Programación Orientada a Objetos en Java

PROFESOR JUAN LUIS HERENCIA GUERRA

Programación Orientada a Objetos en Java

Acerca del profesor Juan Luis Herencia Guerra

Con maestría de especialización en Computer Science, graduado en la Universidad Nacional de Ingeniería con muchos años de experiencia en dirección y desarrollo de software en diferentes lenguajes y arquitecturas para las empresas privadas y públicas.

Conceptos de POO

APLICADOS A JAVA

Abstracción

- La abstracción es el proceso de ocultar los detalles de implementación y mostrar solo la funcionalidad al usuario. En otras palabras, se centra en lo que hace un objeto en lugar de como lo hace.

```java
// Definición de una clase abstracta
abstract class Animal {
    abstract void makeSound();
}

// Implementación de la clase abstracta
class Dog extends Animal {
    void makeSound() {
        System.out.println("Woof");
    }
}
```

Definición de Clase

- Una clase en Java es una plantilla para crear objetos. Define un tipo de objeto de acuerdo a sus atributos y métodos.

```java
public class Persona {
    // Atributos
    String nombre;
    int edad;

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

    // Métodos
    void mostrarDetalles() {
        System.out.println("Nombre: " + nombre + ", Edad: " + edad);
    }
}
```

Objeto e Instancia

- Un objeto es una instancia de una clase. En términos simples, es una entidad con estado y comportamiento definido por su clase.

```java
public class Main {
    public static void main(String[] args) {
        // Crear una instancia de la clase Persona
        Persona persona1 = new Persona("Juan", 25);
        persona1.mostrarDetalles();
    }
}
```

Implementación

- La implementación es la creación del código dentro de un método que realiza el comportamiento definido por una clase o una interfaz.

```java
public class Main {
    public static void main(String[] args) {
        // Crear una instancia de la clase Persona
        Persona persona1 = new Persona("Juan", 25);
        persona1.mostrarDetalles();
    }
}
```

Atributos

- Los atributos son las variables dentro de una clase que representan las propiedades del objeto.

```java
public class Libro {
    // Atributos
    String titulo;
    String autor;
    int numPaginas;

    // Constructor
    public Libro(String titulo, String autor, int numPaginas) {
        this.titulo = titulo;
        this.autor = autor;
        this.numPaginas = numPaginas;
    }
}
```

Operaciones

- Las operaciones son los métodos dentro de una clase que definen el comportamiento de los objetos.

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

Operador ‘new’

- El operador ‘new’ en Java se utiliza para crear nuevos objetos de una clase.

```java
public class Main {
    public static void main(String[] args) {
        // Uso del operador new
        Persona persona2 = new Persona("Ana", 30);
        persona2.mostrarDetalles();
    }
}
```

Paquetes

- Los paquetes son un mecanismo para organizar las clases e interfaces en grupos lógicos y evitar conflictos de nombres. Físicamente los archivos de las clases en común se organizan en fólderes de archivos.

```java
// Definición del paquete
package com.ejemplo;

// Clase dentro del paquete
public class MiClase {
    void mostrarMensaje() {
        System.out.println("Hola desde el paquete com.ejemplo");
    }
}
```

Paquetes

- Los paquetes son un mecanismo para organizar las clases e interfaces en grupos lógicos y evitar conflictos de nombres. Físicamente los archivos de las clases en común se organizan en fólderes de archivos.

```java
// Uso del paquete
package com.otroejemplo;

import com.ejemplo.MiClase;

public class Main {
    public static void main(String[] args) {
        MiClase obj = new MiClase();
        obj.mostrarMensaje();
    }
}
```

Clase

ESTUDIO DE LA CLASE EN JAVA

Miembros de Clase

- Los miembros de clase son los atributos y métodos que pertenecen a una clase. Pueden ser tanto variables como métodos.

```java
public class Ejemplo {
    // Miembros de clase: variables
    int numero;
    String texto;

    // Miembros de clase: métodos
    void mostrar() {
        System.out.println("Número: " + numero + ", Texto: " + texto);
    }
}
```

Variables

- Las variables en Java pueden ser de instancia, de clase (estáticas) o locales. Las variables de instancia pertenecen a una instancia de la clase, las variables estáticas pertenecen a la clase en sí, y las variables locales son las definidas dentro de métodos.

```java
public class Ejemplo {
    // Variable de instancia
    int numero;

    // Variable estática (de clase)
    static int contador;

    void incrementar() {
        // Variable local
        int incremento = 1;
        numero += incremento;
        contador += incremento;
    }
}
```

Métodos

- Los métodos en Java son bloques de código que realizan una tarea específica y pueden recibir parámetros y devolver valores.

```java
public class Calculadora {
    // Método sin parámetros y sin valor de retorno
    void saludar() {
        System.out.println("Hola");
    }

    // Método con parámetros y con valor de retorno
    int sumar(int a, int b) {
        return a + b;
    }
}
```

Control de Acceso

- Java proporciona diferentes niveles de control de acceso para los miembros de clase (atributos y métodos): public, private, protected y paquete (sin modificador).

```java
public class Ejemplo {
    // Acceso público
    public int publico;

    // Acceso privado
    private int privado;

    // Acceso protegido
    protected int protegido;

    // Acceso por defecto (paquete)
    int porDefecto;

    // Métodos de acceso
    public void setPrivado(int valor) {
        privado = valor;
    }

    public int getPrivado() {
        return privado;
    }
}
```

Acceso Protegido

- El acceso protected permite que los miembros de la clase sean accesibles dentro del mismo paquete y por las subclases, incluso si están en paquetes diferentes.

```java
package paquete1;

public class ClaseBase {
    protected int valorProtegido;

    protected void metodoProtegido() {
        System.out.println("Método protegido en ClaseBase");
    }
}
```

Acceso Protegido

- El acceso protected permite que los miembros de la clase sean accesibles dentro del mismo paquete y por las subclases, incluso si están en paquetes diferentes.

```java
// En otro paquete
package paquete2;

import paquete1.ClaseBase;

public class SubClase extends ClaseBase {
    void mostrar() {
        valorProtegido = 10;  // Acceso permitido
        metodoProtegido();    // Acceso permitido
    }
}
```

Encapsulación

- La encapsulación es el mecanismo de restringir el acceso directo a algunos componentes de un objeto y solo permitir su modificación a través de métodos definidos. Se logra usando modificadores de acceso y métodos getter y setter.

```java
public class Persona {
    private String nombre;
    private int edad;

    // Método getter
    public String getNombre() {
        return nombre;
    }

    // Método setter
    public void setNombre(String nombre) {
        this.nombre = nombre;
    }

    public int getEdad() {
        return edad;
    }

    public void setEdad(int edad) {
        if (edad > 0) {
            this.edad = edad;
        }
    }
}
```

Constructor

- Un constructor es un método especial que se invoca automáticamente cuando se crea una instancia de una clase. Tiene el mismo nombre que la clase y no tiene un tipo de retorno.

```java
public class Persona {
    String nombre;
    int edad;

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

    void mostrarDetalles() {
        System.out.println("Nombre: " + nombre + ", Edad: " + edad);
    }
}
```

Destructor

- Java no tiene destructores como en otros lenguajes (como C++), pero tiene el método finalize() que puede ser sobreescrito para realizar acciones de limpieza antes de que un objeto sea recolectado por el recolector de basura. Sin embargo, el uso de finalize() es desalentado en favor de otros mecanismos de gestión de recursos, como el uso de bloques try-with-resources.

```java
public class Ejemplo {
    // Método finalize
    @Override
    protected void finalize() throws Throwable {
        try {
            System.out.println("Objeto recolectado por el recolector de basura");
        } finally {
            super.finalize();
        }
    }

    public static void main(String[] args) {
        Ejemplo obj = new Ejemplo();
        obj = null;
        System.gc(); // Sugerencia para ejecutar el recolector de basura
    }
}
```

Destructor con try-with-resources

- En Java, el uso de try-with-resources es una forma moderna y recomendada de gestionar recursos que necesitan ser cerrados después de su uso, como archivos, conexiones de bases de datos, etc. Las clases que implementan la interfaz AutoCloseable pueden ser usadas en un bloque try-with-resources, y su método close() se llamará automáticamente al final del bloque, asegurando que los recursos se liberen adecuadamente.

```java
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

public class EjemploTryWithResources {
    public static void main(String[] args) {
        // Usar try-with-resources para asegurar el cierre del recurso
        try (BufferedReader br = new BufferedReader(new FileReader("archivo.txt"))) {
            String linea;
            while ((linea = br.readLine()) != null) {
                System.out.println(linea);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Destructor con try-with-resources

- Si se quiere definir que la propia clase se pueda usar con try-with-resources, simplemente implemente una interfaz AutoCloseable y proporcione una implementación del método close().

```java
public class RecursoPersonalizado implements AutoCloseable {
    public void usarRecurso() {
        System.out.println("Usando el recurso");
    }

    @Override
    public void close() {
        System.out.println("Cerrando el recurso");
    }

    public static void main(String[] args) {
        try (RecursoPersonalizado recurso = new RecursoPersonalizado()) {
            recurso.usarRecurso();
        }
    }
}
```

Sobrecarga y alcance de una clase

Objetivo

- El objetivo se refiere al propósito general o la meta que se busca alcanzar en el diseño de una clase o método. En Java, el objetivo de una clase puede ser representar una entidad específica y su comportamiento.

```java
public class Vehiculo {
    String tipo;
    int velocidad;

    // Objetivo: Representar un vehículo y sus propiedades
}
```

Contexto

- El contexto se refiere al entorno o escenario en el cual se utiliza una clase o método. Esto incluye la relación de la clase con otras clases y su uso dentro de una aplicación.

```java
public class Main {
    public static void main(String[] args) {
        // Contexto: Crear y usar una instancia de Vehiculo
        Vehiculo coche = new Vehiculo();
        coche.tipo = "Coche";
        coche.velocidad = 120;
    }
}
```

Sobrecarga de Métodos

- La sobrecarga de métodos es una característica que permite definir múltiples métodos con el mismo nombre, pero con diferentes parámetros.

```java
public class Calculadora {
    // Sobrecarga de métodos
    public int sumar(int a, int b) {
        return a + b;
    }

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

Sobrecarga de Constructores

- La sobrecarga de constructores permite definir múltiples constructores con diferentes parámetros para inicializar objetos de diferentes maneras.

```java
public class Persona {
    String nombre;
    int edad;

    // Constructor sin parámetros
    public Persona() {
        this.nombre = "Desconocido";
        this.edad = 0;
    }

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

Métodos con Argumentos Variables

- Los métodos con argumentos variables (varargs) permiten que un método acepte cero o más argumentos de un tipo específico.

```java
public class VarargsEjemplo {
    public static void imprimirNumeros(int... numeros) {
        for (int numero : numeros) {
            System.out.println(numero);
        }
    }

    public static void main(String[] args) {
        imprimirNumeros(1, 2, 3, 4, 5);  // Output: 1 2 3 4 5
    }
}
```

JUAN LUIS HERENCIA GUERRA

Alcance de Instancia

- El alcance de instancia se refiere a las variables y métodos que pertenecen a una instancia específica de una clase. Se accede a ellas a través de los objetos de la clase.

```java
public class InstanciaEjemplo {
    int numero;  // Variable de instancia

    void imprimirNumero() {  // Método de instancia
        System.out.println(numero);
    }

    public static void main(String[] args) {
        InstanciaEjemplo obj = new InstanciaEjemplo();
        obj.numero = 10;
        obj.imprimirNumero();  // Output: 10
    }
}
```

Alcance de Clase

- El alcance de clase se refiere a las variables y métodos que pertenecen a la clase en sí, en lugar de a las instancias de la clase. Se accede a ellos a través del nombre de la clase.

```java
public class ClaseEjemplo {
    static int contador;  // Variable de clase

    static void incrementarContador() {  // Método de clase
        contador++;
    }

    public static void main(String[] args) {
        ClaseEjemplo.incrementarContador();
        System.out.println(ClaseEjemplo.contador);  // Output: 1
    }
}
```

Acceso a Variables y Métodos

- Las variables y métodos pueden tener diferentes niveles de acceso: public, private, protected y paquete (sin modificador).

```java
public class AccesoEjemplo {
    public int publico;
    private int privado;
    protected int protegido;
    int porDefecto;

    public void metodoPublico() {}
    private void metodoPrivado() {}
    protected void metodoProtegido() {}
    void metodoPorDefecto() {}
}
```

Inicializador Estático

- Un inicializador estático es un bloque de código que se ejecuta una vez cuando la clase es cargada en memoria. Se utiliza para inicializar variables estáticas.

```java
public class InicializadorEstaticoEjemplo {
    static int contador;

    // Inicializador estático
    static {
        contador = 100;
        System.out.println("Inicializador estático ejecutado");
    }

    public static void main(String[] args) {
        System.out.println("Contador: " + contador);
    }
}
```