CONSULTA EVALUABLE

**1) En java Modificadores**

los modificadores son palabras clave que se utilizan para definir la accesibilidad y el alcance de las clases, métodos, variables y otros elementos de un programa. A continuación, se presentan los distintos tipos de modificadores que se pueden utilizar en Java:

Modificadores de acceso: son aquellos que determinan el nivel de acceso a una clase, método o variable. Hay cuatro tipos de modificadores de acceso en Java: Public, protected, private y default (sin especificar). Estos modificadores determinan si un elemento puede ser accedido desde otras clases o paquetes.

Modificadores de no acceso: son aquellos que se utilizan para especificar ciertas características adicionales de un elemento. Algunos ejemplos de modificadores de no acceso son static, final, abstract y synchronized. Estos modificadores pueden utilizarse en combinación con los modificadores de acceso para definir el comportamiento y la funcionalidad de los elementos del programa.

Modificadores de clase: son aquellos que se utilizan para definir el comportamiento de una clase. Algunos ejemplos de modificadores de clase son abstract, final y strictfp. Estos modificadores se aplican a la clase en sí misma, no a sus métodos o variables.

Modificadores de método: son aquellos que se utilizan para definir el comportamiento de un método. Algunos ejemplos de modificadores de método son abstract, final, static y synchronized. Estos modificadores se aplican a un método específico dentro de una clase.

Modificadores de variable: son aquellos que se utilizan para definir el comportamiento de una variable. Algunos ejemplos de modificadores de variable son final y volatile. Estos modificadores se aplican a una variable específica dentro de una clase o método.

Para utilizar los modificadores en Java, se debe especificar la palabra clave correspondiente antes del elemento que se desea modificar. Por ejemplo, para definir una clase pública, se utiliza la palabra clave "Public" antes de la palabra clave "class".

EJEMPLO

class MiObjeto {

private short valor = 0;

MiObjeto(MiObjeto otro) {

valor = otro.valor;

}

}

**• Encapsulación**

La encapsulación en Java es uno de los cuatro conceptos fundamentales de la programación orientada a objetos (POO), junto con la abstracción, la herencia y el polimorfismo.

La encapsulación es el proceso de ocultar los detalles internos de un objeto y proporcionar una interfaz pública para interactuar con él. Esto se logra mediante el uso de modificadores de acceso, como Public, private y protected, que definen quién puede acceder a los campos y métodos de un objeto.

La encapsulación ayuda a garantizar la integridad de los datos al evitar que los usuarios de una clase manipulen directamente sus campos internos. En su lugar, los usuarios deben utilizar los métodos públicos de la clase para acceder y modificar los datos. Además, la encapsulación facilita la modificación de una clase sin afectar su uso en otras partes del programa.

Por ejemplo, en la siguiente clase de ejemplo, el campo "saldo" está encapsulado y solo se puede acceder a él a través de los métodos públicos.

EJEMPLO

"getSaldo" y "setSaldo":

public class CuentaBancaria {

private double saldo;

public double getSaldo() {

return saldo;

}

public void setSaldo(double nuevoSaldo) {

saldo = nuevoSaldo;

}

}

En este ejemplo, el campo "saldo" está marcado como privado para evitar que se modifique directamente desde fuera de la clase. En su lugar, los usuarios pueden obtener el saldo actual llamando al método público "getSaldo", o actualizarlo llamando al método público "setSaldo".

**• Paquetes / Api**

Los paquetes y APIs son herramientas que permiten a los programadores acceder a una serie de funciones y recursos que les permiten desarrollar aplicaciones y software más fácilmente.

Un ejemplo común de API es la API de Google Maps, que proporciona a los desarrolladores acceso a mapas y datos geográficos que pueden integrar en sus propias aplicaciones. Otro ejemplo es la API de Twitter, que permite a los desarrolladores acceder a datos de Twitter y utilizarlos en sus propias aplicaciones.

En cuanto a paquetes, estos pueden variar según el lenguaje de programación utilizado. Por ejemplo, en el lenguaje de programación Python, uno de los paquetes más populares es NumPy, que proporciona herramientas para realizar cálculos numéricos complejos en matrices y arreglos. Otro paquete popular es Pandas, que proporciona herramientas para el análisis y la manipulación de datos.

Un ejemplo de uso de un paquete en Python podría ser el siguiente:

import numpy as np

# Crear una matriz de 2x2

matriz = np.array([[1, 2], [3, 4]])

# Calcular la media de todos los elementos de la matriz

media = np.mean(matriz)

# Imprimir la matriz y la media

print("Matriz:")

print(matriz)

print("Media:")

print(media)

En este ejemplo, estamos utilizando el paquete NumPy para crear una matriz de 2x2 y calcular la media de todos sus elementos. Al importar NumPy con import numpy as np, podemos usar las funciones del paquete utilizando np.<funcion>(), como se ve en las líneas 4 y 6. El resultado de la ejecución del programa sería el siguiente:

Matriz:

[[1 2]

[3 4]]

Media:

2.5

**• Herencia**

La herencia es un concepto importante en la programación orientada a objetos, que permite crear nuevas clases basadas en clases existentes, y heredar sus atributos y métodos. Un ejemplo de herencia podría ser la creación de una clase "Animal", que tenga atributos como "nombre", "edad" y "especie", y métodos como "moverse" y "emitir sonidos". Luego, se podría crear una clase "Perro", que herede de la clase "Animal", y tenga atributos adicionales como "raza" y métodos específicos como "ladrar". De esta manera, la clase "Perro" tendría acceso a todos los atributos y métodos de la clase "Animal", así como a los propios que se le han añadido.

EJEMPLO

class Animal:

def \_\_init\_\_(self, nombre, edad):

self.nombre = nombre

self.edad = edad

def comer(self):

print(f"{self.nombre} está comiendo.")

def dormir(self):

print(f"{self.nombre} está durmiendo.")

Y luego, la clase "Perro" podría heredar de esta clase de la siguiente manera:

class Perro(Animal):

def \_\_init\_\_(self, nombre, edad, raza):

super().\_\_init\_\_(nombre, edad)

self.raza = raza

def ladrido(self):

print(f"{self.nombre} está ladrando.")

De esta manera, la clase "Perro" tendría acceso a los métodos "comer()" y "dormir()" de la clase "Animal", y también tendría su propio método "ladrido()".

**• Polymorfismo**

En Java es necesario que las clases compartan una superclase común para implementar el polimorfismo, luego veremos que también se puede implementar el polimorfismo en Java mediante interfaces. Con el polimorfismo podemos implementar programas que luego son fácilmente extensibles.

EJEMPLO

public class Polimorfismo { public static void main(String[] args) {

mostrarobjeto (new Rectangulo (1, 1, "negro", true)); mostrarobjeto (new Circulo (1, "rojo", false));

}

public static void mostrarobjeto (FiguraGeometrica object) {

System.out.println("Creado en + object.obtenerFechaCreado () +

M. suncolor es + object.obtenerColor())

• **Clases inner (Anidadas)**

Las clases internas o anidadas (también conocidas como clases dentro de clases) son clases que están definidas dentro de otra clase. Estas clases tienen acceso a los miembros privados de la clase externa y se utilizan a menudo para encapsular una funcionalidad relacionada con la clase externa.

Aquí hay un ejemplo de cómo se podría utilizar una clase interna en Java para modelar un juego de cartas:

EJEMPLO

public class Baraja {

private Carta[] cartas;

// Constructor que inicializa el mazo con 52 cartas

public Baraja() {

cartas = new Carta[52];

int index = 0;

for (int r = 1; r <= 13; r++) {

for (int s = 1; s <= 4; s++) {

cartas[index++] = new Carta(r, s);

}

}

}

// Método para mezclar las cartas

public void mezclar() {

Random rnd = new Random();

for (int i = 51; i > 0; i--) {

int j = rnd.nextInt(i + 1);

Carta temp = cartas[i];

cartas[i] = cartas[j];

cartas[j] = temp;

}

}

// Clase interna para representar una carta

private class Carta {

private int rango;

private int palo;

public Carta(int r, int s) {

rango = r;

palo = s;

}

public String toString() {

String[] rangos = {"As", "2", "3", "4", "5", "6", "7", "8", "9", "10", "J", "Q", "K"};

String[] palos = {"Picas", "Corazones", "Diamantes", "Tréboles"};

return rangos[rango-1] + " de " + palos[palo-1];

}

}

// Método para imprimir las cartas en la baraja

public void imprimir() {

for (Carta c : cartas) {

System.out.println(c.toString());

}

}

// Método para jugar la primera carta de la baraja

public Carta jugar() {

Carta primeraCarta = cartas[0];

System.arraycopy(cartas, 1, cartas, 0, cartas.length-1);

cartas[cartas.length-1] = null;

return primeraCarta;

}

}

En este ejemplo, la clase Baraja contiene una clase interna llamada Carta que representa una carta del juego. La clase Carta tiene acceso a los miembros privados de

**• Clases abstractas**

Las clases abstractas son clases que no pueden ser instanciadas directamente, sino que se utilizan como modelos para crear otras clases que sí pueden ser instanciadas. Estas clases pueden tener métodos abstractos, los cuales no tienen una implementación definida en la clase abstracta, sino que deben ser implementados por las clases hijas que la extienden.

Un ejemplo de clase abstracta puede ser una clase Animal, la cual puede tener métodos como "comer", "dormir" y "moverse". Sin embargo, esta clase abstracta no tiene una implementación concreta para cada uno de estos métodos, ya que los animales pueden comer, dormir y moverse de maneras muy diferentes. En cambio, cada clase hija que extienda la clase Animal deberá implementar estos métodos de acuerdo a las características específicas del animal que representa.

Por ejemplo, se podría tener una clase hija Perro que extienda la clase Animal y que implemente los métodos "comer", "dormir" y "moverse" de acuerdo a las características específicas de los perros. Así, se puede crear una instancia de la clase Perro, pero no de la clase Animal, ya que esta última es abstracta y solo sirve como modelo para otras clases que sí pueden ser instanciadas.

EJEMPLO

public abstract class Figura {

private String color;

public Figura (String color){//Constructor this.color=color;

}

abstract double area(); //Método abstracto abstract double perimetro(); //Método abstracto

public String getColor() {//Método no abstracto return color;

}

}

**• Interface**

Una interfaz es un medio de comunicación entre dos sistemas o componentes que les permite intercambiar información y realizar acciones. En el contexto de la informática, una interfaz puede ser gráfica o de línea de comandos y permite a los usuarios interactuar con un programa o sistema operativo.

Algunos ejemplos de interfaces son:

Interfaz de usuario (UI): es una interfaz gráfica que permite a los usuarios interactuar con un programa o dispositivo a través de elementos visuales como botones, menús y ventanas. Un ejemplo de interfaz de usuario es la pantalla táctil de un smartphone.

Interfaz de programación de aplicaciones (API): es una interfaz que define cómo los programas pueden comunicarse entre sí. Por ejemplo, una API de redes sociales podría permitir a los desarrolladores acceder a la información del perfil de un usuario.

Interfaz de línea de comandos (CLI): es una interfaz que permite a los usuarios interactuar con un programa o sistema operativo a través de comandos de texto. Un ejemplo de interfaz de línea de comandos es la Terminal en MacOS o la ventana de comandos en Windows.

EJEMPLO

interface WaterBottleInterface {

String color = "Blue";

void fillUp();

}

public class InterfaceExample implements WaterBottleInterface

public static void main(String[] args) {

System.out.println(color)

}

2) **Paquetes java para: QR, Bar Code, Bluetooth, WhatsApp**

3) **Public, private, protect, final**

"public", "private", "protected" y "final" son cuatro de las palabras clave más importantes en la programación orientada a objetos en lenguajes de programación como Java, Python, C ++, entre otros.

"public" es un modificador de acceso que indica que un miembro (método, variable, etc.) es accesible desde cualquier parte del programa, incluyendo desde fuera de la clase donde se define.

Ejemplo:

public class Ejemplo {

public int edad = 30; // variable pública

public void imprimirNombre(String nombre) { // método público

System.out.println("Nombre: " + nombre);

}

}

"private" es un modificador de acceso que indica que un miembro es accesible solo dentro de la misma clase donde se define.

Ejemplo:

public class Ejemplo {

private int id = 123; // variable privada

private void imprimirId() { // método privado

System.out.println("ID: " + id);

}

}

"protected" es un modificador de acceso que permite el acceso al miembro dentro de la misma clase y también en las subclases (heredadas) de la clase donde se define.

Ejemplo:

public class Ejemplo {

protected String mensaje = "Hola"; // variable protegida

protected void imprimirMensaje() { // método protegido

System.out.println("Mensaje: " + mensaje);

}

}

"final" es una palabra clave que indica que una variable o un método no puede ser modificado después de su definición (para variables) o sobreescrito por una subclase (para métodos).

Ejemplo:

public class Ejemplo {

public final double PI = 3.1416; // variable final

public final void imprimirMensaje() { // método final

System.out.println("Este método no puede ser sobreescrito");

}

}

4) **Como saber el tipo de dato de una variable en java**

5) **Manejo de cadenas y RegEx Java**