

**SERIE I:**

Se presentan conceptos generales para el curso de estructura de datos, deberá dar una explicación clara y concisa de cada uno de los temas que se plantean a continuación, respalde su explicación con un ejemplo práctico.

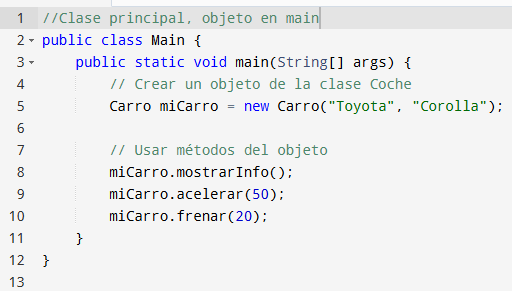
1. ¿Qué es POO? Explique con ejemplos en Java

La Programación Orientada a Objetos (POO) es un paradigma de programación que se basa en la creación de objetos que combinan datos y comportamientos, y el código se organiza en torno a esos objetos. Los objetos representan entidades del mundo real con atributos (propiedades) y comportamientos (métodos). La POO se basa en los principios de encapsulación, herencia y polimorfismo. En Java, la POO se implementa a través de clases y objetos.

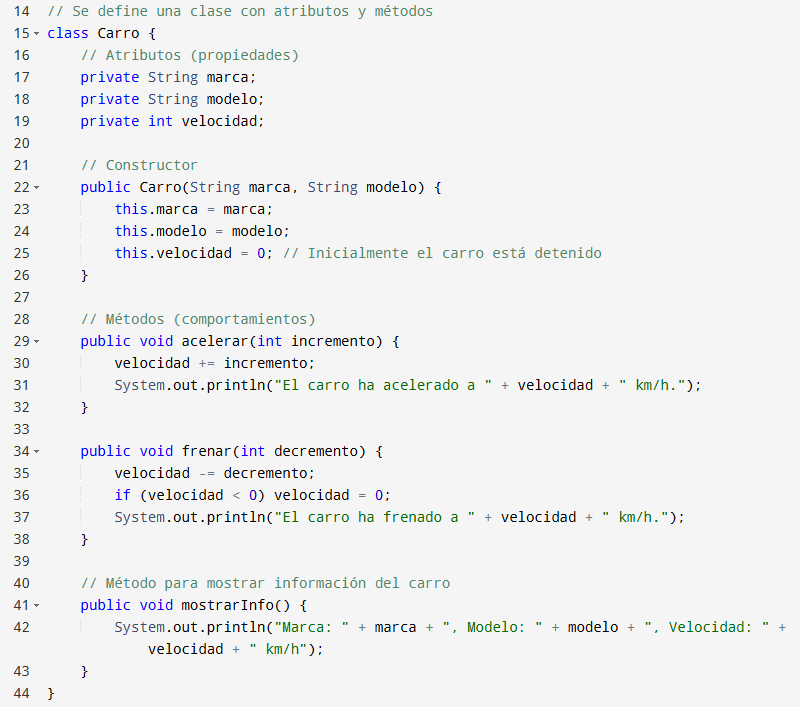
Principales conceptos de POO

* 1. Clases: Son plantillas o moldes que definen las características y comportamientos de un objeto.
  2. Objetos: Son instancias de una clase.
  3. Encapsulamiento: Protege los datos dentro de una clase y controla su acceso mediante modificadores (private, public).
  4. Herencia: Permite que una clase herede atributos y métodos de otra.
  5. Polimorfismo: Permite que un mismo método tenga diferentes implementaciones en distintas clases.
  6. Abstracción: Permite definir clases base sin implementar completamente sus métodos.

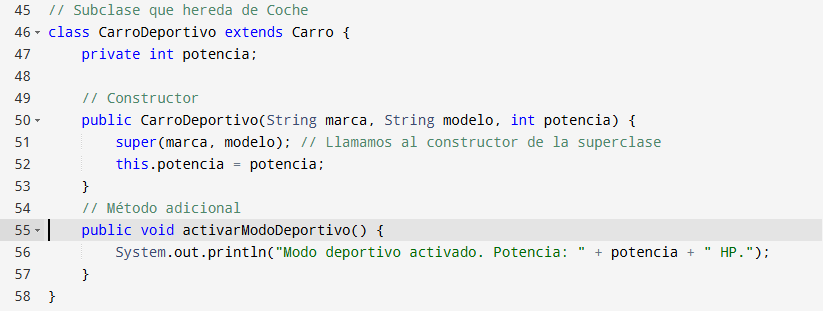
Ejemplo de la creación de un objeto en la clase principal



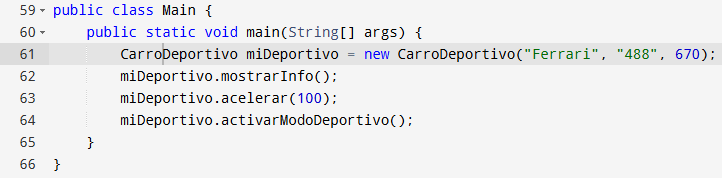
Ejemplo en Java de la Clase Carro



Ejemplo de una herencia



Ejemplo de creación de un objeto



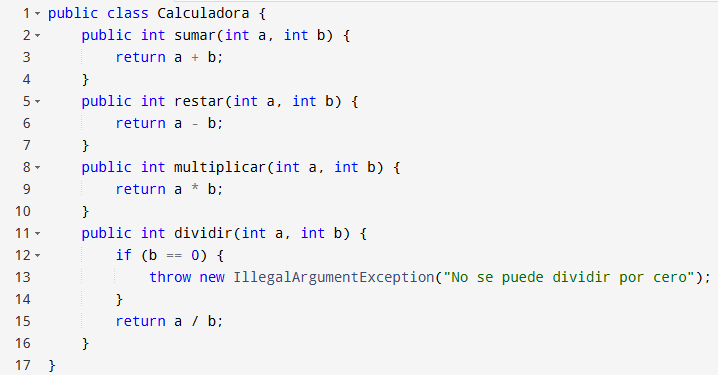
1. ¿Que son pruebas unitarias?

Las pruebas unitarias son un tipo de prueba de software que verifica el correcto funcionamiento de una unidad individual de código, generalmente una función o método de una clase. Estas pruebas ayudan a detectar errores en las primeras etapas del desarrollo, asegurando que cada parte del código funcione como se espera.

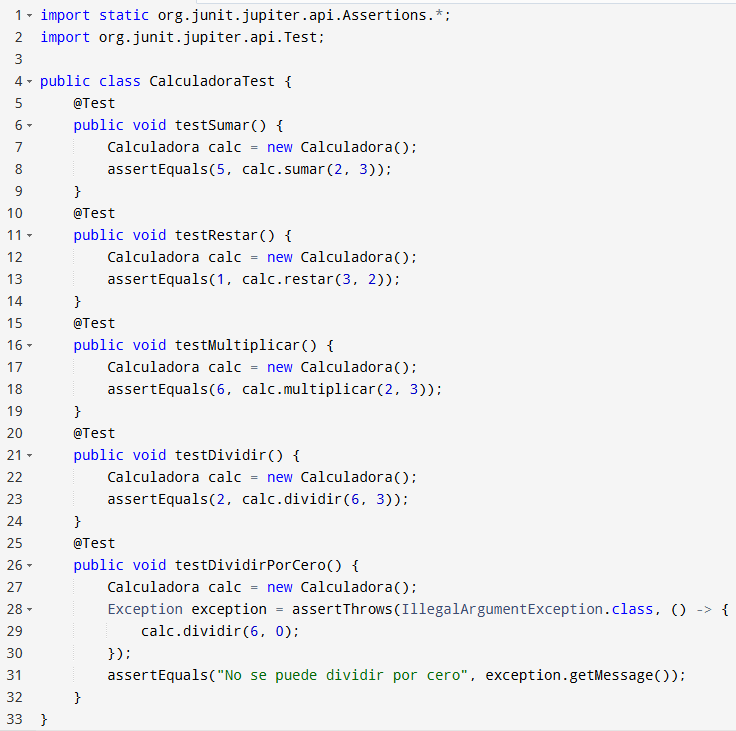
Características de las pruebas unitarias

* 1. Automatizadas: Se ejecutan con herramientas especializadas sin intervención manual.
  2. Rápidas: Son pruebas pequeñas y se ejecutan en pocos milisegundos.
  3. Aisladas: Prueban unidades independientes sin depender de otras partes del sistema.
  4. Repetibles: Se pueden ejecutar varias veces con los mismos resultados.
  5. Fáciles de mantener: Facilitan la detección y corrección de errores en el código.

Ejemplo de la clase Calculadora a la que se le realizará prueba unitaria, en java con JUnit



Creación de una clase de prueba en JUnit con métodos para verificar el comportamiento de la clase Calculadora



Luego se ejecuta el comando

mvn test

ya sea en Eclipse, IntelliJ IDEA o Maven.

1. ¿Qué es un lenguaje de programación fuertemente tipado y que características puede mencionar versus un lenguaje de programación que no tenga tipado fuerte?

|  |  |
| --- | --- |
| **Lenguaje fuertemente tipado** | **Lenguaje débilmente tipado** |
| Más seguro y estructurado | Más flexibilidad y rapidez para los prototipos |
| los tipos de datos están estrictamente definidos y no pueden convertirse automáticamente a otro tipo sin una conversión explícita. | permite conversiones de tipo de manera implícita, lo que puede llevar a errores difíciles de detectar. |
| No permite conversiones implícitas peligrosas (por ejemplo, no convierte un int a String automáticamente). | Permite conversiones implícitas entre tipos sin advertencias del compilador. |
| El compilador detecta errores de tipo en tiempo de compilación, reduciendo errores en ejecución. | Puede generar errores inesperados en tiempo de ejecución. |
| Cada variable tiene un tipo fijo que no puede cambiar durante la ejecución. | Mayor flexibilidad, pero también mayor riesgo de errores lógicos. |
| Ejemplo en Java (fuertemente tipado)    Conversión explícita en Java | Ejemplo en Java (débimente tipado) |
| Ejemplos de lenguajes fuertemente tipados:  Java, Python, C#, Kotlin, Swift. | Ejemplos de lenguajes débilmente tipados:  JavaScript, PHP, Perl, Lua |

1. ¿Qué son convenciones de código?

Las convenciones de código son un conjunto de reglas y buenas prácticas que definen la forma en que debe escribirse el código en un lenguaje de programación. Estas convenciones ayudan a mejorar la legibilidad, mantenimiento y consistencia del código, especialmente en proyectos colaborativos.

Ejemplos de aplicación en nombres de variables y métodos

* 1. Usar camelCase:
     1. Los nombres deben ser descriptivos



* + 1. La primera palabra en minúsculas y las siguientes con la primera letra en mayúscula.



Ejemplos de clases e Interfaces

* 1. Usar PascalCase

(Cada palabra inicia con mayúscula)



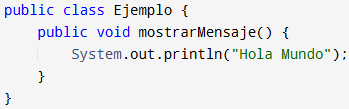
Ejemplo de uso de Constantes

* 1. Las constantes deben estar en mayúsculas con guiones bajos (\_).



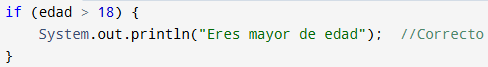
Ejemplo de identación y espaciado

* 1. Usar 4 espacios por nivel de identación (NO tabulaciones).



Ejemplo de uso de llaves en Condicionales y Bucles

* 1. Siempre usar llaves {} incluso si la estructura tiene una sola línea
     1. Correcto

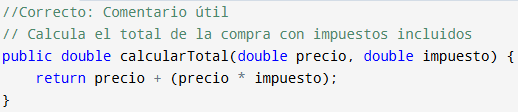


* + 1. Incorrecto

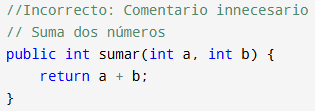


Ejemplo de comentarios en el Código

* 1. Usar comentarios para explicar bloques de código complejos.



* 1. Evitar comentarios innecesarios.

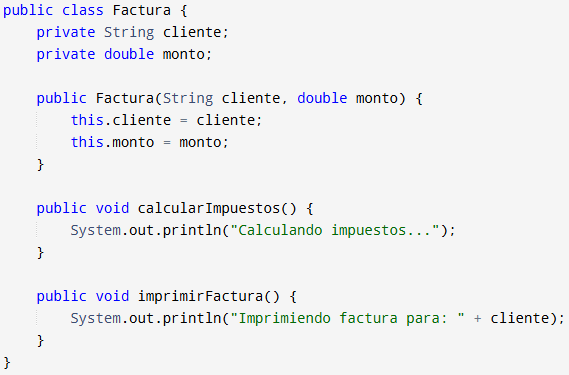


1. Según los principios SOLID, explique con un ejemplo en Java la forma correcta e incorrecta de aplicar el principio Single Responsability

El principio de responsabilidad única (SRP) establece que una clase debe tener una única razón para cambiar, es decir, debe encargarse de una sola responsabilidad dentro del sistema.

* 1. Ejemplo Incorrecto (Violación de SRP)

La clase Factura que maneja tanto la lógica de negocio como la impresión del recibo. Esto es incorrecto porque una clase debe centrarse en una sola responsabilidad.



Los errores son que la clase Factura tiene dos responsabilidades:

* + 1. Lógica de negocio (calcular impuestos).
    2. Impresión de la factura (formateo e impresión).

Si se necesita modificar el cálculo de impuestos o cambiar la forma de impresión, la clase se verá afectada, violando SRP.

* 1. Ejemplo Correcto (Aplicando SRP)

Separamos las responsabilidades en dos clases:

* + 1. Factura: Solo maneja la lógica de negocio.
    2. ImpresoraFactura: Se encarga de imprimir la factura.



Está bien porque …

1. Cada clase tiene una única responsabilidad:
   * + 1. Factura se encarga solo de los datos y la lógica.
       2. ImpresoraFactura maneja únicamente la impresión.
2. Mayor flexibilidad y mantenibilidad:
   * + 1. Si se cambia la forma de impresión, no afectará la lógica de facturación.
       2. Si se cambian los cálculos de impuestos, la clase de impresión no se verá afectada.

**SERIE II:**

Deberá resolver los siguientes desafíos de código usando arrays, dejar documentación necesaria en el código fuente, usar convenciones de código recomendadas para codificación en java, aplicar el principio SOLID Single Responsability subir el código fuente a un repositorio en github o bitbucket.

1. Se Representa el juego de escaleras y serpientes en Java usando solo arrays en el siguiente enlace:

<http://reglasdejuegosimples.blogspot.com/2013/05/serpientes-y-escaleras.html> el tablero (lineal) tiene 64 elementos donde en ciertas posiciones, iremos a una posición u otra. Te dejo las posiciones de ejemplo del enlace anterior como ejemplo.

int[] origenCasillasEspeciales = {7, 11, 31, 30, 40, 43, 50, 59};

int[] destinoCasillasEspeciales = {38, 37, 46, 2, 21, 60, 5, 42};

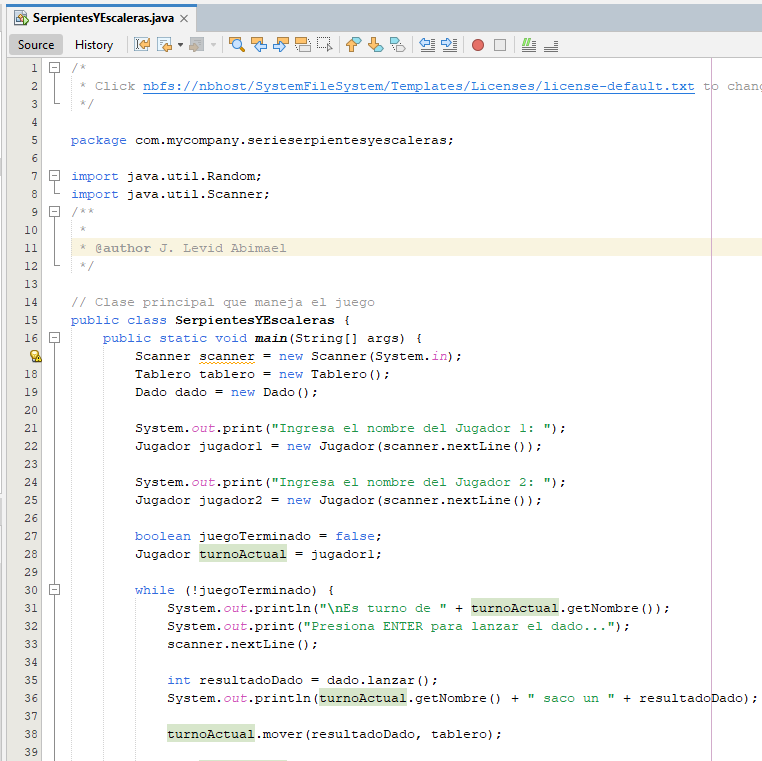
Los jugadores tiran el dado de 6 caras, si sacan un 6, vuelven a tirar, pero si sacan 3 veces consecutivas un 6, el jugador volverá a la posición 1 y se quedará bloqueado hasta que saque de nuevo un 6. El objetivo del juego es llegar a la última posición del tablero, en el caso de que se pase, tendrá que retroceder las posiciones restantes, como en los clásicos juegos de mesa.

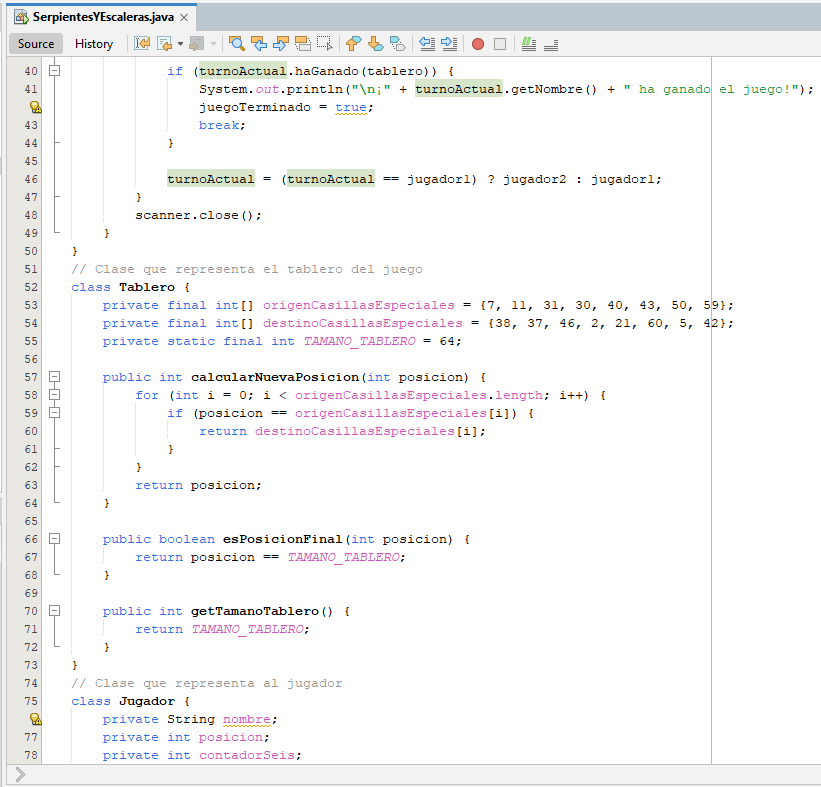
Respuesta: 🎲🐍⬆️

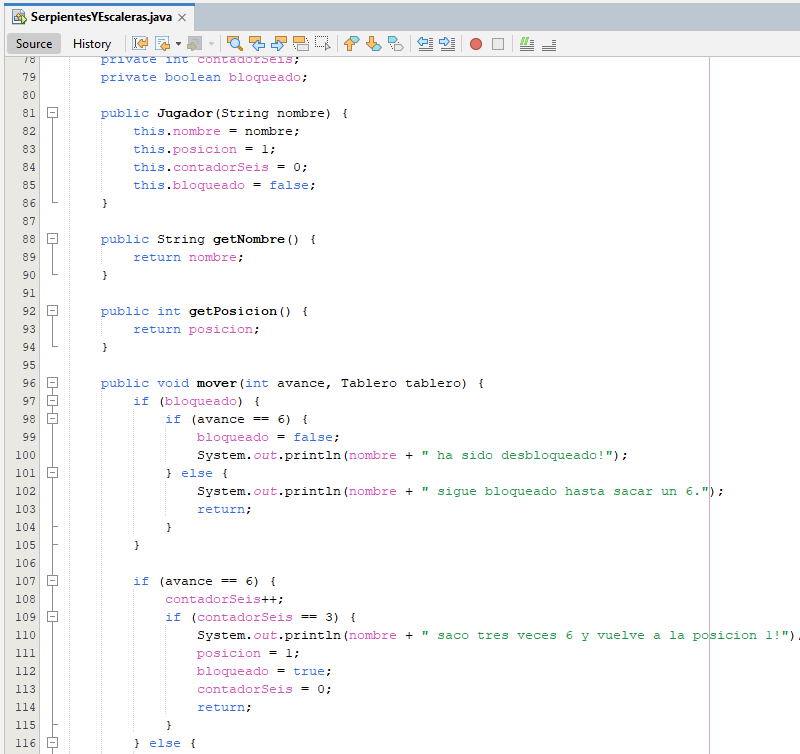
Para resolverlo correctamente en Java, se estructurará el código siguiendo el principio de responsabilidad única (SRP):

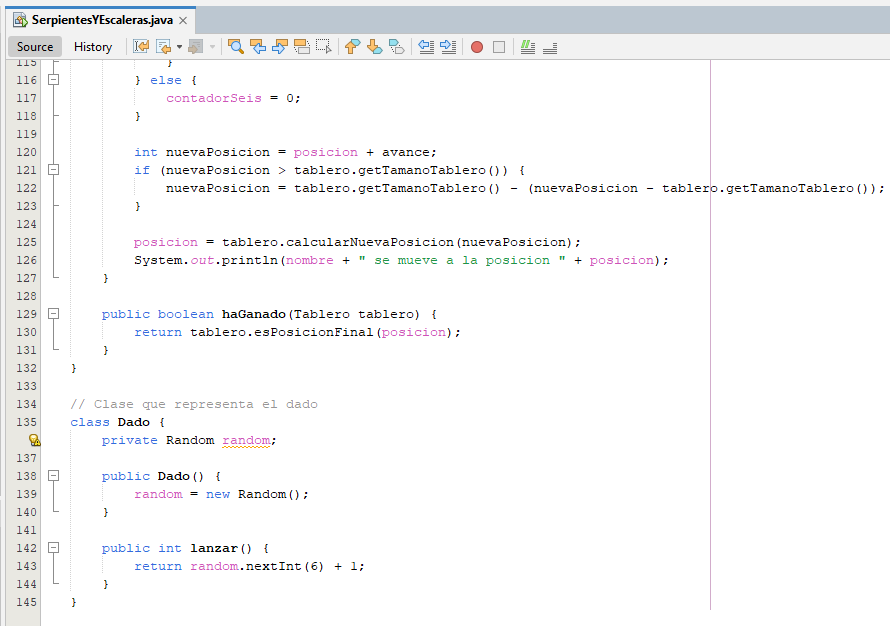
Clases:

* 1. Clase Tablero: Maneja el tablero y las casillas especiales (serpientes y escaleras).
  2. Clase Jugador: Almacena la información de cada jugador (posición actual y contador de seis consecutivos).
  3. Clase Dado: Se encarga de simular la tirada del dado.
  4. Clase Juego: Controla la lógica del juego y la interacción entre jugadores.









Explicación del código:

1. Tablero: Administra las posiciones del juego y maneja las casillas especiales.
2. Jugador: Almacena la posición del jugador y gestiona la lógica de bloqueos y movimiento.
3. Dado: Simula el lanzamiento del dado.
4. SerpientesYEscaleras: Controla la ejecución del juego, maneja turnos y verifica el ganador.

Este diseño modular sigue SRP (Single Responsibility Principle) y permite fácil mantenimiento.

1. Dada una matriz de números enteros y una posición específica dentro de ella, se requiere encontrar las posiciones adyacentes a esta posición en la matriz. Las posiciones adyacentes se definen como aquellas que comparten un lado con la posición dada, es decir, arriba, abajo, izquierda o derecha.

Por ejemplo, si tenemos una matriz de tamaño 3x3:

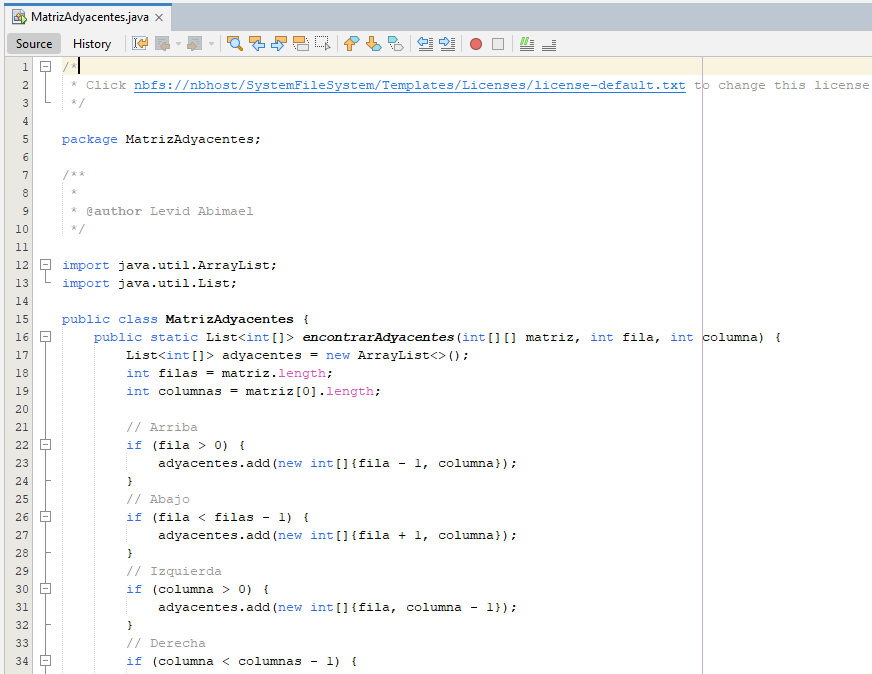
1 2 3

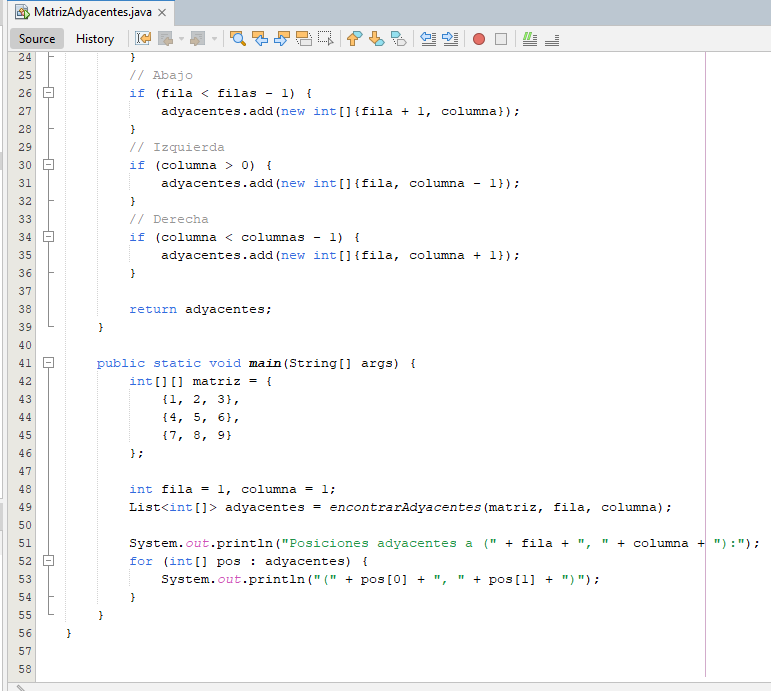
4 5 6

7 8 9

y queremos encontrar las posiciones adyacentes a la posición (1,1) (fila 1, columna 1), las posiciones adyacentes serían (0,1), (2,1), (1,0) y (1,2).

La tarea consiste en implementar un algoritmo que, dados una matriz y una posición dentro de ella, devuelva las posiciones adyacentes a esa posición.





1. Dada la head de una lista enlazada, devuelva la lista después de ordenarla en orden ascendente.

Prueba 1:

Entrada: cabeza = [4,2,1,3]

Salida: [1,2,3,4]

Prueba 2:

Entrada: cabeza = [-1,5,3,4,0]

Salida: [-1,0,3,4,5]

Prueba 3:

Entrada: cabeza = []

Salida: []

class ListNode {

int val;

//??

ListNode(int val) { this.val = val; this.next = null; }

}

public class OrdenarListaEnlazada {

public static ListNode sortList(ListNode head) {

if (head == null || head.next == null) {

return head;

}

// Dividir la lista en dos mitades

ListNode mid = getMiddle(head);

ListNode nextToMid = mid.next;

mid.next = null;

// Ordenar ambas mitades

ListNode left = sortList(head);

ListNode right = sortList(nextToMid);

// Mezclar las dos mitades ordenadas

return //??

}

private static ListNode getMiddle(ListNode head) {

ListNode slow = head, fast = head;

while (fast.next != null && fast.next.next != null) {

slow = slow.next;

fast = fast.next.next;

}

//??

}

private static //?? merge(ListNode l1, ListNode l2) {

ListNode dummy = new ListNode(0);

ListNode current = dummy;

while (l1 != null && l2 != null) {

if (l1.val < l2.val) {

// ??

// ??

} else {

current.next = l2;

l2 = l2.next;

}

current = current.next;

}

if (l1 != null) current.next = l1;

if (// ?? ) current.next = l2;

return dummy.next;

}

public static void printList(ListNode head) {

ListNode current = head;

while (current != null) {

System.out.print(current.val + " -> ");

current = current.next;

}

System.out.println("null");

}

public static void main(String[] args) {

ListNode head = new ListNode(4);

head.next = new ListNode(2);

head.next.next = new ListNode(1);

head.next.next.next = new ListNode(3);

System.out.println("Lista original:");

printList(head);

head = // ??

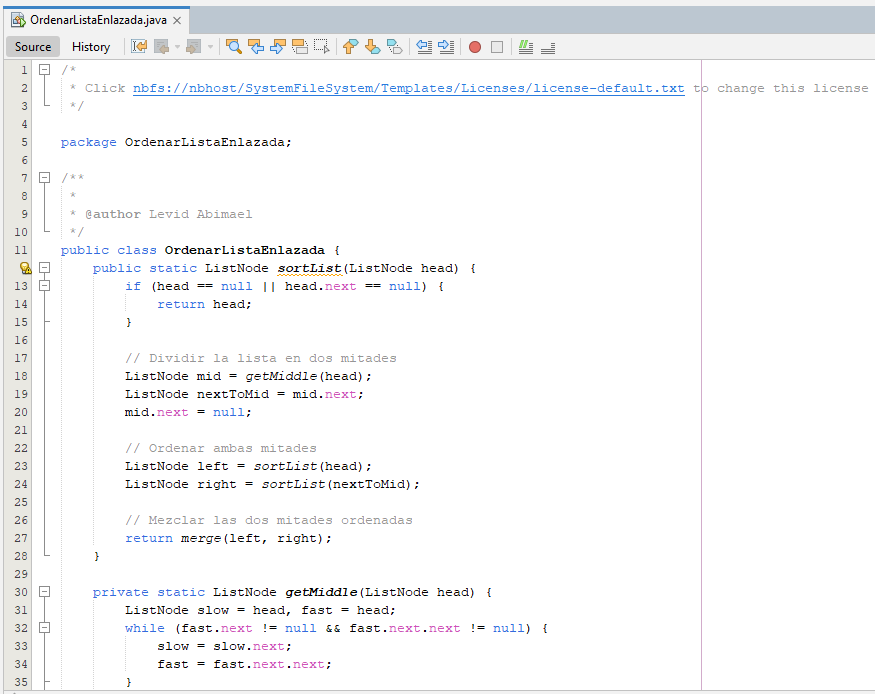
System.out.println("Lista ordenada:");

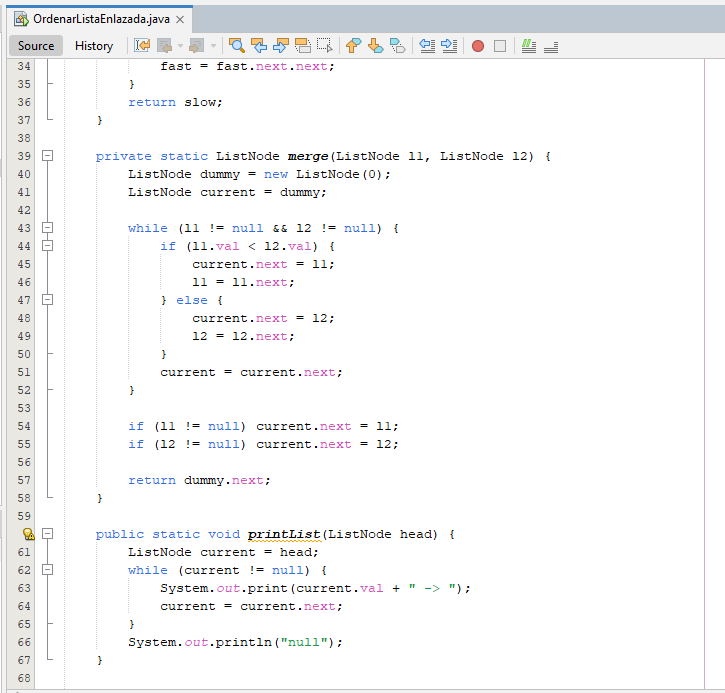
printList(head);

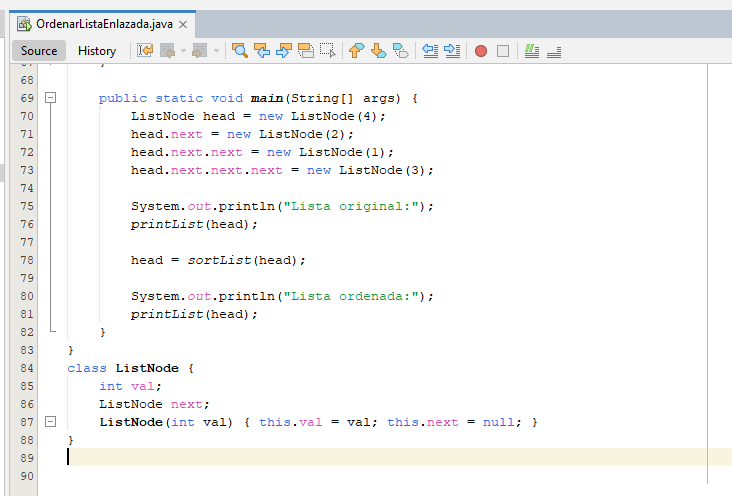
}

}

Código en IDE







“Primero resuelve el problema, después escribe el código”