeto** y exponer solo lo necesario para su uso desde el exterior.

**¿Qué es el encapsulamiento?**

Es el **proceso de agrupar los datos (atributos)** y el **comportamiento (métodos)** que operan sobre esos datos en una misma unidad (la clase), y **restringir el acceso directo** a algunos de los componentes del objeto.

**¿Por qué es útil?**

1. **Protección de datos:** impide que el estado interno del objeto sea modificado directamente desde fuera.
2. **Control del acceso:** puedes definir exactamente cómo se puede interactuar con el objeto.
3. **Facilita el mantenimiento:** si los detalles internos cambian, no afecta al código que usa el objeto.
4. **Mayor seguridad y consistencia:** puedes validar valores antes de asignarlos.

**¿Cómo se aplica?**

En lenguajes como Java, C# o Python, el encapsulamiento se aplica utilizando **modificadores de acceso**:

* private: acceso solo dentro de la misma clase.
* public: acceso desde cualquier lugar.
* protected: acceso desde la misma clase o subclases (y en algunos lenguajes, el mismo paquete o módulo).
* En Python, se simula con convenciones como \_privado o \_\_muy\_privado.

**Ejemplo en Java**

public class Persona {

// Atributos privados

private String nombre;

private int edad;

// Constructor

public Persona(String nombre, int edad) {

this.nombre = nombre;

this.edad = edad;

}

// Métodos públicos para acceder a los atributos (getters y setters)

public String getNombre() {

return nombre;

}

public void setNombre(String nombre) {

this.nombre = nombre;

}

public int getEdad() {

return edad;

}

public void setEdad(int edad) {

if (edad >= 0) { // validación

this.edad = edad;

}

}

}

Aquí, los atributos nombre y edad están **encapsulados** (son privados) y solo se puede acceder a ellos mediante los métodos públicos getNombre(), setNombre(), etc.

**Encapsulamiento en Python**

Python **no tiene modificadores de acceso** como private o public, pero se **simulan con convenciones**:

* \_atributo: se considera **protegido** (no debe usarse desde fuera, pero no está prohibido).
* \_\_atributo: se considera **privado** (Python aplica *name mangling* para dificultar el acceso externo).

**Ejemplo:**

class Persona:

def \_\_init\_\_(self, nombre, edad):

self.\_\_nombre = nombre # Atributo privado

self.\_\_edad = edad # Atributo privado

# Getter

def get\_nombre(self):

return self.\_\_nombre

# Setter

def set\_nombre(self, nombre):

self.\_\_nombre = nombre

# Getter

def get\_edad(self):

return self.\_\_edad

# Setter con validación

def set\_edad(self, edad):

if edad >= 0:

self.\_\_edad = edad

else:

print("Edad no válida")

# Uso

p = Persona("Juan", 30)

print(p.get\_nombre()) # Juan

p.set\_edad(-5) # Edad no válida

Si intentas acceder directamente: p.\_\_edad, dará error, porque está encapsulado.

En Python moderno, una forma **más elegante y "pythónica"** de aplicar encapsulamiento es usando **propiedades con decoradores**: @property, @<atributo>.setter.

**Encapsulamiento con @property en Python**

Este enfoque permite **usar el atributo como si fuera público**, pero en realidad estás usando métodos **getter y setter ocultos**, lo que mantiene la seguridad y flexibilidad del encapsulamiento.

**Ejemplo completo:**

class Persona:

def \_\_init\_\_(self, nombre, edad):

self.\_\_nombre = nombre

self.\_\_edad = edad

# Getter para nombre

@property

def nombre(self):

return self.\_\_nombre

# Setter para nombre

@nombre.setter

def nombre(self, nuevo\_nombre):

self.\_\_nombre = nuevo\_nombre

# Getter para edad

@property

def edad(self):

return self.\_\_edad

# Setter para edad con validación

@edad.setter

def edad(self, nueva\_edad):

if nueva\_edad >= 0:

self.\_\_edad = nueva\_edad

else:

print("Edad no válida")

# Uso

p = Persona("Lucía", 22)

print(p.nombre) # Accede como si fuera un atributo

p.edad = -5 # Edad no válida

p.edad = 30

print(p.edad) # 30

**¿Qué está pasando aquí?**

* @property convierte un método en un **getter "invisible"**.
* @<nombre>.setter define el **setter correspondiente**.
* Desde fuera de la clase, puedes hacer p.nombre y p.nombre = "Otro" como si accedieras directamente al atributo, pero en realidad estás llamando a métodos ocultos.

**Ventajas de @property**

* Código limpio y claro.
* Se parece al uso de atributos públicos, pero con control total.
* Puedes añadir validación sin cambiar la interfaz pública del objeto.

**Encapsulamiento en C#**

C# **sí tiene modificadores de acceso**, como private, public y protected, lo que hace el encapsulamiento muy similar a Java.

**Ejemplo:**

public class Persona

{

// Atributos privados

private string nombre;

private int edad;

// Constructor

public Persona(string nombre, int edad)

{

this.nombre = nombre;

this.edad = edad;

}

// Getter y Setter para 'nombre'

public string Nombre

{

get { return nombre; }

set { nombre = value; }

}

// Getter y Setter para 'edad' con validación

public int Edad

{

get { return edad; }

set

{

if (value >= 0)

{

edad = value;

}

}

}

}

**Uso:**

Persona p = new Persona("Ana", 25);

Console.WriteLine(p.Nombre); // Ana

p.Edad = -10; // No cambia nada

**Comparación rápida:**

| **Lenguaje** | **Encapsulamiento** | **Sintaxis** |
| --- | --- | --- |
| Python | Convención con \_\_atributo | Métodos get/set manuales |
| C# | Modificadores reales (private) | Propiedades { get; set; } |

# HERENCIA

La **herencia** es otro de los **principios fundamentales de la Programación Orientada a Objetos (POO)**. Vamos a explicarla paso a paso.

**¿Qué es la herencia?**

La **herencia** es un mecanismo que permite crear una **nueva clase (subclase o clase hija)** basada en una **clase existente (superclase o clase padre)**. La clase hija **hereda atributos y métodos** de la clase padre y puede:

* Usarlos tal como están.
* Modificarlos (sobreescribirlos).
* Agregar nuevos.

**¿Para qué sirve?**

* **Reutilizar código**.
* **Organizar jerarquías** lógicas (por ejemplo: Animal → Perro, Gato).
* **Aplicar el principio DRY (Don't Repeat Yourself)**.

**Ejemplo conceptual**

Supón que tienes una clase Animal con un método hablar(). Puedes crear una clase Perro que herede de Animal y sobrescriba ese método para que diga "Guau".

**Ejemplo en Python**

# Clase padre

class Animal:

def \_\_init\_\_(self, nombre):

self.nombre = nombre

def hablar(self):

return "Hace un sonido"

# Clase hija

class Perro(Animal):

def hablar(self):

return "Guau"

# Otra hija

class Gato(Animal):

def hablar(self):

return "Miau"

# Uso

a = Animal("Bicho")

p = Perro("Toby")

g = Gato("Mishi")

print(a.hablar()) # Hace un sonido

print(p.hablar()) # Guau

print(g.hablar()) # Miau

Aquí, Perro y Gato **heredan** el atributo nombre y el método hablar() de Animal, pero lo **sobrescriben** con su propio comportamiento.

**Ejemplo en C#**

// Clase base

public class Animal

{

public string Nombre { get; set; }

public Animal(string nombre)

{

Nombre = nombre;

}

public virtual string Hablar()

{

return "Hace un sonido";

}

}

// Clase derivada

public class Perro : Animal

{

public Perro(string nombre) : base(nombre) {}

public override string Hablar()

{

return "Guau";

}

}

// Otra derivada

public class Gato : Animal

{

public Gato(string nombre) : base(nombre) {}

public override string Hablar()

{

return "Miau";

}

}

**Uso:**

Animal a = new Animal("Bicho");

Perro p = new Perro("Toby");

Gato g = new Gato("Mishi");

Console.WriteLine(a.Hablar()); // Hace un sonido

Console.WriteLine(p.Hablar()); // Guau

Console.WriteLine(g.Hablar()); // Miau

**Palabras clave importantes**

| **Lenguaje** | **Clase Padre** | **Clase Hija** | **Sobrescribir método** |
| --- | --- | --- | --- |
| Python | class A: | class B(A): | Definir de nuevo método |
| C# | class A | class B : A | virtual y override |
| Java | class A | class B extends A | @Override |

Vamos a ver **herencia múltiple en Python**, **clases abstractas en C#** y un **mini ejercicio práctico** al final. Así tendrás una visión clara y completa. ¡Vamos paso a paso!

**Herencia múltiple en Python**

Python **sí permite herencia múltiple**, es decir, una clase puede heredar de **más de una clase base**.

**Ejemplo:**

class Volador:

def volar(self):

return "Estoy volando"

class Nadador:

def nadar(self):

return "Estoy nadando"

# Hereda de ambas

class Pato(Volador, Nadador):

def hablar(self):

return "Cuac cuac"

# Uso

p = Pato()

print(p.volar()) # Estoy volando

print(p.nadar()) # Estoy nadando

print(p.hablar()) # Cuac cuac

En este ejemplo, Pato hereda métodos de **dos clases base** (Volador y Nadador), sin necesidad de repetir código.

**Clases abstractas e interfaces en C#**

En C#, una **clase abstracta** es una clase que no se puede instanciar directamente y que puede contener **métodos sin implementar**. También existen **interfaces**, que son como contratos.

**Clase abstracta**

public abstract class Animal

{

public string Nombre { get; set; }

public Animal(string nombre)

{

Nombre = nombre;

}

public abstract string Hablar(); // método abstracto

}

**Clases hijas que la implementan**

public class Perro : Animal

{

public Perro(string nombre) : base(nombre) {}

public override string Hablar()

{

return "Guau";

}

}

public class Gato : Animal

{

public Gato(string nombre) : base(nombre) {}

public override string Hablar()

{

return "Miau";

}

}

**🔹 Uso:**

csharp

CopiarEditar

Animal p = new Perro("Rocky");

Animal g = new Gato("Luna");

Console.WriteLine(p.Hablar()); // Guau

Console.WriteLine(g.Hablar()); // Miau

Las clases hijas **deben implementar** el método Hablar() porque está definido como abstracto.

**Mini ejercicio práctico (Python y C#)**

**Enunciado:**

Crea una clase base Empleado con nombre y salario, y dos clases hijas Gerente y Desarrollador. El Gerente tiene un bono y el Desarrollador un lenguaje de programación. Cada uno debe implementar un método mostrar\_info().

**Python:**

class Empleado:

def \_\_init\_\_(self, nombre, salario):

self.nombre = nombre

self.salario = salario

def mostrar\_info(self):

return f"{self.nombre} gana {self.salario}"

class Gerente(Empleado):

def \_\_init\_\_(self, nombre, salario, bono):

super().\_\_init\_\_(nombre, salario)

self.bono = bono

def mostrar\_info(self):

return f"{self.nombre} (Gerente) gana {self.salario + self.bono} con bono"

class Desarrollador(Empleado):

def \_\_init\_\_(self, nombre, salario, lenguaje):

super().\_\_init\_\_(nombre, salario)

self.lenguaje = lenguaje

def mostrar\_info(self):

return f"{self.nombre} (Dev) gana {self.salario} y programa en {self.lenguaje}"

# Uso

g = Gerente("Ana", 3000, 500)

d = Desarrollador("Luis", 2500, "Python")

print(g.mostrar\_info()) # Ana (Gerente) gana 3500 con bono

print(d.mostrar\_info()) # Luis (Dev) gana 2500 y programa en Python

**C#:**

public abstract class Empleado

{

public string Nombre { get; set; }

public double Salario { get; set; }

public Empleado(string nombre, double salario)

{

Nombre = nombre;

Salario = salario;

}

public abstract string MostrarInfo();

}

public class Gerente : Empleado

{

public double Bono { get; set; }

public Gerente(string nombre, double salario, double bono)

: base(nombre, salario)

{

Bono = bono;

}

public override string MostrarInfo()

{

return $"{Nombre} (Gerente) gana {Salario + Bono} con bono";

}

}

public class Desarrollador : Empleado

{

public string Lenguaje { get; set; }

public Desarrollador(string nombre, double salario, string lenguaje)

: base(nombre, salario)

{

Lenguaje = lenguaje;

}

public override string MostrarInfo()

{

return $"{Nombre} (Dev) gana {Salario} y programa en {Lenguaje}";

}

}

**Uso:**

Empleado g = new Gerente("Ana", 3000, 500);

Empleado d = new Desarrollador("Luis", 2500, "C#");

Console.WriteLine(g.MostrarInfo());

Console.WriteLine(d.MostrarInfo());

# POLIMORFISMO

**¿Qué es el polimorfismo?**

**Polimorfismo** significa "muchas formas". En POO, es la **capacidad que tienen los objetos de diferentes clases relacionadas por herencia de responder a la misma llamada de método de formas distintas**.

En otras palabras: **mismo método, comportamiento diferente** dependiendo del tipo de objeto.

**¿Por qué es útil?**

* Permite **usar una misma interfaz** para objetos de diferentes tipos.
* Mejora la **flexibilidad y escalabilidad** del código.
* Permite escribir código más **genérico y reutilizable**.

**Ejemplo conceptual**

Imagina una clase base Animal con un método hablar(). Todas las subclases (Perro, Gato, Vaca, etc.) tienen su propia implementación de hablar(). Podemos recorrer una lista de Animal y **llamar a hablar() sin saber qué tipo específico es cada uno**.

**Ejemplo en Python**

class Animal:

def hablar(self):

return "Hace un sonido"

class Perro(Animal):

def hablar(self):

return "Guau"

class Gato(Animal):

def hablar(self):

return "Miau"

class Vaca(Animal):

def hablar(self):

return "Muuu"

# Lista de animales

animales = [Perro(), Gato(), Vaca()]

for animal in animales:

print(animal.hablar()) # Guau, Miau, Muuu

Aquí, aunque no sabemos qué tipo de animal hay en la lista, todos responden correctamente a hablar(). ¡Eso es polimorfismo!

**Ejemplo en C#**

public class Animal

{

public virtual string Hablar()

{

return "Hace un sonido";

}

}

public class Perro : Animal

{

public override string Hablar()

{

return "Guau";

}

}

public class Gato : Animal

{

public override string Hablar()

{

return "Miau";

}

}

public class Vaca : Animal

{

public override string Hablar()

{

return "Muuu";

}

}

**Uso:**

List<Animal> animales = new List<Animal>()

{

new Perro(),

new Gato(),

new Vaca()

};

foreach (var animal in animales)

{

Console.WriteLine(animal.Hablar()); // Guau, Miau, Muuu

}

Igual que en Python, todos responden con su propia versión de Hablar() aunque los tratamos como Animal.

**Tipos de polimorfismo**

| **Tipo** | **Explicación** |
| --- | --- |
| **Polimorfismo de sobrescritura** | Una subclase redefine un método de su clase padre. Ej: override en C#. |
| **Polimorfismo de sobrecarga** | Mismo método con distintos parámetros (C# lo permite, Python no directamente). |

**Comparación rápida**

| **Concepto** | **Python** | **C#** |
| --- | --- | --- |
| Sobrescritura | Solo redefiniendo el método | virtual + override |
| Sobrecarga | No soportado directamente | Sí, métodos con diferentes firmas |
| Interfaces | Por duck typing o ABCs (abc) | Soportadas con interface |

# ABSTRACCION

El **cuarto pilar de la Programación Orientada a Objetos (POO)**:

**¿Qué es la abstracción en POO?**

La **abstracción** consiste en **ocultar los detalles internos de implementación** y mostrar solo lo **esencial y relevante** de un objeto o proceso.

Piensa en un coche: sabes cómo **usarlo** (acelerar, frenar), pero no necesitas saber cómo funciona el motor internamente.

**¿Qué logra la abstracción?**

* **Simplicidad**: Oculta la complejidad.
* **Seguridad**: Evita que el usuario manipule partes sensibles.
* **Modularidad**: Facilita cambios en el código sin afectar otras partes.

**Abstracción vs Encapsulamiento**

| **Concepto** | **Definición rápida** |
| --- | --- |
| **Encapsulamiento** | Ocultar datos internos del objeto. |
| **Abstracción** | Mostrar solo lo necesario, ocultar lo complejo. |

**Ejemplo de abstracción en Python (con clases abstractas)**

Usamos el módulo abc (abstract base classes):

from abc import ABC, abstractmethod

class Figura(ABC):

@abstractmethod

def area(self):

pass

class Cuadrado(Figura):

def \_\_init\_\_(self, lado):

self.lado = lado

def area(self):

return self.lado \* self.lado

class Circulo(Figura):

def \_\_init\_\_(self, radio):

self.radio = radio

def area(self):

return 3.1416 \* self.radio \*\* 2

# Uso

figuras = [Cuadrado(4), Circulo(3)]

for f in figuras:

print(f"Área: {f.area()}")

Aquí la clase Figura es **abstracta**: **no se puede instanciar directamente**. Obliga a que sus subclases implementen el método area().

**Ejemplo en C# (con clases abstractas)**

public abstract class Figura

{

public abstract double CalcularArea();

}

public class Cuadrado : Figura

{

public double Lado { get; set; }

public Cuadrado(double lado)

{

Lado = lado;

}

public override double CalcularArea()

{

return Lado \* Lado;

}

}

public class Circulo : Figura

{

public double Radio { get; set; }

public Circulo(double radio)

{

Radio = radio;

}

public override double CalcularArea()

{

return Math.PI \* Radio \* Radio;

}

}

**Uso:**

List<Figura> figuras = new List<Figura>()

{

new Cuadrado(5),

new Circulo(2.5)

};

foreach (var figura in figuras)

{

Console.WriteLine($"Área: {figura.CalcularArea()}");

}

Igual que en Python, Figura es una **clase abstracta** que **define un contrato** (CalcularArea) que **todas las subclases deben implementar**.

**En resumen:**

| **Concepto** | **Python** | **C#** |
| --- | --- | --- |
| Clase abstracta | Hereda de ABC, usa @abstractmethod | Usa abstract class y abstract method |
| Objetivo | Ocultar detalles, definir estructuras comunes | Igual, con verificación en tiempo de compilación |

# EJEMPLO QUE INTEGRA LOS 4 PILARE DE LA POO

* **Ejercicio Integrador: Gestión de Empleados**

Vamos a modelar un sistema de empleados que tenga:

**Requisitos:**

| **Pilar** | **Aplicación** |
| --- | --- |
| **Encapsulamiento** | Los atributos de cada empleado estarán protegidos. |
| **Herencia** | Habrá una clase base Empleado y clases hijas Gerente, Desarrollador. |
| **Polimorfismo** | Cada empleado calculará su bono de forma distinta con un método calcular\_bono. |
| **Abstracción** | Empleado será una clase abstracta que define lo esencial de cualquier empleado. |

**Versión en Python**

from abc import ABC, abstractmethod

class Empleado(ABC):

def \_\_init\_\_(self, nombre, salario):

self.\_nombre = nombre # Encapsulado

self.\_salario = salario # Encapsulado

def get\_nombre(self):

return self.\_nombre

def get\_salario(self):

return self.\_salario

@abstractmethod

def calcular\_bono(self): # Abstracción

pass

class Gerente(Empleado):

def calcular\_bono(self): # Polimorfismo

return self.\_salario \* 0.20

class Desarrollador(Empleado):

def calcular\_bono(self): # Polimorfismo

return self.\_salario \* 0.10

# Uso

empleados = [

Gerente("Ana", 4000),

Desarrollador("Luis", 3000)

]

for e in empleados:

print(f"{e.get\_nombre()} - Salario: {e.get\_salario()} - Bono: {e.calcular\_bono()}")

**Versión en C#**

using System;

using System.Collections.Generic;

public abstract class Empleado

{

private string nombre; // Encapsulamiento

private double salario; // Encapsulamiento

public Empleado(string nombre, double salario)

{

this.nombre = nombre;

this.salario = salario;

}

public string Nombre => nombre;

public double Salario => salario;

public abstract double CalcularBono(); // Abstracción

}

public class Gerente : Empleado

{

public Gerente(string nombre, double salario) : base(nombre, salario) { }

public override double CalcularBono() // Polimorfismo

{

return Salario \* 0.20;

}

}

public class Desarrollador : Empleado

{

public Desarrollador(string nombre, double salario) : base(nombre, salario) { }

public override double CalcularBono() // Polimorfismo

{

return Salario \* 0.10;

}

}

**Uso:**

class Program

{

static void Main()

{

List<Empleado> empleados = new List<Empleado>()

{

new Gerente("Ana", 4000),

new Desarrollador("Luis", 3000)

};

foreach (var emp in empleados)

{

Console.WriteLine($"{emp.Nombre} - Salario: {emp.Salario} - Bono: {emp.CalcularBono()}");

}

}

}

**¿Qué aprendiste con este ejemplo?**

* **Encapsulaste** datos sensibles (nombre, salario).
* Usaste **herencia** para reutilizar la estructura de Empleado.
* Implementaste **polimorfismo** con métodos sobrescritos (calcular\_bono).
* Aplicaste **abstracción** con una clase Empleado que define el contrato.