# UD 3. Programación orientada a objetos

## 1. CLASES, CONSTRUCTORES Y MODIFICADORES DE ACCESO

### 1.1. Clases

Una **clase** es una plantilla o molde que define las características y comportamientos de los objetos. Es decir, especifica qué atributos tendrá un objeto y qué métodos podrá ejecutar.

Describe:

- **Atributos**: características del objeto (datos).
- **Métodos**: acciones que puede realizar el objeto (comportamientos).

Ejemplo conceptual:

```
Class Persona {
   String nombre;
   Int edad;

  Void hablar(){
    System.out.println(“Hola, soy ” + nombre);
  }
}
```

### 1.2. Constructores

Un constructor es un método especial que se usa para inicializar un objeto. Características:

- Tiene el mismo nombre que la clase.
- No tiene tipo de retorno.
- Puede sobrecargarse (tener varios constructores con distintos parámetros).
- Permite establecer valores iniciales.

```
// Constructor por defecto
public Persona() {
   this.nombre = "Sin nombre";
   this.edad = 0;
}

// Constructor con parámetros
public Persona(String nombre, int edad) {
   this.nombre = nombre;
   this.edad = edad;
```

```
  }
}
```

### 1.3. Modificadores de acceso

Controlan la visibilidad de atributos y métodos:

- **public**: accesible desde cualquier parte.
- **private**: accesible solo dentro de la misma clase.
- **protected**: accesible en la clase, el mismo paquete o clases hijas.
- (sin modificador): acceso por defecto dentro del mismo paquete.

Su objetivo es **encapsular** la información para evitar usos indebidos del objeto.

## 2. HERENCIA Y POLIMORFISMO

### 2.1. Herencia

La **herencia** permite que una clase (subclase) herede atributos y métodos de otra clase (superclase). Facilita la reutilización del código y la creación de jerarquías.

La clase hija:

- Obtiene atributos y métodos de la clase padre.
- Puede agregar nuevos miembros.
- Puede modificar comportamientos heredados.

Tipos de herencia:

- **Simple**: una clase hereda de una sola superclase (Java, C#, Python).
- **Múltiple**: una clase hereda de varias superclases (posible en C++, no en Java).
- **Jerárquica**: varias clases heredan de una sola.
- **Multinivel**: una clase deriva de otra subclase.

```
class Animal {
  void comer() { System.out.println("El animal come"); }
}

// Perro hereda el método comer() de Animal.
class Perro extends Animal {
  void ladrar() { System.out.println("Guau"); }
}
```

### 2.2. Polimorfismo

El **polimorfismo** permite que un mismo método se comporte de forma diferente según el objeto que lo invoque.

➢ **Tipos de polimorfismo:**

1. **Polimorfismo en tiempo de compilación (sobrecarga / overloading)** Se refiere a definir varios métodos con el mismo nombre pero diferentes parámetros.

```
class Matematica {
   int sumar(int a, int b) { return a + b; }
   double sumar(double a, double b) { return a + b; }
}
```

2. **Polimorfismo en tiempo de ejecución (sobrescritura / overriding)** Una subclase redefine el comportamiento de un método heredado.

```
class Animal {
   void sonido() { System.out.println("Sonido genérico"); }
}

class Perro extends Animal {
   @Override
   void sonido() { System.out.println("Guau"); }
}
```

Ejemplo de polimorfismo en acción:

```
Animal a = new Perro();
a.sonido();   // Ejecuta el método de Perro → "Guau"}
```

## 3. INTERFACES E IMPLEMENTACIÓN

Una interfaz funciona como un contrato: define qué métodos debe tener una clase, pero no cómo funcionan. Las clases que la implementan deben escribir la lógica interna.

Características de una interfaz:

- Todos sus métodos son abstractos por defecto (en lenguajes como Java antes de la versión 8).
- Una clase puede implementar múltiples interfaces (solución a la falta de herencia múltiple en Java).
- Permiten desacoplar el código: se programa contra interfaces, no contra implementaciones.

```
interface Reproductor {
   void reproducir();
   void pausar();
   void detener();
}

class MP3 implements Reproductor {
   public void reproducir() { System.out.println("Reproduciendo MP3"); }
   public void pausar() { System.out.println("Pausando MP3"); }
   public void detener() { System.out.println("Deteniendo MP3"); }
}
```

**Ventajas:**

- Fomentan el polimorfismo.
- Permiten intercambiar implementaciones sin modificar el código cliente.
- Ayudan a construir sistemas más flexibles.

## 4. PROPIEDADES Y MÉTODOS ESTÁTICOS

El modificador static indica que un atributo o método pertenece a la **clase**, no a sus objetos.

Esto significa:

- Se comparte entre todas las instancias.
- Puede accederse sin crear objetos.

```
class Usuario {
   static int totalUsuarios = 0;

   public Usuario() {
      totalUsuarios++;
   }
}
```

Cada vez que se crea un usuario, se incrementa un contador compartido.

**Métodos estáticos**

Normalmente se usan para funciones auxiliares o operaciones comunes.

```
class Utilidad {
   static void imprimirLinea() {
      System.out.println("------------------");
   }
}

//Se invoca así:
Utilidad.imprimirLinea();
```

**Reglas importantes:**

- Un método estático **no puede acceder directamente** a atributos no estáticos.
- Se ejecutan sin necesidad de crear objetos.

```
class Config {
   static final double PI = 3.14159;
}
```