La estructura general de un programa realizado en un lenguaje orientado a objetos como **Java** sigue ciertos principios clave. A continuación te presento la estructura típica de un programa en Java basado en la orientación a objetos:

1. Paquete (Opcional)

Si el programa pertenece a un paquete, la declaración del paquete debe ser lo primero en el archivo.

```
package nombre_del_paquete;
```

2. Importaciones

Después del paquete (si lo hay), se incluyen las importaciones de las clases necesarias de otros paquetes o librerías.

```
import java.util.Scanner;
```

3. Definición de la Clase

Todo el código en Java debe estar contenido dentro de una clase. Es común que la clase tenga el mismo nombre que el archivo que la contiene.

```
public class MiPrograma {
```

4. Atributos de la Clase (Opcional)

Dentro de la clase, primero se definen los atributos (variables) si es necesario. Estos atributos pueden ser estáticos o de instancia, y se declaran según el nivel de visibilidad (public, private, protected).

```
private int numero;
```

5. Método Constructor (Opcional)

Si la clase necesita un constructor para inicializar sus atributos u objetos, este se define a continuación. El constructor no tiene valor de retorno y comparte el nombre de la clase.

```
public MiPrograma(int numero) {
    this.numero = numero;
}
```

6. Métodos de la Clase

Los métodos de la clase definen el comportamiento de los objetos. El método más importante y requerido es el método `main`, que es el punto de entrada de cualquier aplicación Java. Otros métodos se definen según sea necesario.

```
public void mostrarNumero() {
    System.out.println("El número es: " + this.numero);
}

public static void main(String[] args) {
    MiPrograma programa = new MiPrograma(10);
    programa.mostrarNumero();
}
```

7. Comentarios (Opcional pero recomendado)

Es recomendable incluir comentarios para mejorar la legibilidad y el mantenimiento del código.

```
// Este es un comentario de una sola línea

/* Este es un comentario
   de varias líneas */
```

Ejemplo completo:

```java
package com.ejemplo;        //Si el programa pertenece a un paquete

import java.util.Scanner;  //import de paquetes y librerias si necesario

public class MiPrograma {   //Definiendo la clase

    private int numero;     //Definir atributos si necesario (variables)

    // Constructor de la clase
    public MiPrograma(int numero) {
        this.numero = numero;
    }

    // Método para mostrar el número
    public void mostrarNumero() {
        System.out.println("El número es: " + this.numero);
    }

    // Método principal, punto de entrada del programa
    public static void main(String[] args) {
        Scanner entrada = new Scanner(System.in);
        System.out.println("Introduce un número: ");
        int num = entrada.nextInt();

        MiPrograma programa = new MiPrograma(num);
        programa.mostrarNumero();
    }
}
```

Esta es una estructura básica que puedes encontrar en cualquier programa orientado a objetos en Java. Cada componente puede expandirse dependiendo de la complejidad del programa, pero esta es la base esencial.

Recuerda seguir las convenciones de estilo mencionadas en los documentos, como usar nombres descriptivos para las clases y variables.

Recomendaciones:

En un programa orientado a objetos, como los que se desarrollan en Java, cada una de las partes de la estructura cumple un rol específico y tienen relaciones que son fundamentales para organizar el código de manera lógica, mantenerlo entendible y facilitar su reutilización. Aquí te dejo recomendaciones clave sobre cómo se relacionan estas partes y por qué son importantes:

1. Paquete y Estructura Modular

- **Relación con las clases y el proyecto completo:** Los **paquetes** te permiten agrupar las clases que tienen funciones relacionadas entre sí. Piensa en los paquetes como carpetas que organizan el código en módulos. Si tienes un proyecto grande, organizar las clases en paquetes hace que el código sea más estructurado y fácil de mantener. Por ejemplo, si tienes clases de "Usuarios" y "Productos", podrías tener un paquete `usuarios` y otro `productos` para dividir las funcionalidades.
- **Por qué es importante:** Esta estructura modular ayuda a evitar conflictos de nombres entre clases y permite la reutilización de código en proyectos más grandes, porque te da un espacio de nombres exclusivo para tus clases.

Relación: Las **clases** dependen de los **paquetes** para ubicarse en un contexto organizativo, lo cual es útil cuando trabajas en proyectos grandes o colaborativos.

---

2. Importaciones

- **Relación con los objetos y las clases:** Las **importaciones** conectan tu programa con bibliotecas externas o con clases definidas en otros paquetes. Por ejemplo, si necesitas trabajar con una clase de utilidad como `Scanner` para leer datos de entrada, la importas desde `java.util.Scanner`.
- **Por qué es importante:** Permite que tu clase utilice funcionalidad ya existente sin tener que escribir todo el código desde cero. Java tiene un vasto ecosistema de bibliotecas que puedes reutilizar para resolver problemas comunes, como trabajar con archivos, manipular fechas, hacer cálculos, etc.

Relación: Los **métodos** que uses dentro de tu clase pueden necesitar instancias de otras clases (por ejemplo, un objeto `Scanner`). Estas clases deben ser importadas para poder ser usadas.

3. Clase Principal

- **Relación con los métodos y atributos:** La **clase principal** (como `MiPrograma` en el ejemplo) encapsula los datos (atributos) y comportamientos (métodos) relacionados con el propósito del programa. Dentro de una clase, los **atributos** definen el estado de los objetos y los **métodos** definen las acciones que esos objetos pueden realizar.
- **Por qué es importante:** El concepto de **encapsulación** es clave en la programación orientada a objetos (POO). Mantienes los datos y los métodos que operan sobre esos datos dentro de una clase, lo que mejora la organización y evita interferencias accidentales con otras partes del programa.

Relación: Los **atributos** y **métodos** están fuertemente relacionados en una clase. Los métodos, como `mostrarNumero()`, utilizan y manipulan los atributos (`numero`). Además, las clases se pueden relacionar con otras clases a través de composición o herencia, lo que refuerza la reutilización de código.

---

4. Método `main`

- **Relación con la ejecución del programa:** El método `main` es el **punto de entrada** del programa. Aquí es donde Java empieza a ejecutar el código. En este método es donde generalmente se crean instancias de la clase (objetos) y se llaman los métodos de esos objetos.
- **Por qué es importante:** El método `main` marca la ejecución de las tareas y define cómo el flujo de ejecución del programa se desarrollará. Es crucial porque **conecta todas las demás partes**: invoca los métodos de la clase, manipula atributos, e interactúa con el usuario o el sistema.

Relación: El **método** `main` coordina la interacción entre los objetos que creas y los métodos que llamas. Por ejemplo, en el método `main` se puede crear un objeto de la clase `MiPrograma` y luego utilizar ese objeto para invocar otros métodos como `mostrarNumero()`.

---

5. Atributos y Métodos

- **Relación interna de la clase:** Los **atributos** son las características o propiedades de una clase y los **métodos** son las acciones que pueden realizar sobre esos atributos. El concepto de **encapsulación** es vital aquí: los atributos normalmente se declaran como `private` para proteger el estado interno del objeto, y se acceden a ellos mediante métodos públicos, llamados **getters** y **setters**, o a través de otros métodos de la clase.
- **Por qué es importante:** Mantener los atributos privados garantiza que el estado del objeto solo pueda ser modificado de maneras controladas. Los métodos son los responsables de cambiar este estado de una manera coherente y segura.

Relación: Los **métodos** actúan sobre los **atributos**. Los métodos de la clase, como `mostrarNumero()`, acceden a los atributos (`numero`) para mostrar o manipular sus valores.

---

6. Constructores

- **Relación con la inicialización de objetos:** El **constructor** de una clase es el método especial que se ejecuta cuando se crea una nueva instancia de la clase. Su propósito es inicializar los atributos del objeto.
- **Por qué es importante:** Permite asegurarse de que los objetos de la clase siempre se crean en un estado válido. En el ejemplo, el constructor de `MiPrograma` recibe un número como parámetro y lo asigna al atributo `numero`.

Relación: El **constructor** trabaja junto con los **atributos** para asegurar que cada vez que se cree un nuevo objeto, se establezca un estado inicial para ese objeto.

---

7. Comentarios

- **Relación con la legibilidad y mantenimiento del código:** Los **comentarios** son importantes para documentar el propósito de diferentes partes del código. No tienen un impacto directo en la ejecución del programa, pero ayudan a futuros desarrolladores (o a ti mismo) a entender qué hace cada parte del código.

- **Por qué es importante:** El código bien documentado es más fácil de mantener, entender y extender. Es especialmente crucial en proyectos grandes o cuando varias personas colaboran en un mismo código.

Relación: Los **comentarios** explican las relaciones entre las diferentes partes del código y por qué se ha tomado una decisión particular en el diseño del programa.

Relación Global: **Encapsulación y Modularidad**

El principio de **encapsulación** en la POO agrupa datos y comportamientos en clases, protegiendo el estado interno de los objetos y exponiendo solo los métodos necesarios. Los **paquetes** permiten estructurar las clases en módulos, facilitando la organización de grandes proyectos. El **método** `main` conecta todos los elementos al ser el punto de entrada que orquesta la creación de objetos y la ejecución de los métodos.

Estas relaciones refuerzan la **modularidad**, un concepto clave para escribir código limpio, reutilizable y mantenible.

Verificacion de versiones por terminal

Para verificar qué versiones del JDK (Java Development Kit) y del JRE (Java Runtime Environment) tienes instaladas en **Windows** y **Debian** a través de la línea de comandos, puedes seguir los pasos a continuación:

1. En Windows

Comprobar la versión del **JDK** y **JRE**:

- **Abrir el Símbolo del Sistema (CMD):**
  - Pulsa `Windows + R`, escribe `cmd`, y presiona **Enter**.

Comando para verificar la versión de **Java**:

- Ejecuta el siguiente comando para comprobar la versión del JRE y del JDK:

  ```
  java -version
  ```

  Este comando te mostrará la versión del **JRE** (Java Runtime Environment).

  Para comprobar específicamente la versión del **JDK**, usa:

  ```
  javac -version
  ```

  Esto te mostrará la versión del compilador **Javac**, que es parte del **JDK**.

Ejemplo de salida:

```
java version "17.0.2" 2022-01-18 LTS
Java(TM) SE Runtime Environment (build 17.0.2+8-LTS-86)
Java HotSpot(TM) 64-Bit Server VM (build 17.0.2+8-LTS-86, mixed mode, sharing)
```

Para `javac`:

```
javac 17.0.2
```

2. En Debian (o cualquier distribución Linux basada en Debian)

Comprobar la versión del **JDK** y **JRE**:

- **Abrir una Terminal:**
  - Puedes abrir la terminal desde el menú o con el atajo `Ctrl + Alt + T`.

Comando para verificar la versión de **Java**:

- Para verificar la versión del **JRE**, ejecuta:

  ```
  java -version
  ```

- Para verificar la versión del **JDK** (específicamente el compilador `javac`), ejecuta:

  ```
  javac -version
  ```

Ejemplo de salida:

```
openjdk version "11.0.11" 2021-04-20
OpenJDK Runtime Environment (build 11.0.11+9-Debian-1deb10u2)
OpenJDK 64-Bit Server VM (build 11.0.11+9-Debian-1deb10u2, mixed mode, sharing)
```

Para `javac`:

```
javac 11.0.11
```

Notas adicionales:

- Si obtienes un error de comando no encontrado (`java: command not found` o `javac: command not found`), esto indica que Java no está instalado o que no está correctamente añadido a la variable de entorno `PATH`.
- **En Debian**, si necesitas cambiar entre diferentes versiones instaladas de Java, puedes usar el comando:

  ```
  sudo update-alternatives --config java
  ```

  Y para cambiar entre diferentes versiones de `javac`:

  ```
  sudo update-alternatives --config javac
  ```

Con estos comandos, puedes gestionar las versiones instaladas y seleccionar la predeterminada.