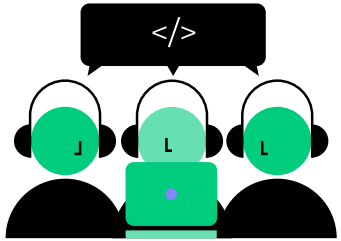
DIGITALHOUSE

# Programación Orientada a Objetos

# 01 ¿Qué es la POO?

# Programación orientada a objetos

La Programación Orientada a Objetos (POO) es un *paradigma de programación* que se basa en el concepto de "objetos", los cuales representan entidades del mundo real que pueden tener tanto datos (atributos) como funcionalidades (métodos). En lugar de centrarse en el flujo de control del programa y las funciones, la POO organiza el código alrededor de estos objetos, lo que permite una mejor organización, reutilización y mantenimiento del código.



01

**Clase**: Una clase es una plantilla o un "molde" para crear objetos. Define los atributos (datos) y métodos (funcionalidades) que todos los objetos de esa clase compartirán. Por ejemplo, si estuviéramos modelando un sistema de gestión de empleados, la clase "Empleado" podría tener atributos como nombre, salario y departamento, y métodos como calcular salario o cambiar departamento.

02

**Objeto**: Un objeto es una instancia específica de una clase. Se crea utilizando la clase como plantilla y representa una entidad específica. Siguiendo el ejemplo anterior, un objeto de la clase "Empleado" podría ser Juan, con un salario de 3000 y perteneciendo al departamento de ventas.

03

**Atributos**: Los atributos son variables que pertenecen a un objeto específico. Representan características o datos asociados con el objeto. En el ejemplo del empleado, los atributos podrían ser el nombre, el salario y el departamento.

04

**Métodos**: Los métodos son funciones asociadas con un objeto específico que pueden realizar operaciones sobre los datos del objeto o interactuar con otros objetos. En el ejemplo del empleado, los métodos podrían ser calcular salario, cambiar departamento, o imprimir información del empleado.

# Ejemplo:

En este ejemplo define una **clase Empleado** con tres ***atributos*** (nombre, salario y departamento) y un ***método*** *mostrar_informacion*() que imprime la información del empleado.

```python
class Empleado:
    def __init__(self, nombre, salario, departamento):
        self.nombre = nombre
        self.salario = salario
        self.departamento = departamento

    def mostrar_informacion(self):
        print(f"Nombre: {self.nombre}, Salario: {self.salario}, Departamento: {self.departamento}")

# Crear objetos (instancias) de la clase Empleado
empleado1 = Empleado("Juan", 3000, "Ventas")
empleado2 = Empleado("María", 3500, "Marketing")

# Acceder a los atributos y llamar a métodos
empleado1.mostrar_informacion()
empleado2.mostrar_informacion()
```

# 02 Herencia

# Herencia:

La **herencia** es un concepto fundamental en la **POO** que permite la creación de **nuevas clases** basadas en **clases existentes**.

En Python, una clase puede ***heredar*** **atributos** y **métodos** de otra clase, conocida como clase base o **superclase**. La clase que hereda se conoce como clase derivada o **subclase**.

```python
class Animal:
    def __init__(self, nombre):
        self.nombre = nombre

    def sonido(self):
        pass

class Perro(Animal):  # Perro hereda de Animal
    def sonido(self):
        return "Guau"

class Gato(Animal):  # Gato hereda de Animal
    def sonido(self):
        return "Miau"
```

# 03 Polimorfismo

# Polimorfismo:

El **polimorfismo** permite tratar objetos de diferentes clases de manera uniforme.

En Python, el **polimorfismo** se logra al definir **métodos** en las **clases derivadas** que tienen el *mismo nombre* que los *métodos en la clase base*, pero con **implementaciones** específicas para cada *clase derivada*.

```python
class Animal:
    def sonido(self):
        pass

class Perro(Animal):
    def sonido(self):
        return "Guau"

def hacer_sonar(animal):
    return animal.sonido()

perro = Perro()
print(hacer_sonar(perro))  # Salida: Guau
```

# 04 Encapsulamiento

# Encapsulamiento

El encapsulamiento es uno de los principios fundamentales de la programación orientada a objetos que consiste en ocultar los detalles internos de un objeto y solo exponer lo que es necesario para su uso externo. Esto se logra definiendo atributos y métodos como públicos, privados o protegidos.

- El encapsulamiento permite mantener la integridad de los datos al restringir el acceso directo a ellos y proporcionar métodos para interactuar con ellos de manera controlada.
- Ayuda a ocultar la complejidad interna de un objeto, lo que facilita su uso y mantenimiento.
- Promueve la modularidad y el reuso del código al definir interfaces claras entre los distintos componentes del programa.

# Niveles de encapsulamiento en Python:

**Público**

Los atributos y métodos que no comienzan con un guion bajo (_) se consideran públicos y pueden ser accedidos desde cualquier parte del programa.

**Protegido**

Los atributos y métodos que comienzan con un solo guión bajo (_) se consideran protegidos y deberían ser tratados como no destinados a la API pública, aunque aún se pueden acceder desde fuera de la clase.

**Privado**

Los atributos y métodos que comienzan con dos guiones bajos (__) se consideran privados y están ocultos del acceso directo desde fuera de la clase. Sin embargo, Python ofrece una forma de acceder a ellos utilizando el mecanismo de name mangling (_NombreClase__atributo).

En este ejemplo, la clase **Persona** tiene dos **atributos privados** __nombre y __edad.

Estos **atributos** están *ocultos* del acceso directo desde fuera de la **clase**.

En su lugar, se proporcionan **métodos públicos** obtener_nombre(), obtener_edad() y establecer_edad() para obtener y modificar estos **atributos** de manera controlada.

```python
class Persona:
    def __init__(self, nombre, edad):
        self.__nombre = nombre  # atributo privado
        self.__edad = edad      # atributo privado

    def obtener_nombre(self):
        return self.__nombre

    def obtener_edad(self):
        return self.__edad

    def establecer_edad(self, nueva_edad):
        if nueva_edad > 0:
            self.__edad = nueva_edad
        else:
            print("La edad debe ser un número positivo.")

persona1 = Persona("Juan", 30) # Crear un objeto persona

# Acceder a los atributos a través de métodos públicos
print("Nombre:", persona1.obtener_nombre())
print("Edad:", persona1.obtener_edad())

# Modificar la edad a través de un método público
persona1.establecer_edad(35)
print("Nueva Edad:", persona1.obtener_edad())
```

05

# Métodos Especiales (Magic Methods)

# Métodos Especiales (Magic Methods):

Los métodos especiales, también conocidos como magic methods o dunder methods (por "double underscore"), son métodos predefinidos en Python que tienen nombres especiales rodeados por doble guion bajo. Estos métodos son invocados automáticamente en ciertas operaciones de los objetos.

- ➔ Los métodos especiales permiten definir el comportamiento de los objetos en operaciones comunes, como la creación, comparación, representación y manipulación.
- ➔ Permiten que los objetos se comporten de manera similar a los tipos integrados de Python, como números, secuencias y contenedores, lo que facilita su integración en el lenguaje.
- ➔ Al implementar estos métodos en una clase, se puede personalizar el comportamiento de los objetos y hacer que interactúen de manera intuitiva con otros objetos y operaciones.

**__init__**: Se utiliza para inicializar los atributos nombre y edad de cada objeto Persona.
**__str__**: Devuelve una representación de cadena legible para humanos del objeto Persona, mostrando su nombre y edad.
**__eq__**: Se utiliza para comparar si dos objetos Persona son iguales, en función de si tienen el mismo nombre y edad.

```python
class Persona:
    def __init__(self, nombre, edad):
        self.nombre = nombre
        self.edad = edad

    def __str__(self):
        return f"{self.nombre} tiene {self.edad} años."

    def __eq__(self, otra_persona):
        return self.nombre == otra_persona.nombre and self.edad == otra_persona.edad

# Crear objetos persona
persona1 = Persona("Juan", 30)
persona2 = Persona("María", 25)
persona3 = Persona("Juan", 30)

# Mostrar la representación de cadena de los objetos
print(persona1)  # Salida: Juan tiene 30 años.
print(persona2)  # Salida: María tiene 25 años.

# Comparar objetos
print(persona1 == persona2)  # Salida: False
print(persona1 == persona3)  # Salida: True
```