OOP - Java

Jorge Mario Londoño Peláez & Varias AI

May 7, 2025

1 Repaso Conceptos OOP - Versión Java

1.1 Objetivos de Aprendizaje

Al finalizar este módulo, serás capaz de:

- Comprender los fundamentos de la Programación Orientada a Objetos
- Implementar clases y objetos en Java
- Aplicar los principios de encapsulamiento, herencia y polimorfismo
- Utilizar interfaces y clases abstractas
- Manejar excepciones en programas orientados a objetos
- Implementar patrones de diseño básicos

1.2 Qué es la 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 pueden contener datos, en forma de campos, y código, en forma de procedimientos. Un objeto es una instancia de una clase. La POO se diferencia de la programación procedimental en que esta última se centra en la lógica del programa, mientras que la POO se centra en los objetos y sus interacciones. También se diferencia de la programación funcional en que esta última se centra en la evaluación de funciones y la inmutabilidad de los datos, mientras que la POO se centra en la mutabilidad de los objetos.

1.3 Comparación entre programación procedimental y POO

La programación procedimental se centra en funciones y procedimientos, mientras que la POO organiza el código en torno a objetos. A continuación, se muestra un ejemplo comparativo:

```
/* Programacion procedimental */
int calcularAreaRectangulo(int ancho, int alto) {
    return ancho * alto;
}
System.out.println(calcularAreaRectangulo(5, 10)); // Imprime: 50

/* Programacion orientada a objetos */
class Rectangulo {
    int ancho, alto;
    public Rectangulo(int ancho, int alto) {
```

```
        this.ancho = ancho;
        this.alto = alto;
    }
    public int calcularArea() {
        return ancho * alto;
    }
}
Rectangulo rect = new Rectangulo(5, 10);
System.out.println(rect.calcularArea()); // Imprime: 50
```

Listing 1: Comparación entre programación procedimental y POO

1.4 Clases y Objetos

En Java, una clase es una plantilla para crear objetos. Define las características (atributos) y comportamientos (métodos) que tendrán los objetos de esa clase. Un objeto es una instancia específica de una clase.

La referencia `this`

La palabra clave `this` es una referencia al objeto actual. Se utiliza para acceder a los miembros de la clase (atributos y métodos) desde dentro de la clase, especialmente cuando hay una ambigüedad de nombres (por ejemplo, cuando el nombre de un parámetro es el mismo que el de un atributo).

```
public class Coche {
    String marca;
    String modelo;
    public Coche(String marca, String modelo) {
        this.marca = marca;
        this.modelo = modelo;
    }
    public void mostrarDatos() {
        System.out.println("Marca: " + marca + ", Modelo: " + modelo);
    }
}

// Instancias de la clase Coche
Coche coche1 = new Coche("Toyota", "Corolla");
Coche coche2 = new Coche("Honda", "Civic");
coche1.mostrarDatos(); // Imprime: Marca: Toyota, Modelo: Corolla
coche2.mostrarDatos(); // Imprime: Marca: Honda, Modelo: Civic
```

Listing 2: Definicion de una clase

1.5 Sobrecarga y Sobrescritura

Sobrecarga (Overloading): Es cuando dos o más métodos en la misma clase tienen el mismo nombre pero diferentes parámetros.

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

```
}
```

Listing 3: Ejemplo de sobrecarga

Variables y métodos estáticos

Los miembros estáticos de una clase (variables y métodos) pertenecen a la clase en sí, no a las instancias de la clase. Se declaran con la palabra clave `static`. Se puede acceder a ellos directamente a través del nombre de la clase, sin necesidad de crear un objeto de la clase. Son útiles para definir constantes o métodos utilitarios que no dependen del estado de un objeto específico.

```
public class Contador {
    private static int cuenta = 0;
    public Contador() {
        cuenta++;
    }
    public static int getCuenta() {
        return cuenta;
    }
}

Contador c1 = new Contador();
Contador c2 = new Contador();
System.out.println(Contador.getCuenta()); // Imprime: 2
```

Listing 4: Ejemplo definición de una clase

1.6 Encapsulamiento

El encapsulamiento es el mecanismo que permite ocultar los detalles internos de un objeto y exponer solo la información necesaria. En Java, se logra mediante el uso de modificadores de acceso (public, private, protected). Los atributos de una clase suelen ser privados, y se accede a ellos mediante métodos públicos (getters y setters).

```
public class Persona {
    private String nombre;
    public Persona(String nombre) {
        this.nombre = nombre;
    }
    public String getNombre() {
        return nombre;
    }
    public void setNombre(String nombre) {
        this.nombre = nombre;
    }
}
```

Listing 5: Ejemplo de implementación del encapsulamiento

1.7 Herencia

La herencia es un mecanismo que permite crear nuevas clases (subclases) a partir de clases existentes (superclases). Las subclases heredan los atributos y métodos de la superclase, y pueden añadir nuevos atributos y métodos, o modificar los heredados.

```java
class Animal {
    String nombre;
    public Animal(String nombre) {
        this.nombre = nombre;
    }
    public void hacerSonido() {
        System.out.println("Sonido generico");
    }
}

class Perro extends Animal {
    public Perro(String nombre) {
        super(nombre);
    }
    @Override
    public void hacerSonido() {
        System.out.println("Guau");
    }
}

class Gato extends Animal {
    public Gato(String nombre) {
        super(nombre);
    }
    @Override
    public void hacerSonido() {
        System.out.println("Miau");
    }
}
```

Listing 6: Ejemplo de Herencia en Java: Super-clase y subclases

1.7.1 La palabra clave super

La palabra clave `super` se utiliza en una subclase para referirse a la superclase (clase padre) directamente. Tiene dos usos principales:

- **Llamar al constructor de la superclase:** Cuando se crea una instancia de una subclase, se puede usar `super()` para llamar al constructor de la superclase. Esto es útil para inicializar los atributos heredados de la superclase. Si no se llama explícitamente al constructor de la superclase, Java inserta automáticamente una llamada a `super()` sin argumentos al principio del constructor de la subclase. Si la superclase no tiene un constructor sin argumentos, es obligatorio llamar explícitamente a uno de sus constructores usando `super()`.
- **Acceder a miembros de la superclase:** Se puede usar `super.nombreDelMiembro` para acceder a un atributo o método de la superclase que ha sido ocultado (overridden) en la subclase. Esto permite acceder a la implementación original del miembro en la superclase.

```java
class Animal {
    String nombre;
    public Animal(String nombre) {
        this.nombre = nombre;
    }
    public void hacerSonido() {
        System.out.println("Sonido generico");
```

```
    }
}

class Perro extends Animal {
    String raza;
    public Perro(String nombre, String raza) {
        super(nombre); // Llama al constructor de Animal
        this.raza = raza;
    }
    @Override
    public void hacerSonido() {
        super.hacerSonido();            // Llama al metodo hacerSonido de Animal
        System.out.println("Guau");
    }
    public void mostrarNombre() {
        System.out.println("El nombre del perro es: " + super.nombre);
    }
}
```

Listing 7: Ejemplo del uso de super()

Sobrescritura (Overriding): Es cuando una subclase redefine un método de su superclase para proporcionar una implementación específica.

1.8 Clases Abstractas

Una clase abstracta es una clase que no se puede instanciar. Se utiliza como base para crear subclases concretas. Puede contener métodos abstractos (sin implementación) y métodos concretos (con implementación).

```
abstract class Figura {
    abstract double calcularArea();
}

class Circulo extends Figura {
    double radio;
    public Circulo(double radio) {
        this.radio = radio;
    }
    @Override
    double calcularArea() {
        return Math.PI * radio * radio;
    }
}
```

Listing 8: Definición de superclase abstracta y subclase concreta

1.9 Polimorfismo

El polimorfismo es la capacidad de un objeto de tomar muchas formas. En Java, se logra mediante la herencia y la implementación de interfaces. Un objeto de una subclase puede ser tratado como un objeto de su superclase.

```
Animal animal1 = new Perro("Bobby");
Animal animal2 = new Gato("Michi");
animal1.hacerSonido(); // Imprime: Guau
animal2.hacerSonido(); // Imprime: Miau
```

Listing 9: Ejemplo de Polimorfismo en Java

1.10 Interfaces

Una interfaz es un contrato que define un conjunto de métodos que una clase debe implementar. Una clase puede implementar múltiples interfaces.

Clases Abstractas vs. Interfaces

Tanto las clases abstractas como las interfaces se utilizan para definir comportamientos que las clases deben implementar, pero existen diferencias clave:

- **Implementación:** Una clase abstracta puede contener tanto métodos abstractos (sin implementación) como métodos concretos (con implementación). Una interfaz solo puede contener métodos abstractos (o métodos default con implementación en Java 8+).
- **Herencia Múltiple:** Una clase solo puede heredar de una clase abstracta, pero puede implementar múltiples interfaces.
- **Variables:** Una clase abstracta puede tener variables de instancia, mientras que una interfaz solo puede tener constantes (variables `static final`).
- **Uso:** Las clases abstractas se utilizan para definir una jerarquía de clases con un comportamiento base común, mientras que las interfaces se utilizan para definir un contrato que las clases deben cumplir, independientemente de su posición en la jerarquía de clases.

Cuándo usar una clase abstracta:

- Cuando se quiere definir una jerarquía de clases con un comportamiento base común.
- Cuando se quiere proporcionar una implementación parcial de algunos métodos.
- Cuando se necesita tener variables de instancia.

Cuándo usar una interfaz:

- Cuando se quiere definir un contrato que las clases deben cumplir, independientemente de su posición en la jerarquía de clases.
- Cuando se quiere permitir que una clase implemente múltiples comportamientos (herencia múltiple de tipos).
- Cuando solo se necesita definir métodos abstractos y constantes.

```java
interface Sonido {
    void hacerSonido();
}

class Pato implements Sonido {
    @Override
    public void hacerSonido() {
        System.out.println("Cuac");
    }
}
```

Listing 10: Definicion e implementacion de una interfaz en Java

1.11 Excepciones

Las excepciones son eventos que interrumpen el flujo normal de un programa. En Java, se utilizan bloques try-catch para atrapar excepciones, y la palabra clave throw para lanzar excepciones.

```
try {
    int resultado = 10 / 0; // Lanza una ArithmeticException
} catch (ArithmeticException e) {
    System.out.println("Error: Division por cero");
}

// Ejemplo de lanzar una excepcion
public void verificarEdad(int edad) throws IllegalArgumentException {
    if (edad < 0) {
        throw new IllegalArgumentException("La edad no puede ser negativa");
    }
}
```

Listing 11: Ejemplo de manejo de excepciones

1.12 Clases internas o anidadas

Una clase interna (o clase anidada) es una clase que se define dentro de otra clase. Las clases internas pueden ser de varios tipos: clases miembro (no estáticas), clases estáticas, clases locales (definidas dentro de un método) y clases anónimas (sin nombre). Las clases internas tienen acceso a los miembros de la clase externa, incluso si son privados. Se utilizan para agrupar clases relacionadas y para implementar patrones de diseño como el patrón de iterador.

```
    public class ClaseExterna {
    private int x = 10;
    class ClaseInterna {
        public void mostrar() {
            System.out.println("x = " + x);
        }
    }
    public static void main(String[] args) {
        ClaseExterna externa = new ClaseExterna();
        ClaseExterna.ClaseInterna interna = externa.new ClaseInterna();
        interna.mostrar(); // Imprime: x = 10
    }
}
```

Listing 12: Ejemplo de una clase interna

2 Diagramas y Visualizaciones

2.1 Diagramas de Clases

Los diagramas de clases son una herramienta fundamental para visualizar la estructura de un sistema orientado a objetos. A continuación se muestran algunos ejemplos:

```
@startuml
class Animal {
    -nombre: String
    +hacerSonido(): void
}

```

```
class Perro {
    -raza: String
    +hacerSonido(): void
}

class Gato {
    -color: String
    +hacerSonido(): void
}

Animal <|-- Perro
Animal <|-- Gato
@enduml
```

Listing 13: Ejemplo de diagrama de clases en formato PlantUML

2.2 Relaciones entre Clases

En OOP, las clases pueden tener diferentes tipos de relaciones:

- **Herencia (is-a)**: Una clase hereda de otra (ej: Perro es un Animal)
- **Composición (has-a)**: Una clase contiene instancias de otra (ej: Coche tiene un Motor)
- **Asociación (uses-a)**: Una clase utiliza otra (ej: Estudiante usa Libro)
- **Agregación (part-of)**: Una clase es parte de otra (ej: Rueda es parte de Coche)

2.3 Flujo de Ejecución

Es importante entender cómo se ejecuta el código en un programa orientado a objetos:

1. Se crea una instancia de la clase (objeto)
2. Se inicializan los atributos
3. Se ejecutan los métodos cuando son llamados
4. Los objetos interactúan entre sí mediante mensajes

3 Ejercicios de Repaso

3.1 Ejercicios Básicos

1. **Libro y Biblioteca**
   - Crea una clase `Libro` con atributos como `titulo`, `autor`, `isbn` y `precio`.
   - Implementa métodos para mostrar los detalles del libro y calcular el precio con descuento.
   - Crea una clase `Biblioteca` que pueda almacenar múltiples libros y realizar búsquedas por título o autor.
2. **Sistema de Empleados**
   - Implementa una jerarquía de clases para diferentes tipos de empleados:

- `Empleado` (clase base con nombre, salario base)
- `Desarrollador` (con especialidad y nivel)
- `Gerente` (con departamento y bonificación)

- Implementa el cálculo de salario considerando bonificaciones y deducciones.
- Añade métodos para mostrar la información del empleado y calcular su antigüedad.

3. **Sistema de Vehículos**

- Crea una interfaz `Vehiculo` con métodos como `mover()`, `detener()` y `obtenerVelocidad()`.
- Implementa esta interfaz en clases como `Coche`, `Bicicleta` y `Motocicleta`.
- Añade características específicas para cada tipo de vehículo (combustible, marchas, etc.).

3.2 Ejercicios Intermedios

1. **Sistema de Reservas**

- Crea un sistema de reservas para un hotel con las siguientes clases:
  - `Habitacion` (número, tipo, precio)
  - `Cliente` (datos personales, historial de reservas)
  - `Reserva` (fechas, habitación, cliente)
- Implementa la lógica para verificar disponibilidad y calcular el costo total.
- Maneja excepciones para casos como habitaciones no disponibles o fechas inválidas.

2. **Red Social Simple**

- Implementa un sistema básico de red social con:
  - `Usuario` (perfil, amigos, publicaciones)
  - `Publicacion` (contenido, fecha, likes)
  - `Comentario` (texto, autor, fecha)
- Implementa funcionalidades como añadir amigos, crear publicaciones y comentar.
- Utiliza colecciones para manejar las relaciones entre objetos.

3.3 Ejercicios Avanzados

1. **Sistema de Gestión de Proyectos**

- Crea un sistema para gestionar proyectos de software con:
  - `Proyecto` (nombre, fecha inicio/fin, presupuesto)
  - `Tarea` (descripción, estado, asignado)
  - `Equipo` (miembros, roles)
- Implementa el patrón Observer para notificar cambios en el estado de las tareas.
- Utiliza el patrón Factory para crear diferentes tipos de tareas.

2. **Simulador de Banco**

- Implementa un sistema bancario con:

  - `Cuenta` (abstracta) con subclases `CuentaCorriente` y `CuentaAhorro`
  - `Cliente` (con múltiples cuentas)
  - `Transaccion` (depósitos, retiros, transferencias)
- Implementa manejo de excepciones para operaciones inválidas.
- Utiliza el patrón Singleton para el registro de transacciones.

3.4 Consejos para los Ejercicios

- Comienza con un diseño en papel antes de escribir código.
- Implementa primero la funcionalidad básica y luego añade características más complejas.
- Escribe pruebas unitarias para verificar el comportamiento de tus clases.
- Utiliza diagramas UML para visualizar las relaciones entre clases.
- Documenta tu código con comentarios explicativos.

4 Buenas prácticas

4.1 Principios SOLID

Los principios SOLID son un conjunto de cinco principios de diseño de software que ayudan a crear código más mantenible y escalable:

- **S**ingle Responsibility Principle (SRP): Una clase debe tener una única razón para cambiar.
- **O**pen/Closed Principle (OCP): Las entidades de software deben estar abiertas para su extensión, pero cerradas para su modificación.
- **L**iskov Substitution Principle (LSP): Los objetos de una superclase deben poder ser reemplazados por objetos de sus subclases sin afectar la funcionalidad del programa.
- **I**nterface Segregation Principle (ISP): Los clientes no deben depender de interfaces que no utilizan.
- **D**ependency Inversion Principle (DIP): Los módulos de alto nivel no deben depender de módulos de bajo nivel. Ambos deben depender de abstracciones.

4.2 Patrones de Diseño Comunes

- **Singleton**: Asegura que una clase tenga una única instancia.
- **Factory**: Proporciona una interfaz para crear objetos sin especificar sus clases concretas.
- **Observer**: Define una dependencia uno a muchos entre objetos.
- **Strategy**: Define una familia de algoritmos, encapsula cada uno y los hace intercambiables.

4.3 Depuración de Código OOP

- Utiliza el depurador de tu IDE para inspeccionar el estado de los objetos.
- Implementa logging para rastrear el flujo de ejecución.
- Utiliza pruebas unitarias para verificar el comportamiento de tus clases.
- Aprende a leer y entender los mensajes de error comunes en Java.

5 Referencias

w3schoools: Java OOP
talent500: What is Object Oriented Programming(OOPs) in Java
Javaguides: Java Object-Oriented Programming (OOP) Cheat Sheet