

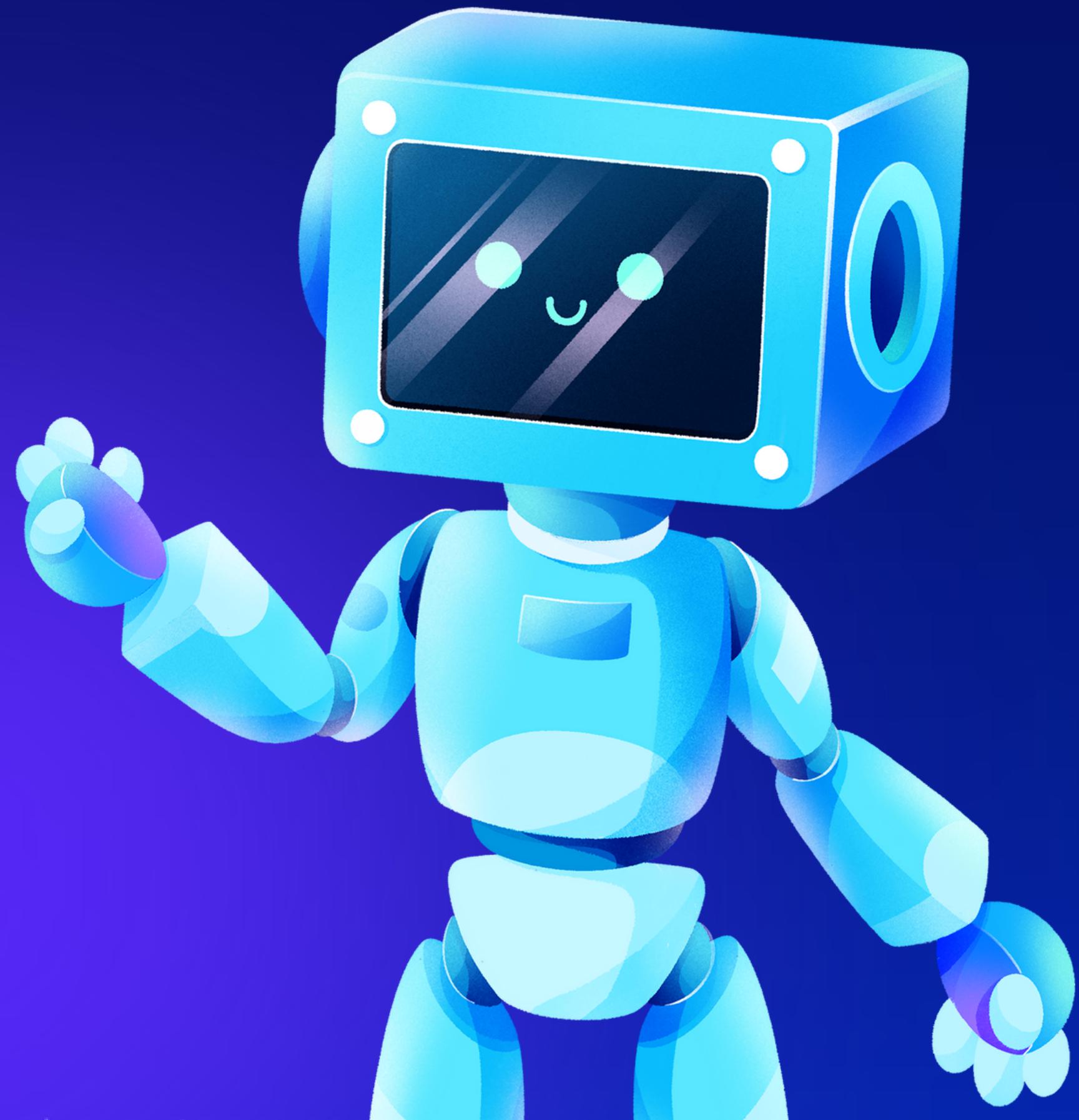
ALGORITMOS 2

JAVA

A

PYTHON

Licenciatura en Ciencias de Datos



INDICE

- Paradigma OO 01
- Clases y Enum 02
- Atributos y Métodos 03
- POO - Herencia 04
- POO - Polimorfismo 05
- Clase Abstracta e Interfaz -
Decorator 06
- Main y Constructor 07
- Property - Funciones Internas 08
- Type Hints 09





PARADIGMA ORIENTADO A OBJETOS

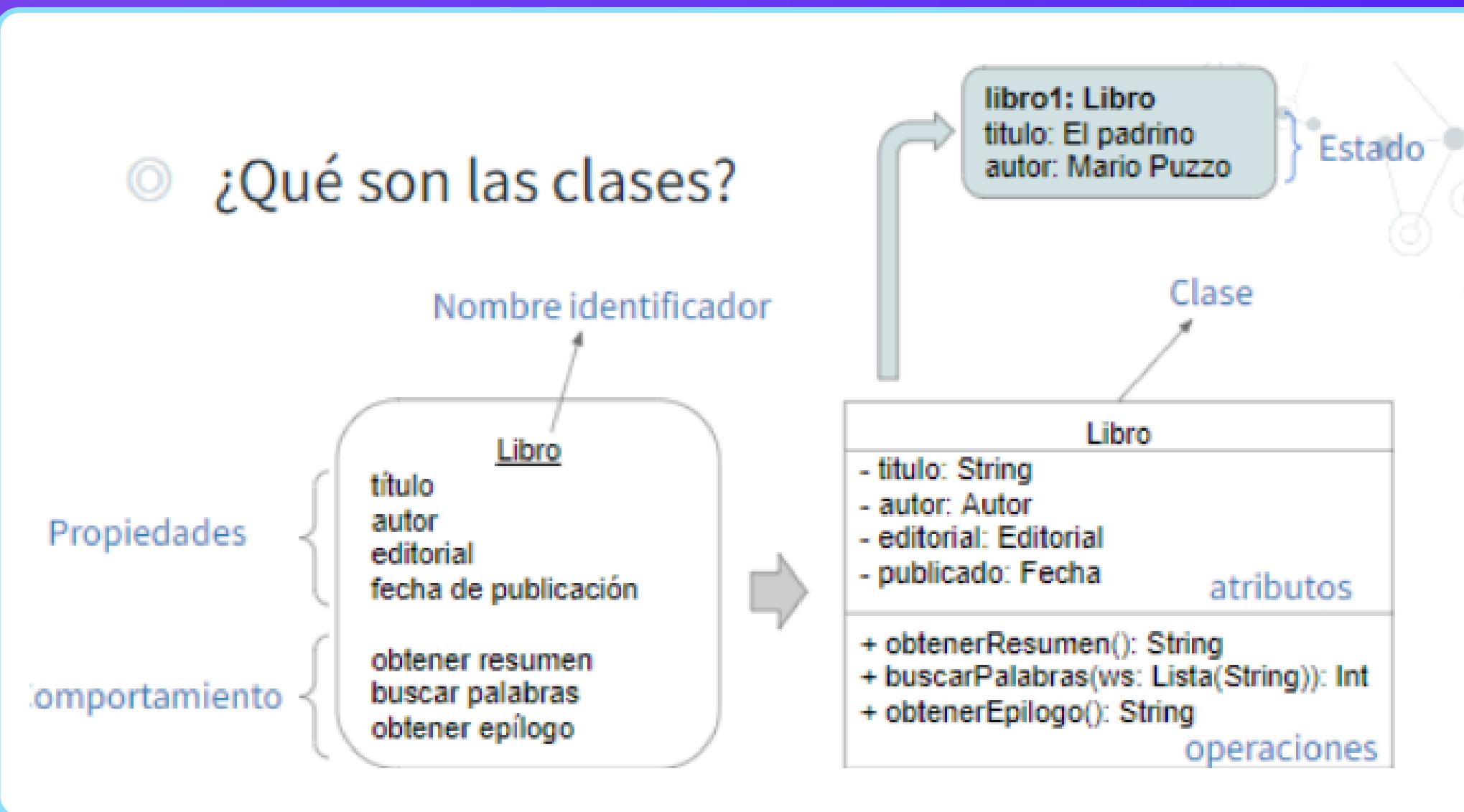
¿Qué son los objetos?

- Una abstracción de dato que captura: Características + Comportamiento
- Se instancia de una Clase
- Tienen un estado asociado
- Interactúan a través de mensajes
- Tienen una representación interna a través de los atributos de datos
- Tienen una interfaz para interactuar con otros objetos



CLASES

○ ¿Qué son las clases?



Clase:

Una clase es una plantilla que define los atributos (propiedades) y métodos (comportamientos) que un objeto puede tener.

Por ejemplo, una clase "Auto" puede tener atributos como "marca" y "modelo", y métodos como "arrancar" y "detener"



CLASES

Python

En Python, definir una clase es sencillo y flexible. Se utiliza la palabra clave `class` seguida del nombre de la clase. Dentro de la clase, puedes definir métodos (funciones) y atributos (variables). Python también soporta herencia, donde una clase puede heredar atributos y métodos de otra clase.

Veamos un ejemplo:

```
class Alumno:  
    def __init__(self):  
        self.nombre = "Pablo"  
  
    def saludar(self):  
        """Imprime un saludo en pantalla."""  
        print(f"¡Hola, {self.nombre}!")
```

Java

En Java, definir una clase es similar pero con algunas diferencias en sintaxis y estructura. Se utiliza la palabra clave `class` seguida del nombre de la clase, y el cuerpo de la clase está encerrado en llaves `{}`. Java requiere que todas las variables de instancia se declaren al principio de la clase y solo soporta herencia simple.

Veamos un ejemplo:

```
public class Alumno {  
    private String nombre;  
  
    public Alumno(String nombre) {  
        this.nombre = nombre;  
    }  
  
    public void saludar() {  
        System.out.println("¡Hola, " + nombre + "!");  
    }  
}
```



CLASES

Permiten definir un conjunto de atributos y métodos que describen las características y el comportamiento de un objeto en particular.

```
class NombreClase:  
    # Definición de atributos y métodos de la clase
```

Instanciación vs Inicialización

La instanciación es el proceso de crear un objeto a partir de una clase. Cuando se **instancia** una clase, se reserva espacio en la memoria para el objeto con el método especial `__new__()` y luego se **inicializa** el objeto utilizando el método especial `__init__()`, también conocido como el constructor de la clase.

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

```
juana = Persona("Juana", 23)  
print(juana)                      # <__main__.Persona at 0x1d2bd5b1750>  
print(hex(id(juana)))              # 1d2bd5b1750  
print(juana.nombre)                # Juana  
print(juana.edad)                  # 23
```



ENUM

Python

Los enum se implementan utilizando el módulo enum. Para definir un enum, se crea una clase que hereda de enum.Enum y se definen los miembros del enum como atributos de la clase.

```
from enum import Enum

class Animal(Enum):
    PERRO = 1
    GATO = 2
    LEON = 3

# Accediendo a los miembros del enumerado
print(Animal.PERRO)
print(Animal.PERRO.name)
print(Animal.PERRO.value)

# Iterando sobre todos los miembros del enumerado
for miembro in Animal:
    print(miembro)
```

Java

Los enum se definen utilizando la palabra clave enum. Los miembros del enum se definen como identificadores seguidos de dos puntos y un valor.

```
public enum Operations {
    ADD, SUBTRACT, MULTIPLY, DIVIDE, NONE
}

// Accediendo a los miembros del enumerado
System.out.println(Operations.ADD);

// Iterando sobre todos los miembros del enumerado
for (Operations operation : Operations.values()) {
    System.out.println(operation);
}
```



ATRIBUTOS

```
public class Persona {  
    String nombre; // Atributo de tipo String  
    int edad;      // Atributo de tipo int  
    double altura; // Atributo de tipo double  
}
```

Python es dinámico, ya que si invocáramos `type(variable)` luego de cada asignación veríamos los distintos tipos de datos comentados

Los **atributos** son variables que se definen dentro de una clase y representan las propiedades o características de un objeto. Estos atributos definen el estado de un objeto y pueden tener diferentes tipos de datos.

```
variable = 45    # variable apunta a tipo int  
variable = 4.3   # variable apunta a tipo float  
variable = 'a'   # variable apunta a tipo str  
variable = [1]   # variable apunta a tipo list
```



ATRIBUTOS

Python

- Dinámicamente tipado: los tipos de variables se definen en tiempo de ejecución.
- Sin necesidad de declarar explícitamente los atributos: al principio de la definición de la clase. Los atributos se pueden definir y modificar libremente dentro de los métodos de la clase.
- Método `__init__`: para inicializar los atributos de una clase. Este método se llama automáticamente cuando se crea una instancia de la clase.

```
class Persona:  
    def __init__(self, nombre, edad):  
        self.nombre = nombre  
        self.edad = edad  
  
    def presentarse():  
        return f"Hola, mi nombre es {self.nombre} y tengo {self.edad} años."  
  
persona = Persona("Juan", 30)  
print(persona.presentarse())
```

Java

- Estáticamente tipado: se debe especificar el tipo de dato de una variable al declararla.
- Declaración de atributos al principio: de la definición de la clase.
- Constructores: para inicializar los atributos de una clase.

```
public class Persona {  
    private String nombre;  
    private int edad;  
  
    public Persona(String nombre, int edad) {  
        this.nombre = nombre;  
        this.edad = edad;  
    }  
  
    public String presentarse() {  
        return "Hola, mi nombre es " + nombre + " y tengo " + edad + " años.";  
    }  
  
    public static void main(String[] args) {  
        Persona persona = new Persona("Juan", 30);  
        System.out.println(persona.presentarse());  
    }  
}
```



ATRIBUTOS

```
class Persona:  
    contador_personas = 0  
  
    def __init__(self, nombre, edad):  
        self.nombre = nombre  
        self.edad = edad  
        Persona.contador_personas += 1  
  
juana = Persona("Juana", 23)  
hugo = Persona("Hugo", 33)  
  
juana.nombre # 'Juana'  
hugo.edad # 33  
Persona.contador_personas # 2  
juana.contador_personas # 2  
hugo.contador_personas # 2
```

Los atributos de clase son compartidos por todas las instancias de la clase. Estos miembros se definen fuera de cualquier método de la clase y se accede a ellos utilizando el nombre de la clase. Se pueden utilizar para almacenar datos que son comunes a todas las instancias de la clase. Los atributos de instancia son específicos de cada objeto individual y se definen dentro del método `__init__()` utilizando el parámetro `self`. Cada instancia de la clase tiene sus propias copias de los atributos de instancia.



MÉTODOS

```
public class Auto {  
    // Atributos  
  
    // Métodos  
    public void arrancar() {  
        // Código para arrancar el auto  
    }  
  
    public void detener() {  
        // Código para detener el auto  
    }  
  
    public void acelerar(int velocidad) {  
        // Código para acelerar el auto a la velocidad proporcionada  
    }  
}
```

Los **métodos** en Java son funciones que se definen dentro de una clase y representan el comportamiento de un objeto. Estos métodos definen las acciones que un objeto puede realizar. Los métodos pueden aceptar parámetros y devolver resultados.



MÉTODOS

Python

En Python, los métodos son funciones que pertenecen a una clase y pueden modificar el estado de la instancia de la clase. Para definir un método en Python, simplemente usas la palabra clave `def` seguida del nombre del método y los parámetros que recibe.

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

    def presentar(self):
        return f"Este coche es un {self.marca} {self.modelo}."

mi_coche = Coche("Toyota", "Corolla")
print(mi_coche.presentar())
```

Java

En Java, los métodos son la única forma de definir funciones y deben ser declarados dentro de una clase. Los métodos en Java pueden modificar el estado de la instancia de la clase si operan sobre variables de instancia (atributos de la clase).

```
public class Coche {
    private String marca;
    private String modelo;

    public Coche(String marca, String modelo) {
        this.marca = marca;
        this.modelo = modelo;
    }

    public String presentar() {
        return "Este coche es un " + marca + " " + modelo + ".";
    }

    public static void main(String[] args) {
        Coche miCoche = new Coche("Toyota", "Corolla");
        System.out.println(miCoche.presentar());
    }
}
```



METODOS

Las funciones definidas dentro de una clase son métodos de instancia y tienen la particularidad que el primer parámetro siempre es la instancia actual.

```
class Persona:  
    def __init__(self, nombre, apellido):  
        self.nombre = nombre  
        self.apellido = apellido  
  
    def nombre_completo(self):  
        return f'{self.nombre} {self.apellido}'  
  
juana = Persona("Juana", "Lopez")  
juana.nombre_completo() # 'Juana Lopez'
```



METODOS

Cuando deseamos modelar un comportamiento propio de una clase y no de una instancia, podemos definir un método de clase. Estos métodos sólo pueden acceder a atributos de clase, ya que el primer parámetro es necesariamente la clase. Por convención se nombra a ese parámetro como `cls` y para definirlo se utiliza el decorador `@classmethod`.

`personas_creadas()` es un método de clase, recibe como primer argumento a la clase **`Persona`** para poder acceder a sus miembros. Recordando que una instancia puede acceder a miembros de instancia y miembros de clase, podemos acceder a este método mediante

`Persona.personas_creadas()` o **`juana.personas_creadas()`**.

```
class Persona:  
    contador_personas = 0  
  
    def __init__(self, nombre, apellido):  
        self.nombre = nombre  
        self.apellido = apellido  
        Persona.contador_personas += 1  
  
    @classmethod  
    def personas_creadas(cls):  
        return cls.contador_personas  
  
juana = Persona("Juana", "Lopez")  
Persona.personas_creadas()          # 1  
juana.personas_creadas()          # 1
```



GETTERS Y SETTERS

Python

Se pueden implementar utilizando las propiedades de Python, que son una forma de definir métodos que se comportan como atributos. Para definir un getter, se utiliza el decorador `@property`, y para definir un setter, se utiliza el decorador `@atributo.setter`.

Java

Se definen como métodos públicos dentro de la clase. Los getters se utilizan para acceder al valor de un atributo, y los setters para modificar el valor de un atributo.



GETTERS Y SETTERS

Python

```
class Usuario:  
    def __init__(self, username):  
        self.__username = username  
  
    @property  
    def username(self):  
        return self.__username  
  
    @username.setter  
    def username(self, value):  
        if len(value) < 5:  
            raise ValueError("El nombre de usuario debe tener al menos 5 caracteres")  
        self.__username = value  
  
# Uso de la clase  
user = Usuario('eduardo_gpg')  
print(user.username) # Obtener el valor del atributo  
user.username = 'NuevoUsername' # Establecer un nuevo valor  
print(user.username) # Verificar el nuevo valor
```

Java

```
public class Usuario {  
    private String username;  
  
    public Usuario(String username) {  
        this.username = username;  
    }  
  
    // Getter  
    public String getUsername() {  
        return username;  
    }  
  
    // Setter  
    public void setUsername(String username) {  
        if (username.length() < 5) {  
            throw new IllegalArgumentException("El nombre de usuario debe tener al menos 5 caracteres");  
        }  
        this.username = username;  
    }  
  
    // Uso de la clase  
    public static void main(String[] args) {  
        Usuario user = new Usuario("eduardo_gpg");  
        System.out.println(user.getUsername()); // Obtener el valor del atributo  
        user.setUsername("NuevoUsername"); // Establecer un nuevo valor  
        System.out.println(user.getUsername()); // Verificar el nuevo valor  
    }  
}
```



OPERADORES

Python

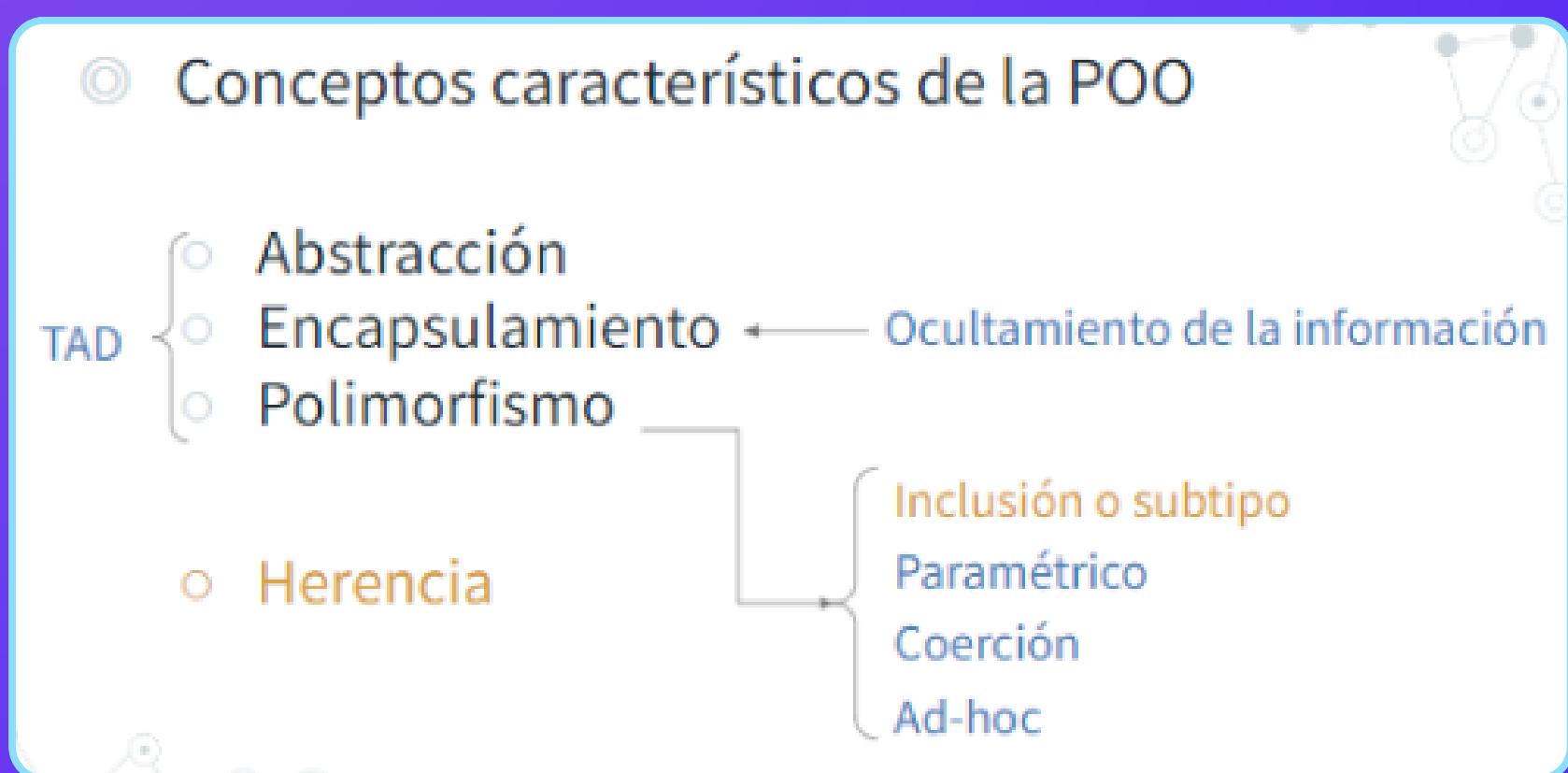
- Operadores aritméticos: +, -, *, /, //, %, **.
- Operadores de comparación: ==, !=, <, <=, >, >=.
- Operadores lógicos: and, or, not.
- Operadores de identidad y membresía: is, is not, in, not in.
- Operadores de asignación: =, +=, -=, *=, /=, %=, **=, //=.
- Operador de indexación y segmentación: [], [:], [::].
- Operador de llamada de función: ()..
- Operador de construcción de tuplas, listas y diccionarios: (), [], {}.
- Operador de desempaquetado: *, **.
- Operador de anulación: None.

Java

- Operadores aritméticos: +, -, *, /, %, ++, --.
- Operadores de comparación: ==, !=, <, <=, >, >=.
- Operadores lógicos: &&, ||, !.
- Operadores de asignación: =, +=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=, >>>= (para desplazamientos).
- Operador de indexación y segmentación: [].
- Operador de llamada de método: ()..
- Operador de construcción de arrays: [].
- Operador de desempaquetado: No existe una funcionalidad directamente equivalente en Java.
- Operador de anulación: null.



PARADIGMA ORIENTADO A OBJETOS



Herencia: La herencia permite que una clase herede atributos y métodos de otra clase. Esto promueve la reutilización de código y facilita la extensibilidad. Por ejemplo, una clase "Camión" puede heredar de la clase "Transporte".

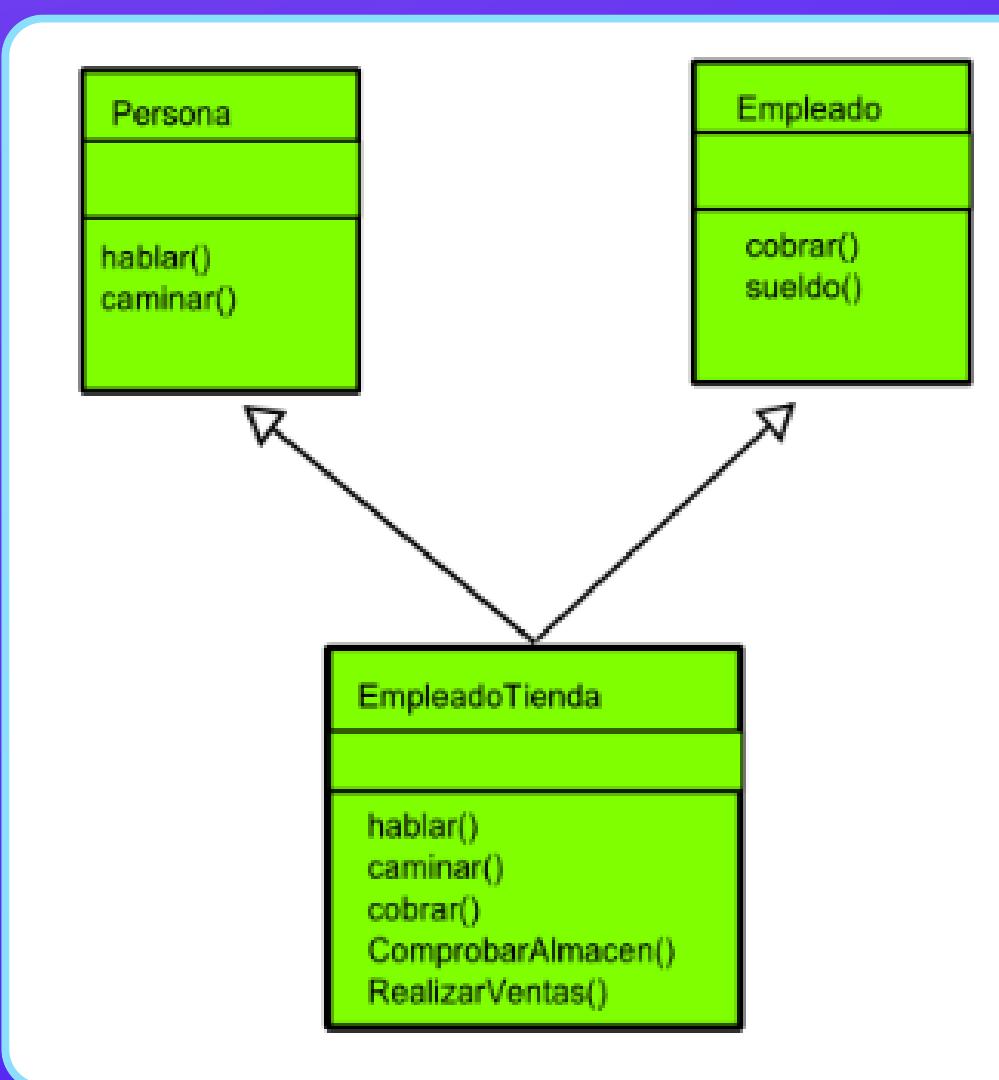
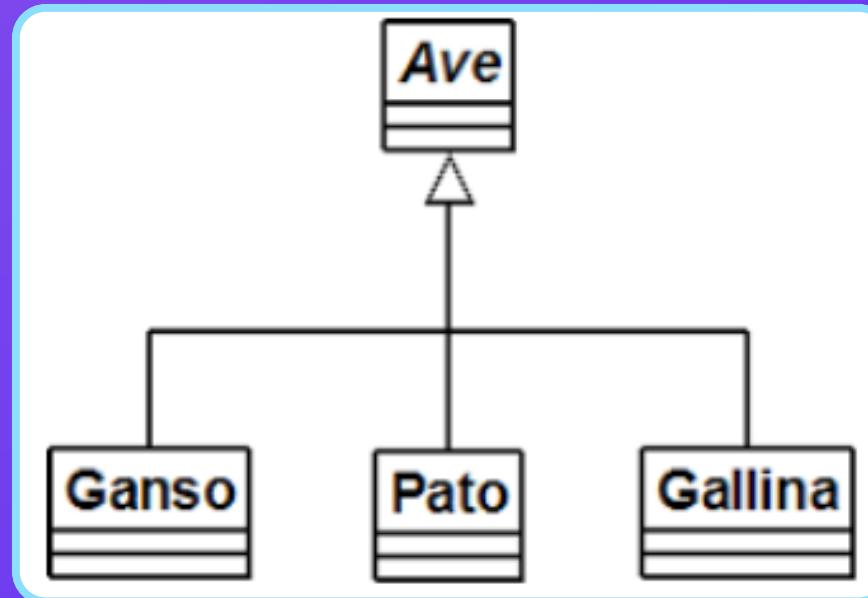
Abstracción: Mecanismo a través del cual podemos concentrarnos en aspectos relevantes e ignorar los detalles. Nos permite definir qué es lo que deseamos modelar sin entrar en detalle de cómo lo realizaremos y así favorecer a la reusabilidad de estos modelos. En la POO podemos construir abstracciones tanto de datos como de comportamiento.

Encapsulamiento: Es el concepto de agrupar datos y métodos relacionados en una sola unidad (un objeto) y ocultar los detalles de implementación del objeto al mundo exterior.

Polimorfismo: Si bien existen distintos tipos de polimorfismo, en la POO el polimorfismo de inclusión o subtipo permite que objetos de diferentes clases respondan a un mismo método de manera diferente. Esto se logra mediante el uso de la herencia.



HERENCIA



La **herencia** es uno de los conceptos clave en la programación orientada a objetos que permite la creación de clases nuevas basadas en clases existentes.

La utilizamos cuando analizamos cómo se relacionan las clases de nuestro sistema existen situaciones donde una clase representa cierta entidad pero también puede representar a otra más abstracta.

Herencia simple: permite que una clase (llamada clase derivada o subclase) herede los atributos y métodos de otra clase (llamada clase base o superclase). Esto facilita la reutilización del código y la creación de jerarquías de clases para favorecer la extensibilidad. Una **subclase** hereda los miembros de una superclase y puede agregar nuevos miembros o modificar los existentes según sea necesario a través de la sobreescritura de métodos de instancia u ocultamiento (hiding) de métodos de clase.

Herencia múltiple: es una característica que permite a una clase heredar atributos y métodos de más de una clase base



HERENCIA EN PYTHON

Se realiza mediante la declaración de una clase derivada o subclase que hereda de una clase extendida o superclase. Entonces, la subclase hereda todos los atributos y métodos de la superclase.

Persona es una superclase que implícitamente hereda de object (la raíz de la jerarquía de clases en Python y proporciona ciertas funcionalidades básicas y métodos comunes a todas las clases), mientras que Estudiante es una subclase de ella.Un objeto de tipo Estudiante será entonces también de tipo Persona y de tipo object.

pass es una palabra clave que se utiliza como marcador de posición cuando no se requiere ningún código

```
class Persona:  
    pass  
  
class Estudiante(Persona):  
    pass  
  
juana = Estudiante()  
isinstance(juana, Estudiante)      # True  
isinstance(juana, Persona)         # True  
isinstance(juana, object)          # True
```



HERENCIA

Python

- Herencia múltiple: una clase puede heredar de múltiples clases (separadas por comas).
- Sintaxis: se indica simplemente al listar las clases base en la definición de la clase, separadas por comas.

```
class Animal:  
    def __init__(self, nombre):  
        self.nombre = nombre  
  
class Perro(Animal):  
    def __init__(self, nombre, raza):  
        super().__init__(nombre)  
        self.raza = raza  
  
class Gato(Animal):  
    def __init__(self, nombre, color):  
        super().__init__(nombre)  
        self.color = color  
  
class PerroMestizo(Perro, Gato):  
    def __init__(self, nombre, raza, color):  
        super().__init__(nombre, raza)  
        Gato.__init__(self, nombre, color)
```

Java

```
public class Animal {  
    protected String nombre;  
  
    public Animal(String nombre) {  
        this.nombre = nombre;  
    }  
  
    public class Perro extends Animal {  
        private String raza;  
  
        public Perro(String nombre, String raza) {  
            super(nombre);  
            this.raza = raza;  
        }  
  
        public class Gato extends Animal {  
            private String color;  
  
            public Gato(String nombre, String color) {  
                super(nombre);  
                this.color = color;  
            }  
        }  
    }  
}
```

- Herencia simple: una clase puede heredar de una sola clase base.
- Sintaxis: se indica utilizando la palabra clave `extends` en la definición de la clase.

Java no soporta herencia múltiple directamente, por lo que no se puede crear una clase PerroMestizo que herede de Perro y Gato simultáneamente.



HERENCIA

Sobreescritura

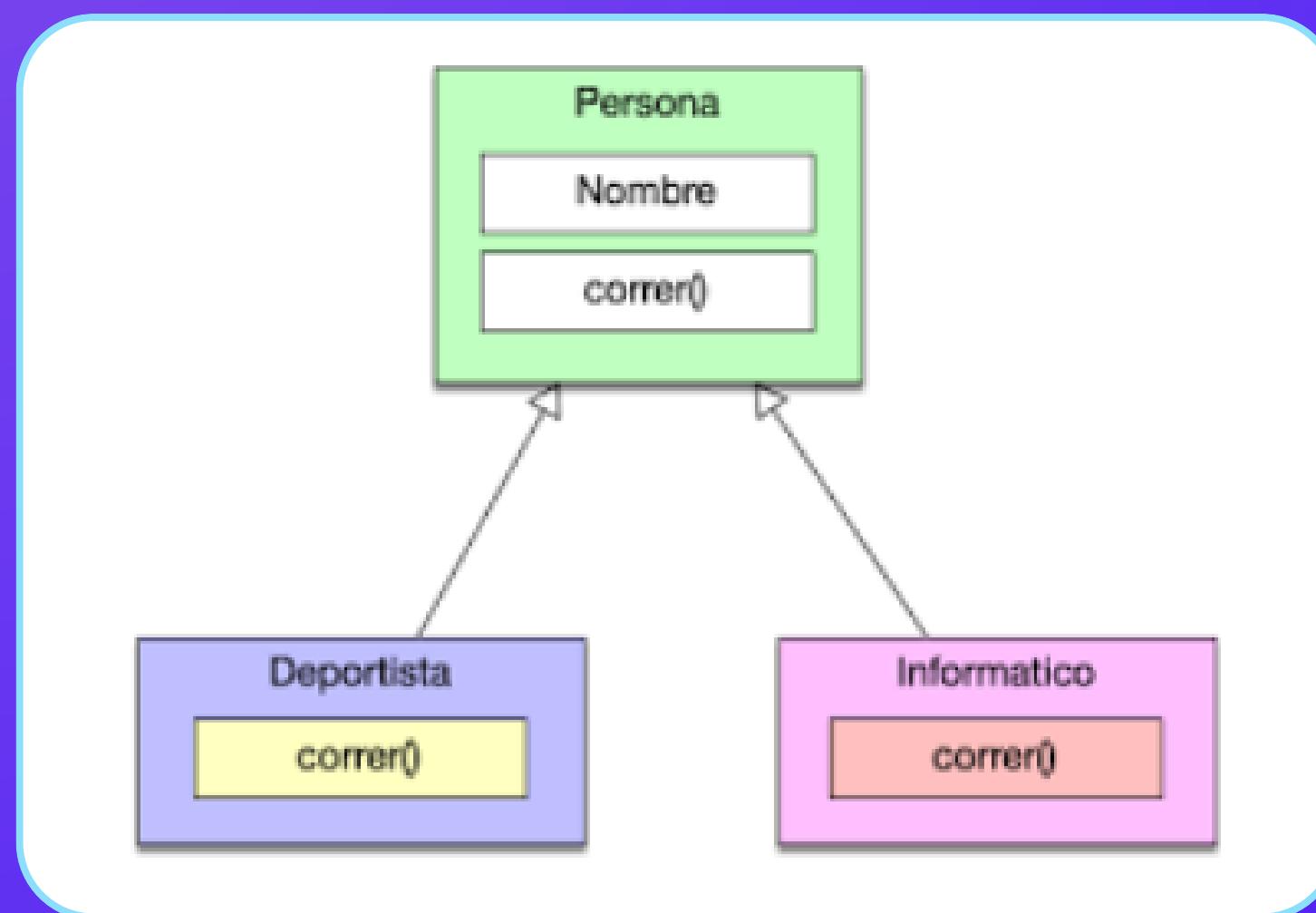
En caso que necesitáramos adaptar el comportamiento heredado a una subclase, podemos hacer uso de la sobreescritura de métodos.

```
class Persona:  
    # Implementación de Persona...  
  
    def __str__(self):  
        return f'{self.nombre} {self.apellido}'  
  
class Estudiante(Persona):  
    # Implementación de Estudiante...  
  
    def __str__(self):  
        return f'Estudiante {super().__str__()}'  
  
juana = Estudiante("Juana", "Lopez", 1234)  
print(juana)      # Estudiante Juana Lopez
```

En este ejemplo, Estudiante hereda el método `__str__()` de Persona, pero lo sobrescribe con su propia versión que casualmente invoca explícitamente el método heredado con `super().__str__()`. `__str__()` es un método especial (también conocido como "método mágico" o "dunder method") que se utiliza para definir la representación de cadena de un objeto. Devuelve una representación de cadena no informativa del objeto, que generalmente consiste en el nombre de la clase y la dirección de memoria del objeto.



POLIMORFISMO



El **polimorfismo** es un componente esencial de la herencia y nos permite escribir código más flexible y reutilizable. Se apoya en el concepto de sobrescritura. Consiste en la capacidad de objetos de diferentes clases de responder al mismo mensaje o método de manera diferente. En otras palabras, el polimorfismo permite que diferentes clases compartan el mismo nombre de método pero implementen ese método de manera específica para cada clase.