

OOP - Java

Jorge Mario Londoño Peláez & Varias AI

February 5, 2025

1 Repaso Conceptos OOP - Versión Java

1.1 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.

1.2 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).

```
1 public class Coche {
2     String marca;
3     String modelo;
4     public Coche(String marca, String modelo) {
5         this.marca = marca;
6         this.modelo = modelo;
7     }
8     public void mostrarDatos() {
9         System.out.println("Marca: " + marca + ", Modelo: " + modelo);
10    }
11 }
12
13 // Instancias de la clase Coche
14 Coche coche1 = new Coche("Toyota", "Corolla");
15 Coche coche2 = new Coche("Honda", "Civic");
16 coche1.mostrarDatos(); // Imprime: Marca: Toyota, Modelo: Corolla
17 coche2.mostrarDatos(); // Imprime: Marca: Honda, Modelo: Civic
```

Listing 1: Definición de una clase

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.

```
1 public class Contador {
2     private static int cuenta = 0;
3     public Contador() {
4         cuenta++;
5     }
6     public static int getCuenta() {
7         return cuenta;
8     }
9 }
10
11 Contador c1 = new Contador();
12 Contador c2 = new Contador();
13 System.out.println(Contador.getCuenta()); // Imprime: 2
```

Listing 2: Ejemplo definición de una clase

1.3 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).

```
1 public class Persona {
2     private String nombre;
3     public Persona(String nombre) {
4         this.nombre = nombre;
5     }
6     public String getNombre() {
7         return nombre;
8     }
9     public void setNombre(String nombre) {
10        this.nombre = nombre;
11    }
12 }
```

Listing 3: Ejemplo de implementación del encapsulamiento

1.4 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.

```
1 class Animal {
2     String nombre;
3     public Animal(String nombre) {
4         this.nombre = nombre;
5     }
6     public void hacerSonido() {
```

```

7     System.out.println("Sonido generico");
8 }
9 }
10
11 class Perro extends Animal {
12     public Perro(String nombre) {
13         super(nombre);
14     }
15     @Override
16     public void hacerSonido() {
17         System.out.println("Guau");
18     }
19 }
20
21 class Gato extends Animal {
22     public Gato(String nombre) {
23         super(nombre);
24     }
25     @Override
26     public void hacerSonido() {
27         System.out.println("Miau");
28     }
29 }

```

Listing 4: Ejemplo de Herencia en Java: Super-clase y subclases

1.4.1 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.

```

1 class Animal {
2     String nombre;
3     public Animal(String nombre) {
4         this.nombre = nombre;
5     }
6     public void hacerSonido() {
7         System.out.println("Sonido generico");
8     }
9 }
10
11 class Perro extends Animal {
12     String raza;
13     public Perro(String nombre, String raza) {

```

```

14     super(nombre); // Llama al constructor de Animal
15     this.raza = raza;
16 }
17 @Override
18 public void hacerSonido() {
19     super.hacerSonido(); // Llama al metodo hacerSonido de Animal
20     System.out.println("Guau");
21 }
22 public void mostrarNombre() {
23     System.out.println("El nombre del perro es: " + super.nombre);
24 }
25 }

```

Listing 5: Ejemplo del uso de super()

1.5 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).

```

1 abstract class Figura {
2     abstract double calcularArea();
3 }
4
5 class Circulo extends Figura {
6     double radio;
7     public Circulo(double radio) {
8         this.radio = radio;
9     }
10    @Override
11    double calcularArea() {
12        return Math.PI * radio * radio;
13    }
14 }

```

Listing 6: Definición de superclase abstracta y subclase concreta

1.6 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.

```

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

```

Listing 7: Ejemplo de Polimorfismo en Java

1.7 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.

```
1 interface Sonido {
2     void hacerSonido();
3 }
4
5 class Pato implements Sonido {
6     @Override
7     public void hacerSonido() {
8         System.out.println("Cuac");
9     }
10 }
```

Listing 8: Definición e implementación de una interfaz en Java

1.8 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.

```
1 try {
2     int resultado = 10 / 0; // Lanza una ArithmeticException
3 } catch (ArithmeticException e) {
4     System.out.println("Error: Division por cero");
5 }
6
7 // Ejemplo de lanzar una excepcion
8 public void verificarEdad(int edad) throws IllegalArgumentException {
9     if (edad < 0) {
10         throw new IllegalArgumentException("La edad no puede ser negativa");
11     }
12 }
```

Listing 9: Ejemplo de manejo de excepciones

1.9 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.

```
1 public class ClaseExterna {
2     private int x = 10;
3     class ClaseInterna {
4         public void mostrar() {
5             System.out.println("x = " + x);
6         }
7     }
8     public static void main(String[] args) {
9         ClaseExterna externa = new ClaseExterna();
10        ClaseExterna.ClaseInterna interna = externa.new ClaseInterna();
11        interna.mostrar(); // Imprime: x = 10
12    }
13 }
```

Listing 10: Ejemplo de una clase interna

2 Referencias

w3schools: [Java OOP](#)

talent500: [What is Object Oriented Programming\(OOPs\) in Java](#)

Javaguides: [Java Object-Oriented Programming \(OOP\) Cheat Sheet](#)