

CUESTIÓN 1. POO en Java y colecciones

- Diseña un ejemplo de clases en donde puedas demostrar la herencia en Java. Explica si es posible realizar herencia múltiple en Java y por qué.

```
package hito1_3t_programacion;

//Creamos una clase base llamada Vehiculo
class Vehiculo {
    // Atributos de la clase Vehiculo
    String marca;
    String modelo;

    // Constructor de la clase Vehiculo
    Vehiculo(String marca, String modelo) {
        this.marca = marca;
        this.modelo = modelo;
    }

    // Método para imprimir los detalles del vehículo
    void imprimirDetalles() {
        System.out.println("Marca: " + marca);
        System.out.println("Modelo: " + modelo);
    }
}

//Creamos una subclase llamada Coche que hereda de la clase Vehiculo
class Coche extends Vehiculo {
    // Atributo específico de la clase Coche
    int puertas;

    // Constructor de la clase Coche
    Coche(String marca, String modelo, int puertas) {
        // Llamamos al constructor de la clase base usando super()
        super(marca, modelo);
        this.puertas = puertas;
    }

    // Método para imprimir los detalles del coche, incluyendo los heredados de Vehiculo
    void imprimirDetalles() {
        super.imprimirDetalles(); // Llamamos al método de la clase base para imprimir los detalles del vehículo
        System.out.println("Número de puertas: " + puertas);
    }
}
```

```

package hito1_3t_programacion;

//Clase principal para probar la herencia en Java
public class Main {
    public static void main(String[] args) {
        // Creamos un objeto de la clase Coche
        Coche miCoche = new Coche("Toyota", "Corolla", 4);
        // Llamamos al método para imprimir los detalles del coche
        miCoche.imprimirDetalles();
    }
}

```

▪ Siguiendo el ejercicio anterior, propón un ejemplo para diferenciar sobrecarga y sobrescritura. El ejemplo funcionará en consola.

```

package hito1_3t_programacion;

// Creamos una clase base llamada Vehiculo
class Vehiculo {
    // Atributos de la clase Vehiculo
    String marca;
    String modelo;

    // Constructor de la clase Vehiculo
    Vehiculo(String marca, String modelo) {
        this.marca = marca;
        this.modelo = modelo;
    }

    // Método para imprimir los detalles del vehículo
    void imprimirDetalles() {
        System.out.println("Marca: " + marca);
        System.out.println("Modelo: " + modelo);
    }
}

// Creamos una subclase llamada Coche que hereda de la clase Vehiculo
class Coche extends Vehiculo {
    // Atributo específico de la clase Coche
    int puertas;

    // Constructor de la clase Coche
    Coche(String marca, String modelo, int puertas) {
        // Llamamos al constructor de la clase base usando super()
        super(marca, modelo);
        this.puertas = puertas;
    }

    // Método para imprimir los detalles del coche, incluyendo los heredados de Vehiculo
    @Override
    void imprimirDetalles() {
        super.imprimirDetalles(); // Llamamos al método de la clase base para imprimir los detalles del vehículo
        System.out.println("Número de puertas: " + puertas);
    }
}

```

```

}

// Creamos una clase llamada Operaciones
class Operaciones {
    // Método para sumar dos números enteros
    int sumar(int a, int b) {
        return a + b;
    }

    // Método sobrecargado para sumar dos números decimales
    double sumar(double a, double b) {
        return a + b;
    }

    // Método para restar dos números enteros
    int restar(int a, int b) {
        return a - b;
    }

    // Método sobrescrito para restar dos números enteros
    int restar(int a, int b, int c) {
        return a - b - c;
    }
}

package hito1_3t_programacion;

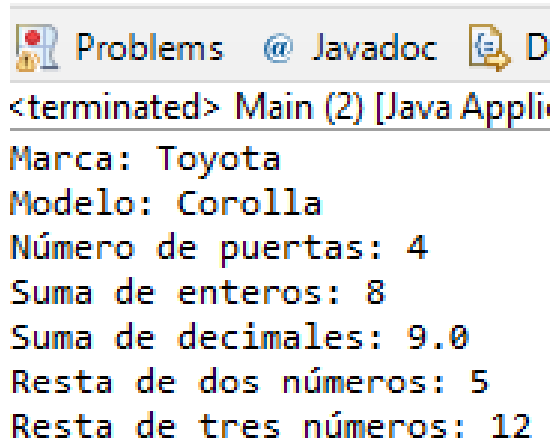
//Clase principal para probar la herencia en Java
public class Main {
    public static void main(String[] args) {
        // Creamos un objeto de la clase Coche
        Coche miCoche = new Coche("Toyota", "Corolla", 4);
        // Llamamos al método para imprimir los detalles del coche
        miCoche.imprimirDetalles();

        // Creamos un objeto de la clase Operaciones
        Operaciones operaciones = new Operaciones();

        // Ejemplo de sobrecarga: suma de enteros y suma de decimales
        int sumaEnteros = operaciones.sumar(5, 3); // Se utiliza el primer
        método sumar
        double sumaDecimales = operaciones.sumar(5.5, 3.5); // Se utiliza el
        segundo método sumar
        System.out.println("Suma de enteros: " + sumaEnteros);
        System.out.println("Suma de decimales: " + sumaDecimales);

        // Ejemplo de sobreescritura: resta de dos números y resta de tres
        números
        int restaDosNumeros = operaciones.restar(10, 5); // Se utiliza el primer
        método restar
        int restaTresNumeros = operaciones.restar(20, 5, 3); // Se utiliza el
        segundo método restar
        System.out.println("Resta de dos números: " + restaDosNumeros);
        System.out.println("Resta de tres números: " + restaTresNumeros);
    }
}

```



```
<terminated> Main (2) [Java Appli  
Marca: Toyota  
Modelo: Corolla  
Número de puertas: 4  
Suma de enteros: 8  
Suma de decimales: 9.0  
Resta de dos números: 5  
Resta de tres números: 12
```

Sobrecarga: Se define el método sumar dos veces en la clase Operaciones, una vez para sumar enteros y otra para sumar decimales. Ambos métodos tienen el mismo nombre pero diferentes parámetros. Cuando se llama al método sumar con diferentes tipos de parámetros, Java determina cuál versión ejecutar en función de los tipos de argumentos.

Sobreescritura: Se define el método restar dos veces en la clase Operaciones, una vez con dos parámetros y otra vez con tres parámetros. Esta es una técnica de herencia donde una subclase proporciona una implementación específica de un método que ya está definido en su superclase. Cuando se llama al método restar con diferentes números de argumentos, Java ejecuta la versión correspondiente según el número de parámetros.

▪ **En Java existen varias opciones para almacenar datos en colecciones. Explica con un ejemplo qué diferencia hay entre colecciones de tipo lista, pila, cola y vector.**

Una lista en Java es una estructura de datos fundamental que permite almacenar elementos de manera ordenada y con la posibilidad de duplicados. Se accede a los elementos de una lista utilizando su índice, lo que facilita la recuperación y manipulación de datos específicos. Por ejemplo, si tenemos una lista de nombres, podemos acceder al primer nombre utilizando el índice 0, al segundo utilizando el índice 1, y así sucesivamente. Esta capacidad de acceso por índice facilita la implementación de algoritmos y operaciones sobre los datos almacenados en la lista.

Las listas son ampliamente utilizadas en Java debido a su versatilidad y facilidad de uso. Pueden contener cualquier tipo de objeto, lo que las hace adecuadas para una amplia gama de aplicaciones. Además, las listas proporcionan métodos para agregar, eliminar, buscar y modificar elementos de manera eficiente, lo que las convierte en una herramienta poderosa para el desarrollo de software.

```
import java.util.ArrayList;
import java.util.List;

public class Main {
    public static void main(String[] args) {
        List<String> lista = new ArrayList<>();
        lista.add("Manzana");
        lista.add("Banana");
        lista.add("Cereza");
        System.out.println("Elementos de la lista: " + lista);
    }
}
```

Una pila se implementa comúnmente utilizando la clase Stack o mediante la interfaz Deque con la implementación ArrayDeque. Esta estructura de datos sigue el principio LIFO (Last In, First Out), lo que implica que el último elemento agregado a la pila será el primero en ser eliminado. Por ejemplo, al apilar elementos en una pila, el último elemento agregado será el primero en ser retirado cuando se desapila la pila.

La capacidad LIFO de una pila la hace útil para muchas aplicaciones, como la inversión de elementos, la reversión de cadenas y la evaluación de expresiones matemáticas en notación posfija. Además, las pilas son eficientes en términos de tiempo para operaciones como inserción y eliminación de elementos en la parte superior de la pila, lo que las convierte en una opción popular para resolver una variedad de problemas de programación.

```
import java.util.Stack;

public class Main {
    public static void main(String[] args) {
        Stack<String> pila = new Stack<>();
        pila.push("Manzana");
        pila.push("Banana");
        pila.push("Cereza");
        System.out.println("Elemento en la cima de la pila: " +
pila.peek());
    }
}
```

una cola se puede implementar utilizando la interfaz Queue y sus diversas implementaciones, como LinkedList o ArrayDeque. La estructura de datos de cola sigue el principio FIFO (First In, First Out), lo que garantiza que el primer elemento agregado a la cola será el primero en ser eliminado cuando se realiza una operación de extracción. Este comportamiento la hace útil en situaciones donde se necesita procesar elementos en el orden en que llegaron, como en sistemas de colas de espera o procesamiento de tareas en lotes. Además, las colas en Java ofrecen métodos

eficientes para agregar, eliminar y consultar elementos tanto al principio como al final de la cola, lo que las convierte en una opción versátil para una amplia gama de aplicaciones de programación.

```
import java.util.LinkedList;
import java.util.Queue;

public class Main {
    public static void main(String[] args) {
        Queue<String> cola = new LinkedList<>();
        cola.add("Manzana");
        cola.add("Banana");
        cola.add("Cereza");
        System.out.println("Elemento al frente de la cola: " +
cola.peek());
    }
}
```

un vector es una estructura de datos dinámica que permite almacenar objetos y se redimensiona automáticamente según se agregan o eliminan elementos. A diferencia de los arrays tradicionales, los vectores en Java pueden cambiar de tamaño de manera dinámica, lo que los hace más flexibles para gestionar colecciones de elementos de longitud variable. Al utilizar la clase Vector proporcionada en Java, los desarrolladores pueden realizar operaciones como agregar elementos al final del vector, insertar elementos en cualquier posición, eliminar elementos por índice o valor, y obtener elementos por índice. Esta capacidad de redimensionamiento automático hace que los vectores sean útiles en situaciones donde la cantidad de elementos puede variar durante la ejecución del programa, permitiendo una gestión eficiente de la memoria y una mayor flexibilidad en el almacenamiento de datos.

```
import java.util.Vector;

public class Main {
    public static void main(String[] args) {
        Vector<String> vector = new Vector<>();
        vector.add("Manzana");
        vector.add("Banana");
        vector.add("Cereza");
        System.out.println("Elementos del vector: " + vector);
    }
}
```