Resumen – Java y UML

JAVA

Clases, Atributos y Métodos

Clase: Plantilla de objetos.

```
public class Persona {
// Contenido de la clase
}
```

Atributo: Variable de una clase.

```
int num = 1;
String texto = hola;
```

Método: Comportamiento/función dentro de una clase.

```
public class Persona {
    String nombre;           // Atributo
    int edad;

    void saludar() {         // Método
        System.out.println("Hola, soy " + nombre);
    }
}
```

Calificadores de Acceso

- `public` : Accesible desde cualquier clase.
- `private` : Solo dentro de la clase.
- `protected` : Accesible en subclases y mismo paquete.
- `friendly` (default): Solo dentro del mismo paquete. Se aplica cuando no se especifica ningún modificador.

```
public class Ejemplo {
    public int publico;      // Accesible desde cualquier clase
    private int privado;     // Solo dentro de esta clase
    protected int protegido; // Accesible desde subclases o mismo paquete
    int amigable;            // Friendly (default): mismo paquete
}
```

Modificador `final`

- **Clase** `final` : No puede heredarse.
- **Método** `final` : No puede sobrescribirse en subclases.
- **Atributo** `final` : Valor constante.
- **Constructores**: No pueden ser `final` .

```java
final class ConstanteClase { }       // Clase final.

class EjemploFinal {
    final int valor = 10;            // Atributo final.

    final void metodoFinal() {       // Método final.
        System.out.println("Final");
    }
}
```

Acceso a Atributos y Métodos

- Uso directo o mediante métodos `get` y `set`.
- Estáticos: `Clase.metodo()`.
- De instancia: `objeto.metodo()`.

```java
public class Persona {
    public static String especie = "Humano";
    private String nombre;

    public void setNombre(String n){
     this.nombre = n;            // No hace falta que sean iguales
    }
    public String() {
     return this.nombre;         // Podría ir sin this
    }
    public void hablar() {
        System.out.println("Hola");
    }
}

// Acceso
Persona p = new Persona();
p.nombre = "Ana";
p.hablar();

System.out.println(p.getNombre());   // Método o atributo de instancia

System.out.println(Persona.especie); // Método/atributo estático
```

Constructores

Método especial para crear objetos.

- Mismo nombre que la clase.
- Sin tipo de retorno.
- Se puede sobrecargar (varios constructores).
- No son `final`.

```java
public class Animal {
    String tipo;
```

```
    public Animal(String tipo) {        // Sobrecarga
        this.tipo = tipo;
    }

    public Animal() {                   // Constructor por defecto sobreescrito ya que tiene acciones dentro y el original que crea Java no tiene contenido, y además no importa el orden en el que estén
        this.tipo = "Desconocido";
    }
}
```

Herencia

Una subclase hereda atributos y métodos de una superclase.

- Sintaxis: `class Hija extends Padre`.
- Hereda atributos y métodos accesibles.

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

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

Redefinición vs Sobrecarga

Redefinición (sobreescritura): Misma firma, diferente implementación en subclase (`@Override`).

> **Firma de un método**
>
> La firma de un método es lo que lo **identifica dentro de una clase** (especialmente para distinguirlo de otros métodos con el mismo nombre). Está compuesta por el nombre del método y la lista de ***tipos de datos de parámetros*** en orden. No incluye el tipo de retorno, modificadores (como `public`) ni excepciones.
> Por ejemplo:
>
> ```
> void saludar(String nombre)             // Firma: saludar(String)
> int saludar(String nombre, int edad)    // Firma: saludar(String, int)
> ```

Sobrecarga: Mismo nombre, diferentes parámetros. Puede ser en la misma clase o en una subclase.

```java
class Ejemplo {
    // Sobrecarga (mismo nombre, diferentes parámetros)
    void saludar() {
        System.out.println("Hola");
    }

    void saludar(String nombre) {
        System.out.println("Hola " + nombre);
    }
}

class SubEjemplo extends Ejemplo {
    @Override
    void saludar() {  // Redefinición
        System.out.println("Hola desde subclase");
    }
}
```

Uso de `super()` y `super.`

- `super()`: Llama al constructor de la superclase.
- `super.`: Accede a métodos o atributos de la superclase.

```java
class Persona {
    String nombre;

    Persona(String nombre) {
        this.nombre = nombre;
    }

    void saludar() {
        System.out.println("Hola " + nombre);
    }
}

class Estudiante extends Persona {
    String carrera;

    Estudiante(String nombre, String carrera) {
        super(nombre); // llama al constructor de Persona
        this.carrera = carrera;
    }

    void presentar() {
        super.saludar(); // llama al método saludar() de Persona
        System.out.println("Estudio " + carrera);
    }
}
```

Miembros Heredados

Los miembros heredados son las cosas que hereda una subclase al extender de otra, estos miembros son:

- **Atributos (campos)**
- **Métodos (funciones)**
- **Constructores NO se heredan** (pero se pueden usar con `super()`)
  Se accede directamente si son `public` o `protected`.
  Se usa `super.` para especificar uso de padre en caso de que haya ambigüedad.

> **En resumen**
>
> ✔️ **Métodos y atributos `protected` o `public`** se heredan y pueden ser accedidos.
> ❌ **Los `private` no son accesibles directamente**, pero existen en la subclase.
> 🛠️ **Getters y setters permiten exponer controladamente los miembros privados.**
> ❌ **Los constructores no se heredan**, pero se pueden llamar con `super()`.

```
class Persona {
    protected String nombre;
    private int edad;

    public String getNombre() {
        return nombre;
    }

    public void setNombre(String n) {
        nombre = n;
    }
    public void hablar() {
        System.out.println("Hola, soy " + nombre);
    }
}

class Estudiante extends Persona {
    void mostrar() {
        System.out.println("Nombre: " + getNombre());
    }
    public void estudiar() {
        System.out.println(nombre + " está estudiando"); // OK: 'nombre' es protected
        hablar();  // OK: método público heredado
        // edad = 20;     ❌ ERROR: edad es private
        // pensar();      ❌ ERROR: método privado no se hereda
    }
}
```

Estructura de Constructores, Set y Get

- `setNombre(tipoDeDato n)` : Asigna valor a un atributo.
- `getNombre()` : Devuelve el valor del atributo.

```
public class Persona {
    private String nombre;

    public Persona(String nombre) {
        this.nombre = nombre;
    }

    public void setNombre(String n) {
```

```
        this.nombre = n;
    }

    public String getNombre() {
        return nombre;
    }
}
```

Interfaz

Definición de métodos sin implemetación.

- Se implementan en una clase con `implements`.
- Una clase puede implemetar varias interfaces.

```
interface Volador {
    void volar(); // Método abstracto
}

class Pajaro implements Volador {
    public void volar() {
        System.out.println("El pájaro vuela");
    }
}
```

Componentes de una interfaz

- Métodos abstractos por defecto.
- Los atributos son `public static final`.
- No tienen constructor.

Abstracción

```
abstract class Figura {
    abstract double calcularArea(); // Método abstracto
}

class Cuadrado extends Figura {
    double lado;

    Cuadrado(double lado) {
        this.lado = lado;
    }

    double calcularArea() {
        return lado * lado;
    }
}
```

Clase abstracta

- No se puede instanciar.
- Puede tener métodos con o sin cuperpo.
- Puede tener constructor:
    - No se puede instanciar directamente porque la clase es abstacta.
    - Sirve para **inicializar atributos comunes**.

Ejemplo del constructor:

```
abstract class Animal {
    String nombre;

    // Constructor
    public Animal(String nombre) {
        this.nombre = nombre;
    }
}

class Perro extends Animal {
    public Perro(String nombre) {
        super(nombre); // llama al constructor de la clase abstracta
    }
}
```

Método abstracto

- No tiene cuerpo.
- Se implemeta en subclases.

UML y Relaciones

Relaciones en el código

Asociación

Relación general entre clases (una usa a otra).

```
class Profesor {
    Alumno alumno; // Profesor usa a Alumno (asociación)
}
```

Dependencia

Relación débil y temporal. Una clase **usa otra solo por un momento**, por ejemplo, como parámetro en un método.

```
class Informe {
    public void imprimir(Impresora p) {
        p.imprimirTexto("Texto del informe"); // Dependencia
    }
}
```

Asociación común (default)

Relación sin implicar propiedad ni dependencia fuerte.

```
class Cliente {
    String nombre;
}

class Pedido {
    Cliente cliente;
```

```
}
```

Agregación

Una clase contiene a otra, pero puede existir por separado.

```
class Departamento {
    Empleado[] empleados; // Contiene empleados, pero pueden existir sin el departamento
}
```

Composición

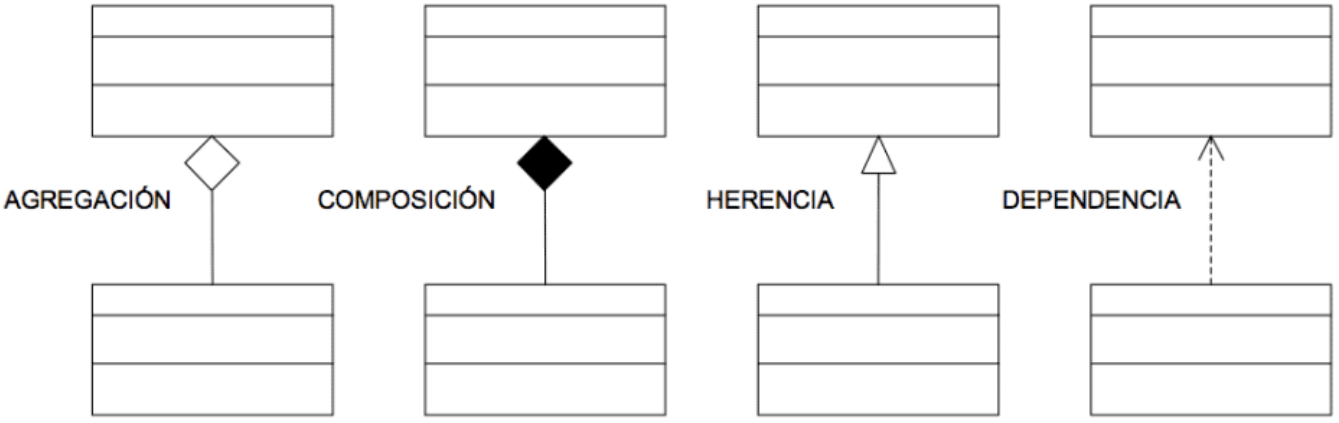
Una clase contiene a otra y depende de ella fuertemente. Si se elimina la principal, la secundaria también.

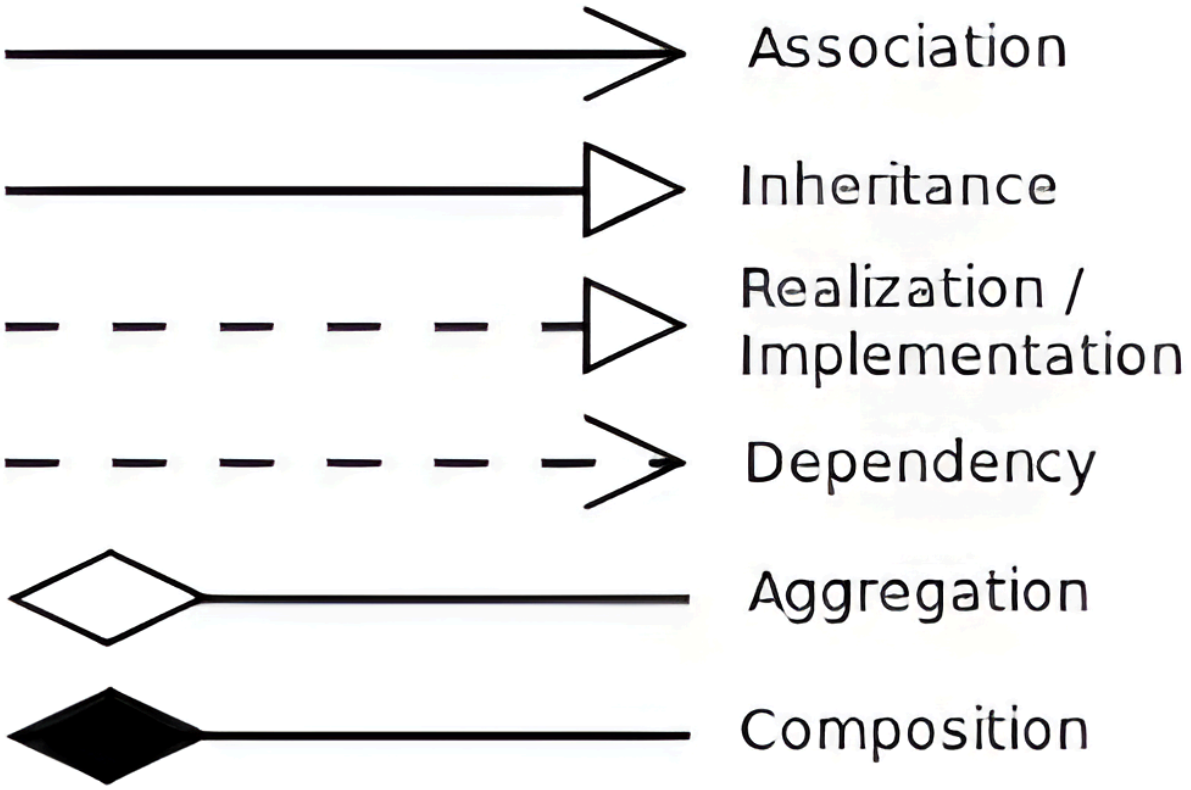
```
class Casa {
    private Habitacion habitacion = new Habitacion(); // Habitacion no existe sin Casa, se crea la instancia de la clase Habitación cuando se hace la instancia de la clase Casa
}
```

Simbología UML

- **Clases**: Rectángulo con nombre, atributos y métodos.
- **Relaciones**:
  - Línea simple: Asociación.
  - Línea punteada con una flecha simple: Dependencia.
  - Línea con rombo blanco: Agregación.
  - Línea con rombo negro: Composición.
  - Flecha con triángulo: Herencia.
  - Línea punteada con triángulo: Implementación (interface).

Ejemplos:




Association
Inheritance
Realization /
Implementation
Dependency
Aggregation
Composition