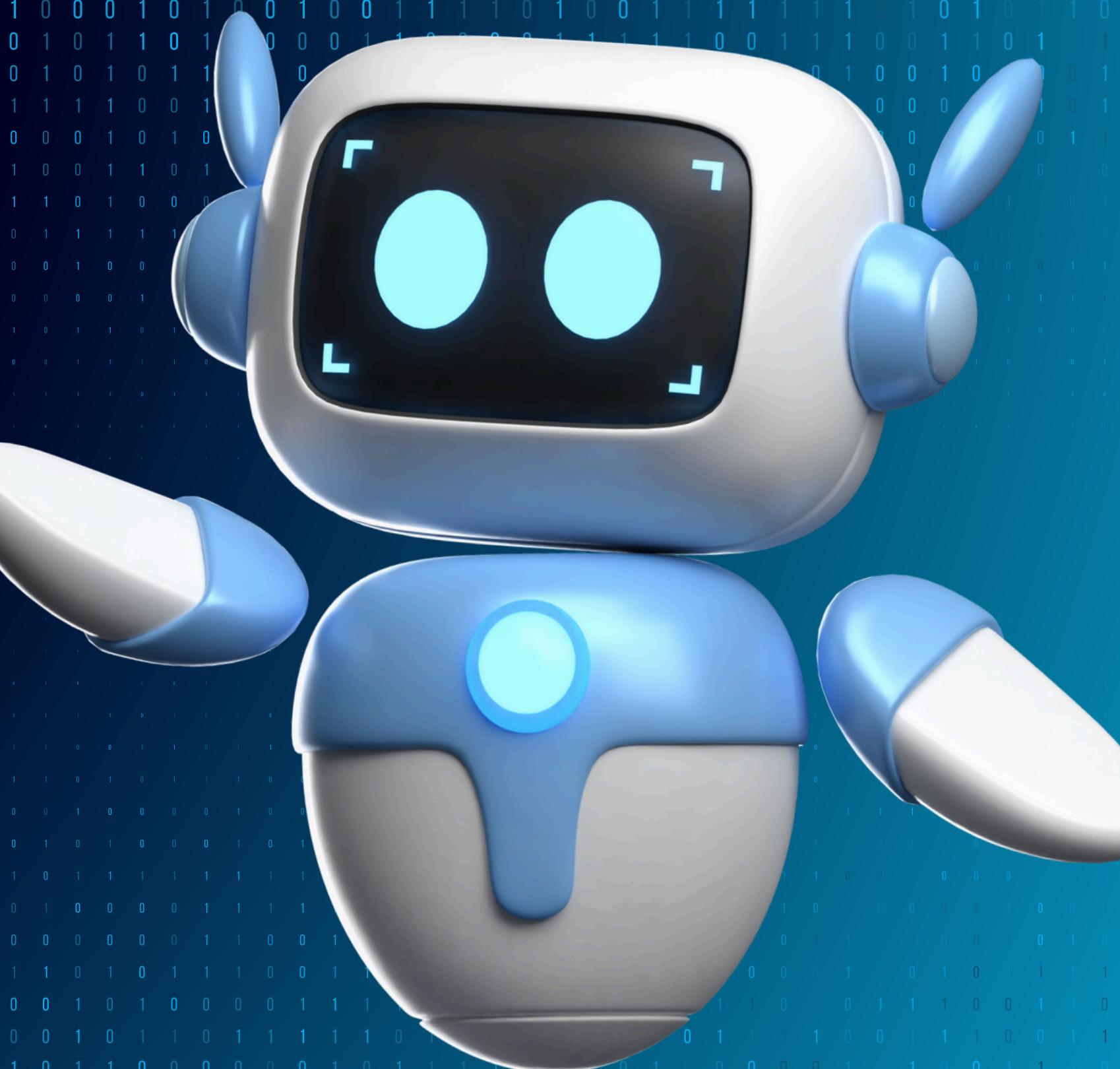
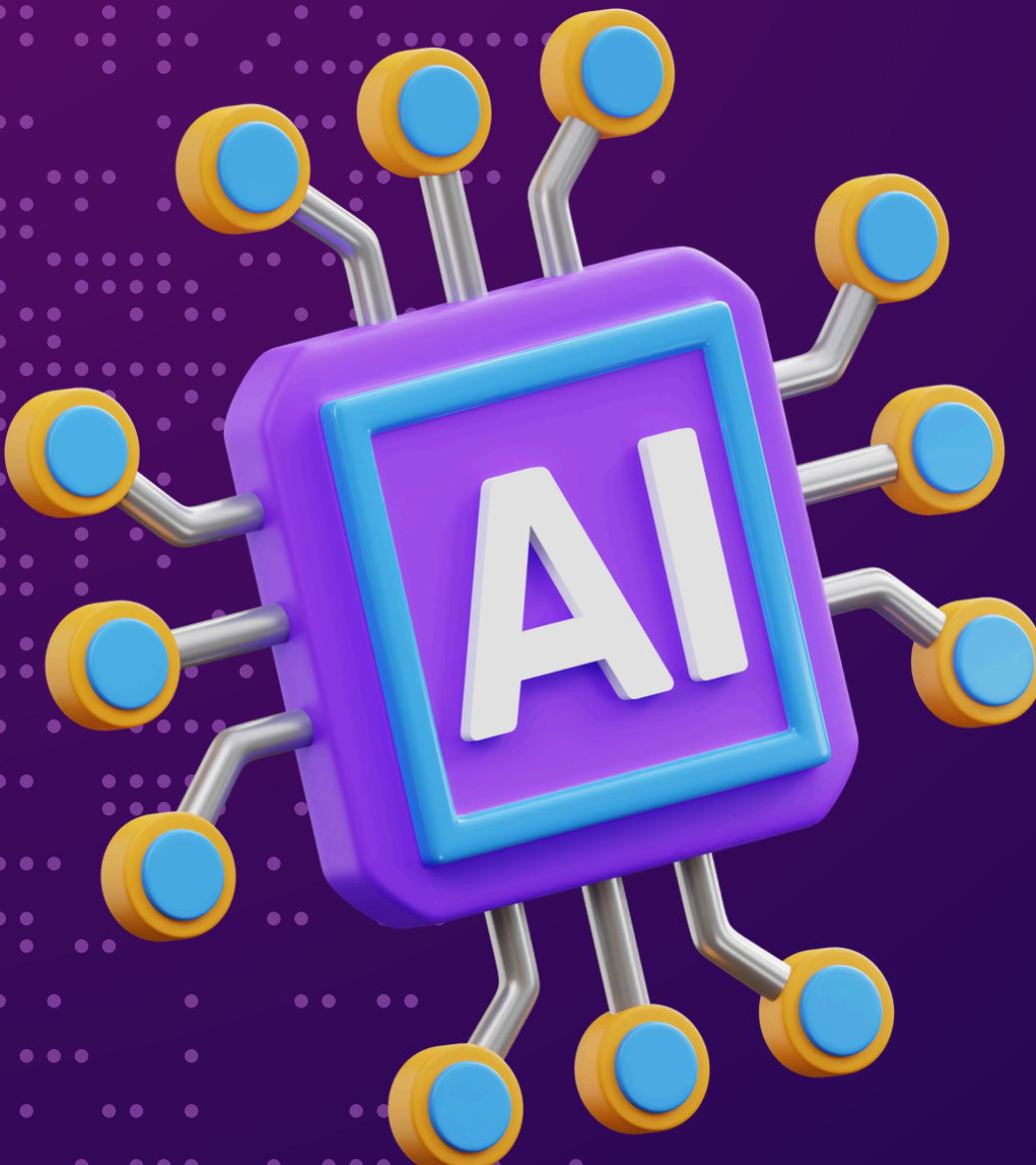


ENCAPSULAMIENTO

Camila Vera, Majo Salinas,
Paula Barrera y Linda
Garnica





CONTENTS

- ① Programación orientada a objetos
- ② Características
- ③ Ejemplos
- ④ Encapsulamiento
- ⑤ Características
- ⑥ Ejemplos



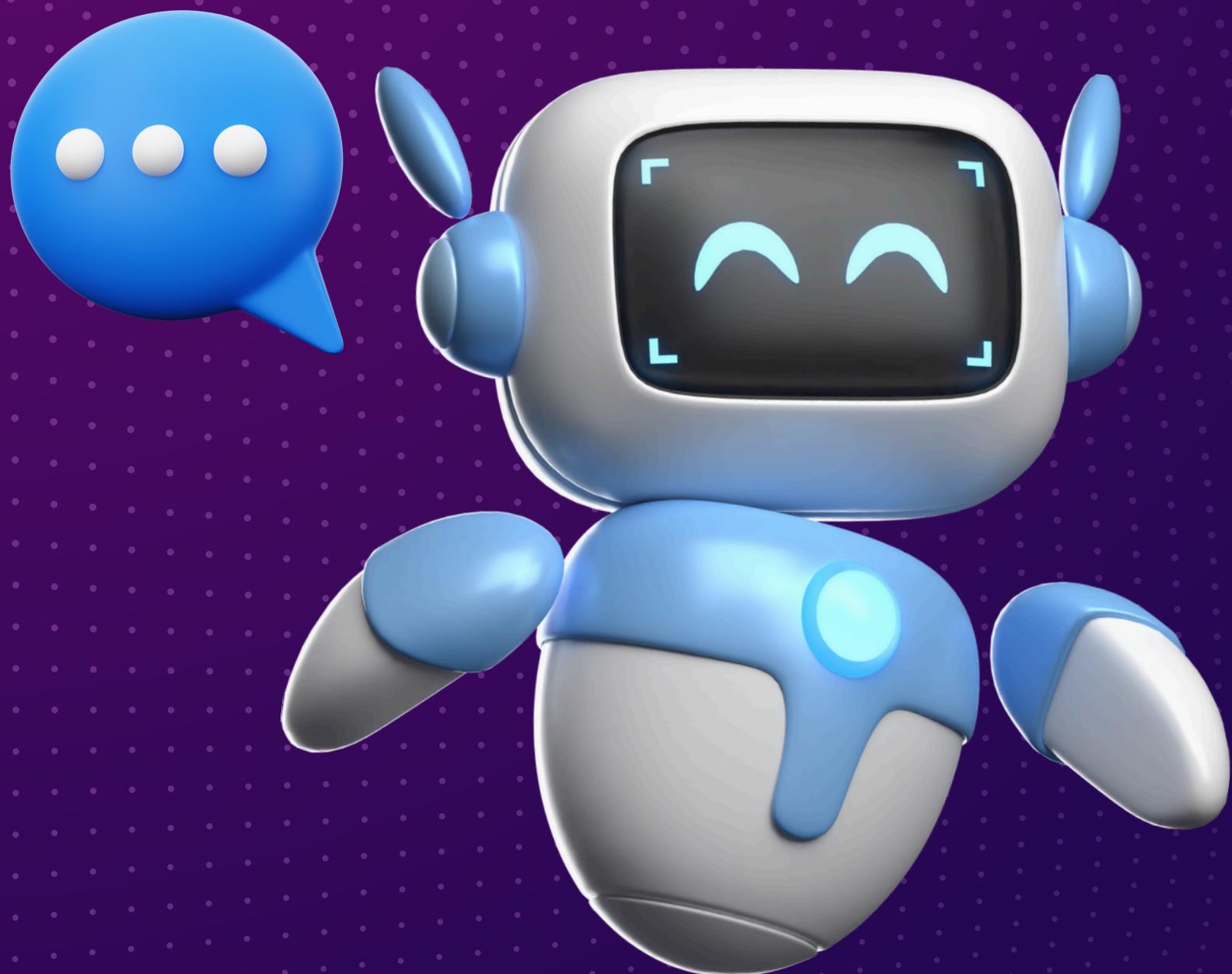
PROGRAMACIÓN ORIENTADA

Es un paradigma de programación que se basa en la conceptualización del mundo real en objetos. Estos objetos agrupan tanto datos (atributos) como comportamiento (métodos o funciones) dentro de una misma entidad. La idea central de la POO es modelar problemas complejos mediante la interacción de estos objetos, que pueden representar cosas concretas o abstractas.



2

CARACTERÍSTICAS



CLASES Y OBJETOS

ENCAPSULAMIENTO

HERENCIA

POLIMORFISMO

ABSTRACCIÓN

CLASES Y OBJETOS

CLASES

Es una plantilla o molde que define las características (atributos) y los comportamientos (métodos) comunes a un grupo de objetos. Es una abstracción que describe la estructura de datos y las operaciones que se pueden realizar sobre esos datos.

OBJETOS

Es una instancia de una clase. Cada objeto tiene sus propios valores para los atributos definidos en la clase y puede usar los métodos definidos en ella.

CLASES Y OBJETOS

EJEMPLO

```
# Definir la clase 'Coche'  
class Coche:  
    def __init__(self, marca, modelo):  
        self.marca = marca # Atributo público  
        self.modelo = modelo # Atributo público  
  
# Crear un objeto de la clase 'Coche'  
mi_coche = Coche("Toyota", "Corolla")  
  
# Acceso a los atributos del objeto  
print(mi_coche.marca) # Salida: Toyota  
print(mi_coche.modelo) # Salida: Corolla
```

ENCAPSULAMIENTO

Los objetos pueden ocultar parte de su información interna, permitiendo acceder y modificar solo a través de métodos específicos. Esto asegura que el acceso a los datos se controle y sea seguro, protegiendo la integridad de los datos.

CLASES Y OBJETOS

EJEMPLO

```
# Definir la clase 'Coche'  
class Coche:  
    def __init__(self, marca, modelo):  
        self.marca = marca # Atributo público  
        self.modelo = modelo # Atributo público  
  
# Crear un objeto de la clase 'Coche'  
mi_coche = Coche("Toyota", "Corolla")  
  
# Acceso a los atributos del objeto  
print(mi_coche.marca) # Salida: Toyota  
print(mi_coche.modelo) # Salida: Corolla
```

HERENCIA

Permite crear nuevas clases basadas en clases existentes. Las clases derivadas heredan los atributos y métodos de las clases base, lo que promueve la reutilización de código y la organización jerárquica.

HERENCIA

EJEMPLO

```
# Clase base 'Animal'  
class Animal:  
    def __init__(self, nombre):  
        self.nombre = nombre # Atributo público  
  
# Clase derivada 'Perro' hereda de 'Animal'  
class Perro(Animal):  
    def __init__(self, nombre, raza):  
        super().__init__(nombre) # Heredar el atributo 'nombre'  
        self.raza = raza # Atributo público  
  
# Crear un objeto de la clase 'Perro'  
mi_perro = Perro("Firulais", "Labrador")  
  
# Acceso a los atributos del objeto  
print(mi_perro.nombre) # Salida: Firulais  
print(mi_perro.raza) # Salida: Labrador
```

POLIMORFISMO

Hace referencia a la capacidad de los objetos de ser tratados como instancias de su clase base o de sus clases derivadas. Permite que un mismo método pueda comportarse de manera diferente dependiendo del objeto que lo invoca.

POLIMORFISMO

EJEMPLO

```
# Clase base 'Animal'  
class Animal:  
    def __init__(self, tipo_sonido):  
        self.tipo_sonido = tipo_sonido # Atributo público  
  
        # Clase derivada 'Perro'  
class Perro(Animal):  
    def __init__(self):  
        super().__init__("Ladrido") # Llamar al constructor de la clase base  
  
        # Clase derivada 'Gato'  
class Gato(Animal):  
    def __init__(self):  
        super().__init__("Maullido") # Llamar al constructor de la clase base  
  
        # Crear objetos de ambas clases  
perro = Perro()  
gato = Gato()  
  
        # Acceder al atributo común  
print(perro.tipo_sonido) # Salida: Ladrido  
print(gato.tipo_sonido) # Salida: Maullido
```

ABSTRACCIÓN

Simplifica la complejidad del sistema al ocultar los detalles de implementación y mostrar solo las funcionalidades esenciales.

Esto permite a los desarrolladores centrarse en el "qué" hace un objeto sin preocuparse por el "cómo" lo hace internamente.

CLASES Y OBJETOS

EJEMPLO

```
# Definir la clase 'Coche'  
class Coche:  
    def __init__(self, marca, modelo):  
        self.marca = marca # Atributo público  
        self.modelo = modelo # Atributo público  
  
# Crear un objeto de la clase 'Coche'  
mi_coche = Coche("Toyota", "Corolla")  
  
# Acceso a los atributos del objeto  
print(mi_coche.marca) # Salida: Toyota  
print(mi_coche.modelo) # Salida: Corolla
```