Informe Conexion y Consultas a Bases de Datos Java

David Felipe Morales

María Fernanda Ibáñez Benavides

Dilan Alexander Robayo

Juan Sebastian Florez

Julio Roberto Galvis

Servicio Nacional de Aprendizaje

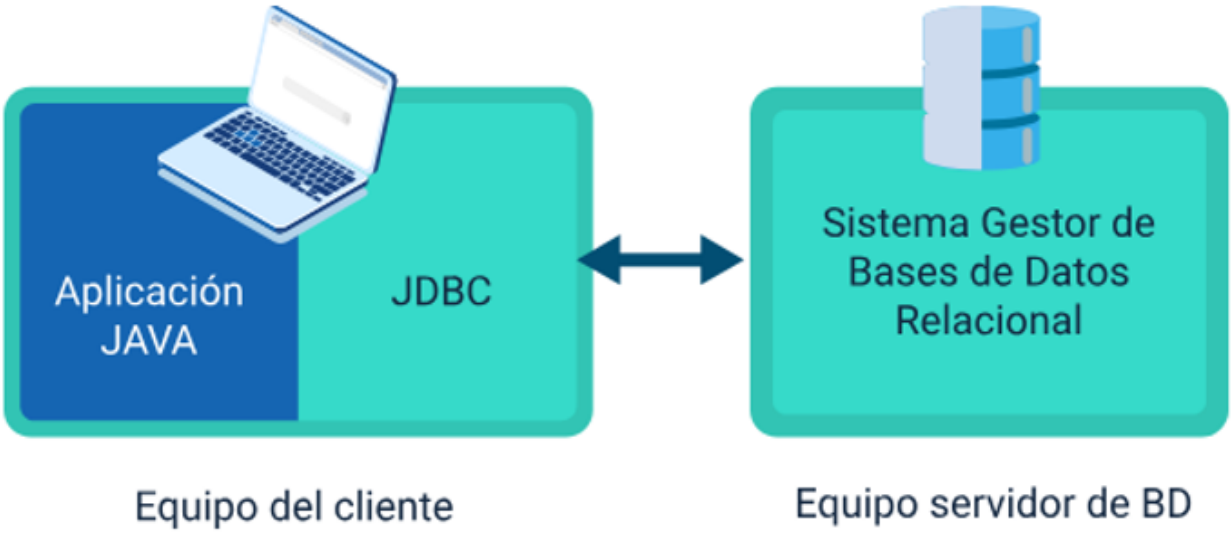
Análisis y desarrollo de Software

2024

Bogotá D.

Construcción de aplicaciones con Java

JDBC hace referencia a un *Driver* que encapsula un conjunto de clases, creadas para procesar datos, establecer conexiones y enviar sentencias a SQL (base de datos)



En estas bases de datos es posible manejar manualmente las conexiones y las desconexiones en las bases de datos, es importante conocer los modelos de bases de datos que se van a manipular, así para manejar una sintaxis correcta y poder generar las consultas de manera adecuada, la siguiente imagen indica cuales son los JDBC de las distintas bases de datos

Base de Datos	Página oficial para descarga de JDBC
Mysql	MySQL Community
SQL Server	Microsoft JDBC Driver for SQL Server
Oracle	Oracle Database
PostgreSQL	PostgreSQL
DB2	IBM DB2

1.1 **Clases e interfaces Java**

Algunas clases que se presentan son:

DriverMananger: es una clase estática que se puede utilizar mediante el getConnection, este se encarga de poder mover la información en la base de datos hacia el destino.

ResultSet: Un ResultSet es un tipo de dato que permite manejar y acceder a datos tabulares obtenidos de una consulta a bases de datos, como los resultados de una instrucción SELECT. Actúa como un cursor que apunta inicialmente a la primera fila, permitiendo recorrer y acceder a cada fila de manera secuencial o bidireccional, según su configuración. Común en

lenguajes como Java con JDBC, facilita la integración y manejo de resultados en bases de datos SQL en aplicaciones empresariales.

Connection: Una interfaz en Java permite conectarse a una base de datos específica para extraer información sobre tablas, procedimientos almacenados y ejecutar sentencias SQL. Para usarla, se crea un objeto con el método getConnection del DriverManager, proporcionando las credenciales (usuario y contraseña) y la cadena de conexión correspondiente.

Statement: Una interfaz en Java permite ejecutar sentencias SQL estáticas y retornar sus resultados. Se obtiene una instancia de este tipo ejecutando el método createStatement() en un objeto de tipo Connection.

Método next: Además de desplazarse de forma lineal dentro de la estructura, también devuelve un valor booleano que indica si el desplazamiento encontró un conjunto de datos válido. Esto permite determinar cuándo se ha alcanzado el final de los elementos.

1.2 **Conexión a la base de datos java**

Para garantizar estabilidad de la base de datos de deben seguir ciertos pasos los cuales son:

1. Descarga el Driver JDBC si aún no lo tienes.

2. Agrega el Driver descargado a las librerías del proyecto en Java.

3. Importa el paquete `java.sql.*;`.

4. Inicializa el Driver utilizando `Class.forName`.

5. Crea un objeto `Connection` a través del `DriverManager`.

6. Genera un objeto `Statement` para configurar sentencias SQL desde el `Connection` creado en el paso anterior.

7. Usa el objeto `Statement` para ejecutar la sentencia SQL mediante alguno de sus métodos.

8. Procesa los resultados de la sentencia ejecutada empleando un `ResultSet` y sus métodos auxiliares.

2. **Servlets y JSP**

Un ejemplo sencillo usando JSP y Servlets es crear un formulario en JSP para recolectar datos como nombre, edad y correo, enviando esta información a un servlet que la almacene en una base de datos, permitiendo al usuario visualizar los datos registrados.

Los servlets procesan las peticiones al servidor, gestionan datos de formularios, generan contenido dinámico y redirigen solicitudes. JSP (Java Server Pages) es un código Java que, ejecutado en el servidor, lee datos de formularios HTML, realiza consultas (generalmente a bases de datos) y devuelve una página HTML dinámica en tiempo real. Los archivos JSP combinan código Java con HTML en etiquetas `<% %>` y deben guardarse en la carpeta de contenido web del servidor. El software requerido incluye un IDE Java JEE (NetBeans o Eclipse) y un servidor web como Tomcat. Para trabajar con JSP y Servlets, se debe instalar y configurar el JDK, un IDE (NetBeans) y Tomcat para crear, ejecutar y probar los servlets y páginas JSP.

2.1 Los servlets y JSP (Java Server Pages) son tecnologías fundamentales en Java para el desarrollo de aplicaciones web dinámicas, donde los servlets procesan las solicitudes HTTP y JSP facilita la creación de contenido dinámico en HTML. Aquí tienes una visión más detallada de ambos, incluyendo ventajas, arquitectura, ciclo de vida, configuración y ejemplos de uso:

1. **Función y uso de Servlets y JSP**

- Servlets: Un servlet es un programa en Java que se ejecuta en un servidor web y actúa como un intermediario entre el servidor y la base de datos. Su función principal es gestionar las solicitudes de clientes y devolver una respuesta adecuada, habitualmente en HTML, que los usuarios pueden ver en sus navegadores. La capacidad de trabajar directamente con solicitudes y respuestas HTTP permite que los servlets manejen datos de formularios, realicen operaciones de bases de datos y generen contenido web en tiempo real.
- JSP: JSP (Java Server Pages) es una tecnología que permite insertar código Java directamente en el HTML. Esto permite a los desarrolladores construir páginas web dinámicas, personalizar el contenido en función de la entrada del usuario y realizar operaciones en tiempo real, como la recuperación de datos de bases de datos.

2. **Ventajas de los Servlets y JSP**

- Rendimiento: Los servlets ofrecen un rendimiento mucho más alto que el CGI (Common Gateway Interface), ya que los servlets no requieren crear un nuevo proceso para cada solicitud, sino que pueden reutilizar una instancia en cada petición.

- Portabilidad y reutilización: Dado que están escritos en Java, los servlets y JSP son altamente portables, lo que facilita la migración entre diferentes plataformas de servidores web.
- Eficiencia: Los servlets manejan múltiples solicitudes concurrentes con facilidad, utilizando un solo hilo por cliente, lo cual optimiza el uso de recursos y permite que la aplicación escale mejor.
- Seguridad: Java proporciona una capa adicional de seguridad, lo que es esencial en aplicaciones web donde se manejan datos sensibles de usuarios.

3. **Arquitectura y ciclo de vida de un Servlet**

- Arquitectura: Los servlets son administrados por el servidor de aplicaciones, que se encarga de inicializarlos, gestionar sus instancias y asegurar su eliminación cuando ya no son necesarios. Un servlet tiene acceso a recursos del servidor, como la base de datos y archivos de configuración, lo que le permite ejecutar operaciones críticas.
- Ciclo de Vida:
    - init(): Inicializa el servlet y se llama una vez cuando se carga en el servidor. Aquí se configuran las dependencias necesarias, como conexiones a bases de datos o variables de entorno.
    - service(): Gestiona las solicitudes HTTP. Este método llama a doGet() para manejar solicitudes GET y a doPost() para solicitudes POST, permitiendo la ejecución de operaciones basadas en la solicitud recibida.
    - destroy(): Elimina el servlet cuando deja de estar en uso, cerrando conexiones y liberando recursos.

4. **Configuración de un entorno de desarrollo para JSP y Servlets**

Para trabajar con JSP y Servlets, es fundamental configurar un entorno de desarrollo adecuado:

- Java JDK: Se necesita instalar el Java Development Kit (JDK), que incluye el compilador y herramientas necesarias para desarrollar en Java.
- IDE (NetBeans o Eclipse): Se recomienda un entorno de desarrollo como NetBeans o Eclipse que soporte Java Enterprise Edition (Java EE) para facilitar la creación de proyectos web.
- Servidor de aplicaciones (Tomcat): Tomcat es un servidor de código abierto que permite implementar aplicaciones Java web, incluyendo Servlets y JSP. Es compatible con aplicaciones web de nivel empresarial y soporta JSP y Servlets.

Si el servlet se conecta a una base de datos para recuperar información, el resultado se obtiene en un objeto ResultSet, que se puede procesar dentro del servlet para generar respuestas dinámicas.

Este flujo entre JSP y Servlets permite a los desarrolladores crear aplicaciones interactivas y personalizadas, con un alto grado de dinamismo y control sobre las solicitudes y respuestas del servidor.

2.2 **Formularios HTML con Servlets**

Los formularios en HTML permiten recoger información de los usuarios mediante elementos como campos de texto, botones de envío, casillas de verificación, entre otros. Para el procesamiento de estos formularios se utilizan Servlets , que son programas Java que funcionan en el lado del servidor. Un formulario se declara mediante la etiqueta <form>, especificando los atributos action(para la URL donde se enviarán los datos) y method(GET o POST). Mientras que

GET envía los datos a través de la URL (más adecuado para solicitudes de búsqueda), POST envía la información en el cuerpo de la solicitud, siendo más seguro para manejar datos sensibles.

```html
<form action="procesarDatos" method="post">
  <label for="nombre">Nombre:</label>
  <input type="text" id="nombre" name="nombre">
  <input type="submit" value="Enviar">
</form>
```

En el servlet correspondiente, se utiliza el método doPosto doGetpara gestionar la solicitud según el método del formulario. Por ejemplo:

protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException { String nombre = request.getParameter("nombre"); response.getWriter().println("Nombre recibido: " + nombre); }

2.3 **Sesión HTTP**

Una sesión HTTP permite almacenar y gestionar información entre múltiples solicitudes del mismo usuario. La clase HttpSessionen Java ayuda a rastrear al usuario mediante un identificador único. Para obtener una sesión, se utiliza request.getSession(), lo que retorna un objeto de tipo HttpSession. A partir de este, se pueden usar varios métodos importantes:

- setAttribute(String name, Object value): Guarde un objeto en la sesión con un nombre específico.
- getAttribute(String name): Recuperar un objeto almacenado.
- invalidate(): Finaliza la sesión y borra todos los datos asociados.

- getId(): Retorna el ID único de la sesión.
- isNew(): Verifica si la sesión es nueva.

El manejo de sesiones es crucial para autenticar a los usuarios, recordar sus preferencias y gestionar carritos de compra en sitios de comercio electrónico.

2.4 **Elementos de JSP**

JavaServer Pages (JSP) permite generar contenido dinámico en HTML utilizando componentes Java. Los elementos JSP se dividen en:

1. Scriptlets (<% ... %>) : Inserta código Java en un archivo JSP.
2. Expresiones (<%= ... %>) : Evalúa una expresión Java y muestra su resultado en la página.
3. Declaraciones (<%! ... %>) : Declara variables o métodos dentro de la página JSP.
4. Directivas (<%@ ... %>) : Configuran atributos de la página, como includepara insertar un archivo o pagepara definir configuraciones generales.
5. Acciones : Usan <jsp:...>para realizar tareas comunes como la inclusión dinámica de otros recursos o la transferencia de control a otro componente.

Estos elementos permiten separar la lógica de negocio y mantener la presentación manejada en HTML.

2.5 **MVC (Modelo-Vista-Controlador)**

El patrón MVC separa una aplicación web en tres componentes principales:

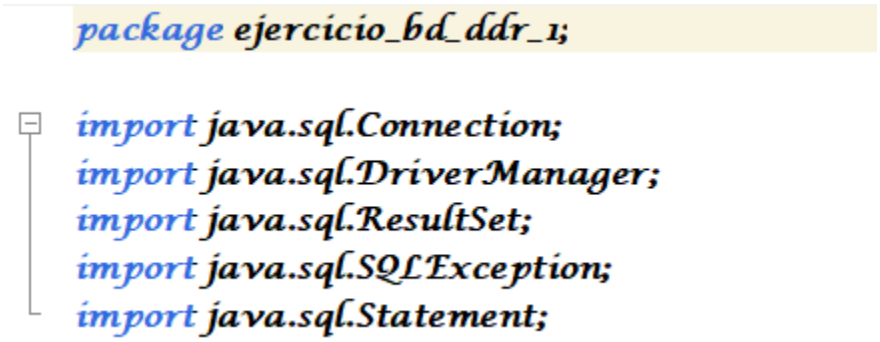
- Modelo : La lógica de la aplicación y la gestión de datos. En Java, se suelen usar JavaBeans o POJO para representar los modelos.
- Vista : La interfaz de usuario, usualmente implementada con JSP, donde se muestra la información al usuario.
- Controlador : La capa que gestiona las solicitudes del usuario, ampliamente implementada con Servlets, que se encargan de interactuar con el modelo y enviar datos a las vistas.

Esta arquitectura permite un desarrollo más organizado y facilita el mantenimiento de la aplicación al separar responsabilidades.

Este informe proporciona una visión detallada de cada concepto clave para desarrollar aplicaciones web en Java, asegurando un buen manejo de formularios, sesiones, generación dinámica de contenido y un diseño estructurado con el patrón MVC

Explicación base de datos

- Base de datos #1:

1. 

   Importación de paquetes de Java que se requieren para el óptimo funcionamiento del programa.

   - Java.sql.connection: representa la conexión a una base de datos específica, es un enlace entre java y la base de datos permitiendo enviar consultas y recibir resultados.
   - Java.sql.DriverManager: Administra los controladores de la base de datos. Se usa para establecer una conexión con la base mediante el método get connection.
   - Java.sql.Resultset: representa los resultados de una consulta a la base de datos.
   - Java.sql.SQLException: una clase que maneja errores relacionados con la base de datos como los problemas de conexión o fallos en la consulta.
   - Java.sql.statement: permite enviar las instrucciones SQL a la base de datos. Se puede ejecutar consultas como select, insert, update, delete y recibir los resultados.

2.

```
public class Ejercicio_BD_DDR_1 {

    public static void main(String[] args) {
```

Definición de clase principal y el método main, es el punto de entrada para la ejecución del programa.

3.

```
try {
    Class.forName("com.mysql.jdbc.Driver");
    //Clase Connection que nos permite conectarnos a una base de datos
    //dado una cadena de conexion y un usuario y pass
    Connection conexion = DriverManager.getConnection("jdbc:mysql://localhost/empleadoss_departamentoss", "root", "root");

    //Creamos una sentencia
    Statement sentencia = conexion.createStatement();
    //ResultSet guarda los resultados de la consulta
    ResultSet resultSet = sentencia.executeQuery("select * from empleados");
```

Establecimiento de la conexión a la base de datos:

- Try: es un bloque que permite cargar el controlador JDBC de MySQL, este permite que se comunique con MySQL.
- DriverManager.getConnection(): tiene tres parámetros: url de conexión "jdbc:mysql://localhost/empleadoss_departamentoss" este indica el tipo de base de datos, la ubicación del servidor y el nombre de la base de datos.
- Tenemos usuario: root y la contraseña también es root.
- Conexión: es una instancia de connection para mejor interacción de la base de datos.

- Statement (creación de sentencia): este permite ejecutar consultas SQL en la base de datos. Sentencia se utiliza para realizar operaciones sobre la conexión.
- ResultSet: ejecución de la consulta y obtención de resultados:
    1. execute Query: envía una consulta a la base de datos.
    2. ResultSet: objeto que guarda el resultado de la consulta anterior.
- Select*From empleados: consulta SQL que obtiene todos los registros de la tabla empleados.

4.

```
while (resultSet.next()) {
    System.out.println(resultSet.getString("nomemp"));
}
```

Procesamiento de los resultados:

- While: ciclo repetitivo que recorre cada fila del resultset.
- resultSet.next() avanza a la siguiente fila en el resultado.
- getString(“nomemp”): obtiene el valor de la columna nomemp como una cadena de texto, por cada iteración el valor de nomemp se imprime en la consola por medio del System.out.println.

6.

```
resultSet.close();
sentencia.close();

conexion.close();
```

Cierre de los recursos en el orden de uso.

7.

```
    } catch (ClassNotFoundException | SQLException ex) {
        System.out.println(ex.getMessage());
    }

}
```

Si ocurre un error al cargar el controlador (ClassNotFoundException) o al interactuar con la base de datos (SQLException), el bloque catch captura la excepción y muestra el mensaje de error en la consola.

- **Base de datos #2:**

```
package ejercicio_bd_ddr_2;

import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.logging.Level;
import java.util.logging.Logger;

public class Ejercicio_BD_DDR_2 {

    public static void main(String[] args) {
        try {

            //Creo la conexión
            ConexionMySQL conexion = new ConexionMySQL("localhost", "empleadoss_departamentoss", "root", "root");

            //Ejecuto la consulta
            conexion.ejecutarConsulta("select * from empleados");

            //Recojo el ResultSet
            ResultSet rs = conexion.getResultSet();
```

```
            //Muestro la consulta
            while (rs.next()) {
                System.out.println(rs.getString("nomEmp"));
            }

        } catch (SQLException ex) {
            System.out.println(ex.getMessage());
        }

    }

}
```

Creación de la conexión:

Aquí se crea una instancia de la clase ConexionMySQL, pasando como parámetros:

- "localhost": el servidor de la base de datos, que en este caso está en la misma máquina.
- "empleadoss_departamentoss": el nombre de la base de datos a la que te conectas.
- "root": el nombre de usuario de MySQL.
- "root": la contraseña asociada a ese usuario.

Esto es una forma encapsulada de inicializar una conexión sin escribir directamente el código de conexión (usando DriverManager o Connection.

Bsae #3

```xml
<?xml version="1.0" encoding="UTF-8" ?>

<Form version="1.3" maxVersion="1.9" type="org.netbeans.modules.form.forminfo.JFrameFormInfo">
  <Properties>
    <Property name="defaultCloseOperation" type="int" value="3"/>
    <Property name="title" type="java.lang.String" value="Login"/>
  </Properties>
  <SyntheticProperties>
    <SyntheticProperty name="formSizePolicy" type="int" value="1"/>
    <SyntheticProperty name="generateCenter" type="boolean" value="false"/>
  </SyntheticProperties>
  <AuxValues>
    <AuxValue name="FormSettings_autoResourcing" type="java.lang.Integer" value="0"/>
    <AuxValue name="FormSettings_autoSetComponentName" type="java.lang.Boolean" value="false"/>
    <AuxValue name="FormSettings_generateFQN" type="java.lang.Boolean" value="true"/>
    <AuxValue name="FormSettings_generateMnemonicsCode" type="java.lang.Boolean" value="false"/>
    <AuxValue name="FormSettings_i18nAutoMode" type="java.lang.Boolean" value="false"/>
    <AuxValue name="FormSettings_layoutCodeTarget" type="java.lang.Integer" value="1"/>
    <AuxValue name="FormSettings_listenerGenerationStyle" type="java.lang.Integer" value="0"/>
    <AuxValue name="FormSettings_variablesLocal" type="java.lang.Boolean" value="false"/>
    <AuxValue name="FormSettings_variablesModifier" type="java.lang.Integer" value="2"/>
  </AuxValues>

  <Layout>
    <DimensionLayout dim="0">
      <Group type="103" groupAlignment="0" attributes="0">
          <Group type="102" alignment="0" attributes="0">
              <EmptySpace min="-2" pref="19" max="-2" attributes="0"/>
              <Group type="103" groupAlignment="1" attributes="0">
                  <Component id="jLabel1" min="-2" max="-2" attributes="0"/>
                  <Component id="jLabel2" min="-2" max="-2" attributes="0"/>
              </Group>
              <EmptySpace min="-2" pref="37" max="-2" attributes="0"/>
              <Group type="103" groupAlignment="0" max="-2" attributes="0">
                  <Component id="btnLogueo" pref="149" max="32767" attributes="0"/>
                  <Component id="pwdPass" alignment="0" max="32767" attributes="0"/>
                  <Component id="txtUsuario" alignment="0" max="32767" attributes="0"/>
              </Group>
              <EmptySpace pref="40" max="32767" attributes="0"/>
          </Group>
      </Group>
    </DimensionLayout>
    <DimensionLayout dim="1">
      <Group type="103" groupAlignment="0" attributes="0">
          <Group type="102" alignment="0" attributes="0">
              <EmptySpace min="-2" pref="50" max="-2" attributes="0"/>
              <Group type="103" groupAlignment="3" attributes="0">
                  <Component id="txtUsuario" alignment="3" min="-2" max="-2" attributes="0"/>
```

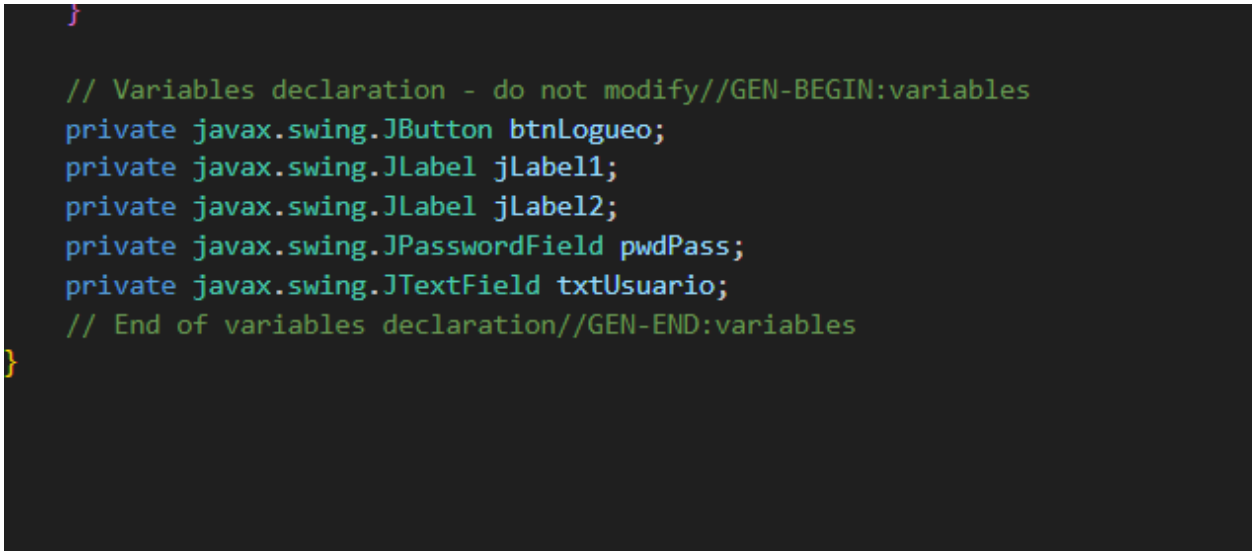
Este archivo XML define la estructura de un formulario de inicio de sesión en Java Swing creado con NetBeans. El formulario es una ventana de tipo JFrame titulada "Login" que permite al usuario ingresar un nombre de usuario y una contraseña. La propiedad defaultCloseOperation está configurada en EXIT_ON_CLOSE (valor 3), lo que significa que cerrar la ventana finalizará la aplicación. El diseño del formulario utiliza GroupLayout, organizando los componentes en dos dimensiones para asegurar un alineado y espaciado adecuados. Los componentes incluyen un campo de texto (JTextField) para el usuario (txtUsuario), un campo de contraseña (JPasswordField) para la clave (pwdPass), y dos etiquetas (JLabel) que muestran los textos "Usuario" y "Password" junto a los campos correspondientes. Además, hay un botón (JButton) etiquetado "Loguear" (btnLogueo) que, al ser presionado, ejecutará el método btnLogueoActionPerformed, el cual debe implementarse en el código fuente para manejar la autenticación. El diseño también incluye espacios vacíos (EmptySpace) para lograr una distribución limpia de los elementos en la ventana. Las propiedades auxiliares del archivo configuran la generación de código en NetBeans, como el uso de nombres de clase totalmente calificados y la creación automática de variables. En resumen, este archivo XML define una interfaz gráfica de usuario básica que permite iniciar sesión, y genera automáticamente el código Java en NetBeans, dejando la implementación de la funcionalidad de autenticación para el código Java asociado al evento del botón.

```
Run | Debug
public static void main(String args[]) {
    /* Set the Nimbus look and feel */
    //<editor-fold defaultstate="collapsed" desc=" Look and feel setting code (optional) ">
    /* If Nimbus (introduced in Java SE 6) is not available, stay with the default look and feel.
     * For details see http://download.oracle.com/javase/tutorial/uiswing/lookandfeel/plaf.html
     */
    try {
        for (javax.swing.UIManager.LookAndFeelInfo info : javax.swing.UIManager.getInstalledLookAndFeels()) {
            if ("Nimbus".equals(info.getName())) {
                javax.swing.UIManager.setLookAndFeel(info.getClassName());
                break;
            }
        }
    } catch (ClassNotFoundException ex) {
        java.util.logging.Logger.getLogger(Login.class.getName()).log(java.util.logging.Level.SEVERE, msg:null, ex);
    } catch (InstantiationException ex) {
        java.util.logging.Logger.getLogger(Login.class.getName()).log(java.util.logging.Level.SEVERE, msg:null, ex);
    } catch (IllegalAccessException ex) {
        java.util.logging.Logger.getLogger(Login.class.getName()).log(java.util.logging.Level.SEVERE, msg:null, ex);
    } catch (javax.swing.UnsupportedLookAndFeelException ex) {
        java.util.logging.Logger.getLogger(Login.class.getName()).log(java.util.logging.Level.SEVERE, msg:null, ex);
    }
    //</editor-fold>
    //</editor-fold>
    //</editor-fold>
    //</editor-fold>

    /* Create and display the form */
    java.awt.EventQueue.invokeLater(new Runnable() {
        public void run() {
            new Login().setVisible(b:true);
        }
    });
}
```

Este código Java implementa una ventana de inicio de sesión en una aplicación Swing llamada Login. La clase extiende JFrame, lo que permite que la interfaz gráfica funcione como una ventana independiente. En el constructor Login(), se llama al método initComponents() para inicializar todos los elementos de la interfaz y situarla en el centro de la pantalla con this.setLocationRelativeTo(null). La interfaz incluye un campo de texto (JTextField) para el nombre de usuario (txtUsuario), un campo de contraseña (JPasswordField) para la clave (pwdPass), y dos etiquetas (JLabel) que indican "Usuario" y "Password". Además, hay un botón (JButton) denominado "Loguear" (btnLogueo), que desencadena el método btnLogueoActionPerformed cuando se hace clic. En el método

btnLogueoActionPerformed, se recuperan los valores de los campos de usuario y contraseña, y se verifica su autenticación mediante una consulta en la base de datos. Si se encuentra un registro que coincide, se crea y muestra una nueva instancia de Principal (una ventana diferente que se asume representa la interfaz principal de la aplicación), mientras que la ventana de Login se cierra usando this.dispose(). Si la autenticación falla, se muestra un cuadro de diálogo de error utilizando JOptionPane para indicar que el acceso es incorrecto. El método main configura el "look and feel" de la aplicación usando Nimbus (si está disponible), y posteriormente crea y muestra la ventana de Login en el hilo de eventos de la GUI.



```
    }

    // Variables declaration - do not modify//GEN-BEGIN:variables
    private javax.swing.JButton btnLogueo;
    private javax.swing.JLabel jLabel1;
    private javax.swing.JLabel jLabel2;
    private javax.swing.JPasswordField pwdPass;
    private javax.swing.JTextField txtUsuario;
    // End of variables declaration//GEN-END:variables
}
```

```
package ejercicio_bd_ddr_3;


public class VariablesGlobales {

    public static ConexionMySQL conexion = new ConexionMySQL("localhost",
"login", "root", "root");

}
```

El código que describes utiliza una clase VariablesGlobales para almacenar y gestionar una conexión a una base de datos MySQL de manera global en la aplicación. La clase tiene una variable estática pública conexion de tipo ConexionMySQL, que se inicializa en el momento de su declaración con los parámetros necesarios para establecer la conexión: la dirección del servidor de la base de datos (localhost), el nombre de la base de datos (login), y las credenciales de acceso (root como usuario y contraseña). Al ser una variable estática, conexion puede ser accedida desde cualquier parte del programa sin necesidad de crear una nueva instancia de VariablesGlobales. Esto facilita el acceso a la base de datos desde diferentes partes de la aplicación, lo que es especialmente útil en aplicaciones que realizan múltiples consultas o modificaciones en la base de datos.

```xml
<AuxValues>
  <AuxValue name="FormSettings_autoResourcing" type="java.lang.Integer" value="0"/>
  <AuxValue name="FormSettings_autoSetComponentName" type="java.lang.Boolean" value="false"/>
  <AuxValue name="FormSettings_generateFQN" type="java.lang.Boolean" value="true"/>
  <AuxValue name="FormSettings_generateMnemonicsCode" type="java.lang.Boolean" value="false"/>
  <AuxValue name="FormSettings_i18nAutoMode" type="java.lang.Boolean" value="false"/>
  <AuxValue name="FormSettings_layoutCodeTarget" type="java.lang.Integer" value="1"/>
  <AuxValue name="FormSettings_listenerGenerationStyle" type="java.lang.Integer" value="0"/>
  <AuxValue name="FormSettings_variablesLocal" type="java.lang.Boolean" value="false"/>
  <AuxValue name="FormSettings_variablesModifier" type="java.lang.Integer" value="2"/>
</AuxValues>
```

El código XML que has proporcionado describe una interfaz gráfica de usuario (GUI) en el entorno de desarrollo de NetBeans, específicamente para una clase que extiende JFrame. Esta interfaz parece ser una ventana para un sistema de autenticación, que muestra un mensaje y un botón de cierre de sesión. En la sección de propiedades (Properties), se especifica la configuración básica de la ventana. Se establece el comportamiento al cerrar la ventana con defaultCloseOperation en 3, lo que significa que la aplicación se cierra cuando la ventana se cierra. Además, el título de la ventana se establece como "Principal". Dentro de SyntheticProperties, se define cómo debe comportarse la forma (ventana) en términos de diseño. Por ejemplo, formSizePolicy está configurado en 1, lo que indica que el tamaño de la forma es automático, y generateCenter está en false, lo que sugiere que la forma no se genera centrada por defecto. En la sección AuxValues, se encuentran diversas configuraciones de la forma relacionadas con el diseño y la internacionalización (i18n). Estas configuraciones permiten controlar la manera en que se generan los elementos de la interfaz, como las

variables locales, el estilo de generación de oyentes (listeners), y si se generan o no códigos de mnemotecnia para teclas rápidas. En cuanto al diseño de la interfaz, se especifica mediante DimensionLayout, que organiza los componentes en una cuadrícula. La estructura del diseño tiene varios grupos y espacios vacíos que separan los componentes. Se observa que el btnCerrar (un botón) y jLabel1 (una etiqueta de texto) están dispuestos de una manera ordenada y con espacio adecuado entre ellos, lo que permite que la interfaz sea clara y fácil de usar. Finalmente, en SubComponents, se definen los componentes específicos de la interfaz: un botón (btnCerrar) y una etiqueta (jLabel1). El botón tiene la propiedad text con el valor "Cerrar sesión" y un manejador de eventos (actionPerformed), lo que significa que cuando el usuario haga clic en el botón, se ejecutará el método btnCerrarActionPerformed. La etiqueta tiene un texto con el mensaje "¡Estas logueado!" y un estilo de fuente específico, en este caso, utilizando la fuente "Tahoma" de tamaño 14. En resumen, este XML describe una ventana básica de Java Swing con un botón de cierre de sesión y una etiqueta que muestra un mensaje de autenticación. Está configurada para una interfaz de usuario sencilla con un diseño que organiza los componentes de manera funcional y accesible.

```java
@SuppressWarnings("unchecked")
// <editor-fold defaultstate="collapsed" desc="Generated Code">//GEN-BEGIN:initComponents
private void initComponents() {

    btnCerrar = new javax.swing.JButton();
    jLabel1 = new javax.swing.JLabel();

    setDefaultCloseOperation(javax.swing.WindowConstants.EXIT_ON_CLOSE);
    setTitle(title:"Principal");

    btnCerrar.setText(text:"Cerrar sesion");
    btnCerrar.addActionListener(new java.awt.event.ActionListener() {
        public void actionPerformed(java.awt.event.ActionEvent evt) {
            btnCerrarActionPerformed(evt);
        }
    });

    jLabel1.setFont(new java.awt.Font(name:"Tahoma", style:0, size:14)); // NOI18N
    jLabel1.setHorizontalAlignment(javax.swing.SwingConstants.CENTER);
    jLabel1.setText(text:"¡Estas logueado!");

    javax.swing.GroupLayout layout = new javax.swing.GroupLayout(getContentPane());
    getContentPane().setLayout(layout);
    layout.setHorizontalGroup(
        layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
        .addGroup(layout.createSequentialGroup()
            .addGap(min:21, pref:21, max:21)
            .addGroup(layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
                .addComponent(btnCerrar, javax.swing.GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.DEFAULT_SIZE, Short.MAX_VALUE)
                .addGroup(layout.createSequentialGroup()
                    .addComponent(jLabel1, javax.swing.GroupLayout.PREFERRED_SIZE, pref:238, javax.swing.GroupLayout.PREFERRED_SIZE)
                    .addGap(min:0, pref:19, Short.MAX_VALUE)))
            .addContainerGap())
    );
    layout.setVerticalGroup(
        layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
        .addGroup(javax.swing.GroupLayout.Alignment.TRAILING, layout.createSequentialGroup()
            .addGap(min:23, pref:23, max:23)
            .addComponent(jLabel1, javax.swing.GroupLayout.DEFAULT_SIZE, pref:99, Short.MAX_VALUE)
            .addGap(min:18, pref:18, max:18)
            .addComponent(btnCerrar)
            .addContainerGap())
    );

    pack();
}// </editor-fold>//GEN-END:initComponents
```

La clase Principal en el código proporcionado extiende de javax.swing.JFrame, lo que permite crear una ventana gráfica en una aplicación Java utilizando la biblioteca Swing. Esta ventana contiene dos componentes principales: un botón y una etiqueta. El propósito de la ventana es mostrar un mensaje indicando que el usuario está logueado y ofrecer la opción de cerrar sesión mediante un botón. En el constructor de la clase Principal, se llama al método initComponents(), que es responsable de la creación y disposición de los componentes gráficos en la ventana. Dentro de este método, se

configura un botón (btnCerrar) con el texto "Cerrar sesión", el cual está vinculado a un evento. Al hacer clic en el botón, se ejecuta el manejador de eventos btnCerrarActionPerformed, que, en lugar de cerrar directamente la ventana, abre una nueva instancia de la ventana de inicio de sesión (Login) y cierra la ventana actual utilizando this.dispose(). En cuanto al diseño visual de la interfaz, se utiliza un GroupLayout para organizar los componentes. La etiqueta (jLabel1) que muestra el mensaje "¡Estas logueado!" se posiciona en la parte superior de la ventana, seguida por el botón de "Cerrar sesión". Este diseño asegura que los elementos estén alineados de manera coherente y adecuada. El uso de GroupLayout permite que la interfaz se adapte a distintos tamaños de ventana, manteniendo la disposición de los componentes de forma ordenada. El método btnCerrarActionPerformed no solo cierra la ventana actual, sino que también centra la nueva ventana de inicio de sesión en la pantalla con this.setLocationRelativeTo(null);. Este comportamiento es típico en aplicaciones que requieren un flujo de navegación entre diferentes pantallas, como un sistema de autenticación en el que el usuario pueda cerrar sesión y volver a ingresar sus credenciales. En resumen, el código implementa una ventana funcional con un mensaje de autenticación y un botón para cerrar sesión, gestionando la transición entre la ventana de inicio de sesión y la ventana principal de manera eficiente y organizada.

Base #4:

```
package ejercicio_bd_ddr_3;

import java.sql.*;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.swing.DefaultListCellRenderer;
import javax.swing.JComboBox;
import javax.swing.JLabel;
import javax.swing.JList;
import javax.swing.JOptionPane;
import javax.swing.table.DefaultTableModel;
```

java.sql.*: Este paquete incluye clases para conectarse y trabajar con bases de datos SQL. Aquí encontrarás clases como Connection, Statement, ResultSet, etc., que son esenciales para realizar consultas y actualizaciones en una base de datos.

java.util.logging.Level y java.util.logging.Logger: Estos paquetes se usan para registrar (log) información, advertencias o errores en la aplicación. Logger permite escribir mensajes de registro, y Level define los niveles de severidad (como INFO, WARNING, SEVERE). Esto es útil para rastrear problemas en la ejecución y obtener información de depuración.

javax.swing.* (como DefaultListCellRenderer, JComboBox, JLabel, JList, JOptionPane, DefaultTableModel): Estas clases son parte de Swing, la biblioteca de Java para crear interfaces gráficas de usuario (GUIs).

- **DefaultListCellRenderer**: Permite personalizar la apariencia de los elementos de una lista.
- **JComboBox**: Es una lista desplegable que permite seleccionar un valor de una lista.
- **JLabel**: Permite mostrar texto o imágenes en la interfaz.

- **JList**: Muestra una lista de elementos.
- **JOptionPane**: Se usa para mostrar diálogos como mensajes de información, advertencias o diálogos de confirmación.
- **DefaultTableModel**: Es una clase que gestiona los datos en una tabla (JTable), facilitando la actualización y manipulación de la información mostrada en tablas.

La clase ConexionDB en este código está diseñada para gestionar la conexión a una base de datos en Java y cuenta con algunos atributos y dos constructores sobrecargados. Primero, los atributos principales incluyen conexion, sentencia y resultSet. Estos atributos son protegidos (protected), lo que significa que pueden ser accedidos desde esta clase y sus subclases. conexion es una instancia de Connection que representa la conexión establecida con la base de datos. sentencia es un objeto Statement utilizado para ejecutar consultas SQL en la base de datos, y resultSet almacena el resultado de estas consultas. La clase tiene dos constructores sobrecargados que permiten establecer la conexión a la base de datos de diferentes maneras. El primer constructor recibe dos parámetros: claseNombre (el nombre de la clase del controlador de la base de datos, como "com.mysql.cj.jdbc.Driver" para MySQL) y cadenaConexion (la URL de conexión a la base de datos, como "jdbc:mysql://localhost:3306/miBaseDeDatos"). Este constructor utiliza Class.forName(claseNombre) para cargar el controlador de la base de datos y luego establece la conexión con DriverManager.getConnection(cadenaConexion). También configura la conexión para que las transacciones se confirmen automáticamente (conexion.setAutoCommit(true)), lo que significa que cada operación en la base de datos se guardará de inmediato. Si ocurre algún error al cargar el controlador o al conectarse, el mensaje de error se muestra en la consola. El segundo constructor agrega autenticación a la conexión. Además de claseNombre y cadenaConexion, también recibe usuario y pass (nombre de usuario y

contraseña), que se utilizan para establecer una conexión autenticada con la base de datos. Al igual que en el primer constructor, carga el controlador de la base de datos y establece la conexión, capturando y mostrando cualquier excepción que pueda ocurrir. En resumen, ConexionDB es una clase diseñada para simplificar el proceso de conexión a una base de datos en Java, ya sea con o sin autenticación, y configurar las transacciones para confirmarse automáticamente. Si se produce un error durante la carga del controlador o la conexión, el mensaje de error se imprime en la consola, lo que permite continuar la ejecución del programa sin interrupciones abruptas.

```java
public class ConexionDB {

    protected Connection conexion;
    protected Statement sentencia;
    protected ResultSet resultSet;

    /**
     * @param claseNombre
     * @param cadenaConexion
     */
    public ConexionDB(String claseNombre, String cadenaConexion) {
        try {
            Class.forName(claseNombre);
            conexion = DriverManager.getConnection(cadenaConexion);
            conexion.setAutoCommit(autoCommit:true);
        } catch (ClassNotFoundException | SQLException ex) {
            System.out.println(ex.getMessage());
        }
    }

    public ConexionDB(String claseNombre, String cadenaConexion, String usuario, String pass) {
        try {
            Class.forName(claseNombre);
            conexion = DriverManager.getConnection(cadenaConexion, usuario, pass);
            conexion.setAutoCommit(autoCommit:true);
        } catch (ClassNotFoundException | SQLException ex) {
            System.out.println(ex.getMessage());
        }
    }
```

Esta sección del código en la clase ConexionDB proporciona métodos para gestionar la conexión a la base de datos y realizar operaciones esenciales como confirmar o revertir transacciones, así como liberar los recursos utilizados. Los primeros métodos son los métodos de acceso (getSentencia, getconexion, y getResultSet), que devuelven los objetos Statement, Connection y ResultSet respectivamente. Estos métodos permiten acceder desde fuera de la clase a estos componentes clave, que son necesarios para ejecutar consultas SQL y obtener resultados. Luego, el método commit() permite confirmar transacciones, guardando todos los cambios realizados en la base de datos desde la última vez que se ejecutó un commit o rollback. Esto es útil cuando deseas asegurarte de que las modificaciones en la base de datos sean permanentes. Si ocurre un error al intentar confirmar la transacción, se captura una excepción y se registra el error mediante Logger, con un nivel de severidad alto (Level.SEVERE). El método rollback() permite revertir transacciones, es decir, deshacer los cambios realizados en la base de datos desde el último commit o rollback. Esto es útil en situaciones donde ocurre un error y deseas devolver la base de datos a un estado anterior. Al igual que con commit, se captura cualquier excepción y se registra un mensaje de error. Los métodos de cierre de recursos (cerrarResult, cerrarSentencia, y cerrarConexion) aseguran que los recursos se liberen adecuadamente después de usarlos. cerrarResult cierra el ResultSet, que almacena los resultados de las consultas SQL, mientras que cerrarSentencia cierra el Statement utilizado para ejecutar dichas consultas. Finalmente, cerrarConexion cierra la conexión a la base de datos y, antes de hacerlo, verifica que resultSet y sentencia no sean nulos para cerrarlos también si es necesario. Cada uno de estos métodos captura y registra cualquier error que ocurra al intentar cerrar estos recursos. En conjunto, estos métodos aseguran un manejo seguro y eficiente de la conexión a la base de datos, permitiendo realizar confirmaciones o reversiones de transacciones según sea necesario y liberando los

recursos de manera controlada. Esto ayuda a evitar problemas de rendimiento y fugas de memoria, manteniendo la aplicación y la base de datos estables.

```java
public void commit() {

    try {
        conexion.commit();
    } catch (SQLException ex) {
        Logger.getLogger(ConexionDB.class.getName()).log(Level.SEVERE, msg:null, ex);
    }
}

/**
 * Vuelve a un estado previo a la base de datos
 */
public void rollback() {

    try {
        conexion.rollback();
    } catch (SQLException ex) {
        Logger.getLogger(ConexionDB.class.getName()).log(Level.SEVERE, msg:null, ex);
    }
}

/**
 * Cierra el ResultSet
 */
public void cerrarResult() {
    try {
        resultSet.close();
    } catch (SQLException ex) {
        Logger.getLogger(ConexionDB.class.getName()).log(Level.SEVERE, msg:null, ex);
    }
}
```

Este método, llamado rellenaComboBoxBDInt, está diseñado para rellenar un JComboBox en la interfaz de usuario con datos provenientes de una base de datos. Toma cuatro parámetros: cmb (el JComboBox que se va a rellenar), tabla (el nombre de la tabla en la base de datos de la que se obtendrán los datos), columna (la columna específica de la cual se extraerán los valores) y condicion (una condición opcional para filtrar los datos, sin incluir la palabra clave WHERE). A continuación, se describe cómo funciona este método paso a paso. Primero, se limpia el JComboBox eliminando todos sus elementos mediante cmb.removeAllItems(). Luego, se crea un Statement para ejecutar consultas SQL, y se realiza una consulta para seleccionar valores

distintos de la columna especificada en la tabla de la base de datos (select distinct columna from tabla). Esto devuelve un ResultSet llamado consulta con los datos que se quieren agregar al JComboBox. Si se proporciona una condición (es decir, si condicion no está vacío), se crea un segundo Statement llamado smAux y se ejecuta otra consulta que incluye la condición (select distinct columna from tabla where condicion). Este ResultSet, llamado correspondiente, se usa para determinar cuál valor de la columna debe seleccionarse en el JComboBox como el valor predeterminado. A continuación, el método itera sobre los resultados de consulta y agrega cada valor al JComboBox. Si el valor actual coincide con el valor obtenido en correspondiente, ese valor se establece como seleccionado en el JComboBox. Si no se especifica ninguna condición, el método simplemente agrega todos los valores de consulta al JComboBox sin verificar ninguna coincidencia. Al final, ambos ResultSet y Statement se cierran para liberar recursos, y cualquier excepción que ocurra durante el proceso se captura y registra usando Logger. En resumen, el método rellenaComboBoxBDInt extrae valores de una columna de una tabla en la base de datos y los agrega a un JComboBox. Si se especifica una condición, el método también selecciona automáticamente el valor correspondiente en el JComboBox. Este proceso permite que la interfaz se actualice dinámicamente con datos específicos de la base de datos según los criterios de filtrado dados.

```
public void rellenaComboBoxBDInt(JComboBox cmb, String tabla, String columna, String condicion) {

    cmb.removeAllItems();

    Statement sm;
    try {
        sm = conexion.createStatement();

        ResultSet consulta = sm.executeQuery("select distinct " + columna + " from " + tabla);

        ResultSet correspondiente = null;

        if (!condicion.equals(anObject:"")) {

            Statement smAux = conexion.createStatement();

            correspondiente = smAux.executeQuery("select distinct " + columna + " from " + tabla + " where " + condicion);
            correspondiente.next();

            while (consulta.next()) {

                cmb.addItem(consulta.getInt(columna));
                if (correspondiente.getInt(columna) == consulta.getInt(columna)) {
                    cmb.setSelectedItem(correspondiente.getInt(columna));
                }
            }

            correspondiente.close();
            smAux.close();
        } else {

            while (consulta.next()) {

                cmb.addItem(consulta.getInt(columna));

            }

        }
```

- Base de datos #5:

  #1:

```
package ejercicio_bd_ddr_5;

import com.db4o.Db4oEmbedded;
import com.db4o.ObjectContainer;
import com.db4o.ObjectSet;
import java.io.File;

public class Ejercicio_BD_DDR_5 {
```

1. **package ejercicio_bd_ddr_5;**
   - Esta línea define el paquete (package) al que pertenece la clase

- Los paquetes en Java sirven para organizar y agrupar clases relacionadas
- Ayuda a evitar conflictos de nombres entre clases

2. **import com.db4o.Db4oEmbedded;**
    - Importa la clase Db4oEmbedded de la biblioteca db4o
    - DB4O es una base de datos orientada a objetos para Java
    - Esta clase específica permite crear y abrir bases de datos embebidas

3. **import com.db4o.ObjectContainer;**
    - Importa la interfaz ObjectContainer
    - Es la interfaz principal para trabajar con bases de datos db4o
    - Permite realizar operaciones CRUD (Crear, Leer, Actualizar, Eliminar) en la base de datos

4. **import com.db4o.ObjectSet;**
    - Importa la clase ObjectSet
    - Representa un conjunto de objetos recuperados de la base de datos
    - Se usa para almacenar y manipular los resultados de las consultas

5. **import java.io.File;**
    - Importa la clase File del paquete java.io
    - Permite trabajar con archivos y directorios en el sistema de archivos
    - En este contexto, probablemente se use para especificar la ubicación de la base de datos

6. **public class Ejercicio_BD_DDR_5 {**
    - Define una clase pública llamada Ejercicio_BD_DDR_5
    - Esta será la clase principal donde se implementará la funcionalidad
    - El nombre sugiere que es un ejercicio relacionado con bases de datos

Es el inicio de un programa que trabajará con una base de datos DB4O, que es una base de datos orientada a objetos. Las importaciones indican que se realizarán operaciones de base de datos embebida con manipulación de archivos.

```
public static void main(String[] args) {

    Persona p1 = new Persona("Fernando", 30, 1.73, 73);
    Persona p2 = new Persona("Pepe", 30, 1.75, 80);
    Persona p3 = new Persona("Alfredo", 20, 1.9, 90);
    Persona p4 = new Persona("Roberto", 35, 1.70, 70);
    Persona p5 = new Persona("Domingo", 30, 1.73, 80);
```

1. **public static void main(String[] args)**
    - Es el método principal del programa

- static: indica que el método pertenece a la clase y no a una instancia
- void: indica que no retorna ningún valor
- String[] args: permite recibir argumentos desde la línea de comandos
-

2. **Creación de objetos Persona:**

```
Persona p1 = new Persona("Fernando", 30, 1.73, 73);
Persona p2 = new Persona("Pepe", 30, 1.75, 80);
Persona p3 = new Persona("Alfredo", 20, 1.9, 90);
Persona p4 = new Persona("Roberto", 35, 1.70, 70);
Persona p5 = new Persona("Domingo", 30, 1.73, 80);
```

- Crea 5 objetos Persona con diferentes valores (nombre, edad, altura, peso)

```
File f = new File("personas.db4o");
f.delete();

ObjectContainer db = Db4oEmbedded.openFile(f.getAbsolutePath());

db.store(p1);
db.store(p2);
db.store(p3);
db.store(p4);
db.store(p5);
```

1. **File f = new File("personas.db4o");**
    - Crea un nuevo archivo llamado "personas.db4o"
    - Este será el archivo de la base de datos donde se guardarán los datos
2. **f.delete();**

- Elimina el archivo si ya existe
- Esto asegura empezar con una base de datos limpia

3. **ObjectContainer db = Db4oEmbedded.openFile(f.getAbsolutePath())**;
    - Crea una conexión a la base de datos
    - getAbsolutePath() obtiene la ruta completa del archivo
    - Db4oEmbedded.openFile() abre o crea el archivo de base de datos
    - La conexión se guarda en la variable db

```
db.store(p1);
db.store(p2);
db.store(p3);
db.store(p4);
db.store(p5);
```

- Cada línea db.store() guarda un objeto Persona en la base de datos
- p1 hasta p5 son las personas que creamos anteriormente
- Los datos se almacenan de forma permanente en el archivo "personas.db4o"

```
Persona p = new Persona(null, 30, 1.73, 0);
ObjectSet<Persona> result = db.queryByExample(p)

while(result.hasNext()){
    System.out.println(result.next());
}

db.close();
}
```

1. **Persona p = new Persona(null, 30, 1.73, 0);**
    - Crea una "plantilla" de búsqueda

- El null en nombre significa que no importa el nombre
- Busca personas con edad 30 y altura 1.73
- El 0 en peso significa que no importa el peso

2. **ObjectSet<Persona> result = db.queryByExample(p);**
    - Realiza una búsqueda en la base de datos
    - Busca personas que coincidan con la plantilla (p)
    - Guarda los resultados en la variable result
3. while(result.hasNext()){
    - Inicia un bucle que se ejecuta mientras haya resultados
    - hasNext() verifica si hay más personas para mostrar
4. **System.out.println(result.next());**
    - Imprime cada persona encontrada
    - next() obtiene la siguiente persona del resultado
    - Se ejecuta una vez por cada persona que coincida con la búsqueda
5. **db.close();**
    - Cierra la conexión con la base de datos
    - Es importante cerrar la conexión para liberar recursos

#2:

```
public class Persona {
```

- Esta línea declara una clase pública llamada "Persona". El modificador public significa que la clase es accesible desde cualquier otra clase.

```
private String nombre;
private int edad;
private double peso;
private double altura;
```

- Estas son declaraciones de atributos/variables de instancia de la clase:
- private significa que estos atributos sólo son accesibles dentro de la clase Persona
- String nombre: almacena el nombre de la persona como texto
- int edad: almacena la edad como número entero
- double peso: almacena el peso como número decimal
- double altura: almacena la altura como número decimal

```
public Persona() {
}
```

- Esta es la declaración del constructor de la clase
- Un constructor es un método especial que se ejecuta cuando se crea un nuevo objeto de la clase
- public indica que el constructor puede ser llamado desde cualquier otra clase
- Este parece ser un constructor vacío (sin parámetros) ya que solo se muestra el inicio con {

Este código representa la estructura básica de una clase que modelaría los datos de una persona, con sus atributos básicos como nombre, edad, peso y altura. El encapsulamiento con **private** es una buena práctica ya que protege los datos y solo permite modificarlos a través de métodos específicos.

```java
public Persona(String nombre, int edad, double peso, double altura) {
    this.nombre = nombre;
    this.edad = edad;
    this.peso = peso;
    this.altura = altura;
}
```

1. **public Persona(String nombre, int edad, double peso, double altura)**: Esta es la declaración del constructor de la clase **Persona.** Se llama **Persona** porque el nombre de un constructor debe coincidir con el nombre de la clase. Este constructor tiene cuatro parámetros:
    - **String nombre:** un parámetro de tipo **String** que representa el nombre de la persona.
    - **int edad:** un parámetro de tipo **int** que representa la edad de la persona.
    - **double peso:** un parámetro de tipo **double** que representa el peso de la persona.
    - **double altura:** un parámetro de tipo **double** que representa la altura de la persona.

2. **this.nombre = nombre;**: Asigna el valor del parámetro **nombre** al atributo **nombre** de la clase **Persona. this.nombre** hace referencia al atributo de la instancia actual de **Persona**, mientras que **nombre** es el parámetro que se recibió en el constructor.

3. **this.edad = edad;**: Similar a la línea anterior, esta línea asigna el valor del parámetro edad al atributo **edad** de la clase. **this.edad** hace referencia al atributo de la instancia, mientras que **edad** es el valor recibido como parámetro.
4. **this.peso = peso;**: Asigna el valor del parámetro **peso** al atributo **peso** de la instancia de **Persona.**
5. **this.altura = altura;**: Asigna el valor del parámetro **altura** al atributo **altura** de la instancia.

```java
public String getNombre() {
    return nombre;
}

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

public int getEdad() {
    return edad;
}

public void setEdad(int edad) {
    this.edad = edad;
}

public double getPeso() {
    return peso;
}

public void setPeso(double peso) {
    this.peso = peso;
}

public double getAltura() {
    return altura;
}

public void setAltura(double altura) {
    this.altura = altura;
}

@Override
public String toString() {
    return "Persona{" + "nombre=" + nombre + ", edad=" + edad + ", peso=" + peso + ", altura=" + altura + '}';
}
```

Getters

Los getters son métodos que devuelven el valor de un atributo.

1. **public String getNombre() { return nombre; }**
    - Este método devuelve el valor del atributo nombre.
    - Tipo de retorno: String.
    -
2. **public int getEdad() { return edad; }**
    - Este método devuelve el valor del atributo edad.
    - Tipo de retorno: int.
3. **public double getPeso() { return peso; }**
    - Este método devuelve el valor del atributo peso.
    - Tipo de retorno: double.
4. **public double getAltura() { return altura; }**
    - Este método devuelve el valor del atributo altura.
    - Tipo de retorno: double.

Setters

Los setters son métodos que establecen el valor de un atributo.

5. **public void setNombre(String nombre) { this.nombre = nombre; }**
    - Este método permite asignar un nuevo valor al atributo nombre.
    - this.nombre se refiere al atributo de la instancia, mientras que el parámetro nombre es el nuevo valor que se le quiere asignar.
    - No tiene valor de retorno (void).
6. **public void setEdad(int edad) { this.edad = edad; }**

- Este método permite asignar un nuevo valor al atributo edad.
- this.edad se refiere al atributo de la instancia, y edad es el valor pasado como parámetro.

7. **public void setPeso(double peso) { this.peso = peso; }**
    - Este método permite asignar un nuevo valor al atributo peso.
    - this.peso se refiere al atributo de la instancia, y peso es el valor pasado como parámetro.

8. **public void setAltura(double altura) { this.altura = altura; }**
    - Este método permite asignar un nuevo valor al atributo altura.
    - this.altura se refiere al atributo de la instancia, y altura es el valor pasado como parámetro.

Método toString

9. **@Override**
    - Esta anotación indica que el método toString está sobrescribiendo el método toString heredado de la clase Object.

10. **public String toString() { return "Persona{" + "nombre=" + nombre + ", edad=" + edad + ", peso=" + peso + ", altura=" + altura + '}'; }**

    - Este método devuelve una representación en texto del objeto Persona.
    - Concatena el nombre de la clase y cada uno de los atributos (nombre, edad, peso, altura) en una cadena que sigue el formato "Persona{nombre=valor, edad=valor, peso=valor, altura=valor}".
    - Tipo de retorno: String.

- Base de datos #6:
    - #1

1. **Importaciones:**

```
import com.db4o.Db4oEmbedded;
import com.db4o.ObjectContainer;
import com.db4o.ObjectSet;
import java.io.File;
```

    - Importa las clases necesarias de la biblioteca Db4o y Java. Estas clases permiten manipular la base de datos **(Db4oEmbedded, ObjectContainer, ObjectSet)** y trabajar con archivos (File).

2. **Declaración de la clase principal**

```
public class Ejercicio_BD_DDR_6 {
```

- Define la clase Ejercicio_BD_DDR_6, que contiene el método **main.**

3. Método principal **main:**

```
    public static void main(String[] args) {
```

- Es el punto de entrada del programa.

4. **Creación del archivo de base de datos:}**

```
    File f = new File("personas.db4o");
```

- Crea un archivo **personas.db4o** que servirá como la base de datos para almacenar objetos **Persona.**

5. **Apertura de la base de datos:**

```
ObjectContainer db = Db4oEmbedded.openFile(f.getAbsolutePath());
```

- Abre la base de datos en el archivo especificado y devuelve un **ObjectContainer,** que representa la conexión a la base de datos.

6. **Consulta de un objeto Persona con nombre "Fernando":**

```
Persona p = new Persona("Fernando", 0, 0, 0);
ObjectSet<Persona> result = db.queryByExample(p);
```

- Crea un objeto **Persona** con el nombre "Fernando" y otros atributos con valor **0.** Luego, consulta la base de datos buscando un objeto similar usando **queryByExample.**

7. Actualizar edad de "Fernando":

```
if (result.hasNext()) {
    Persona pAct = result.next();

    pAct.setEdad(35);
    db.store(pAct);
}
```

- Si se encuentra una coincidencia **(hasNext**), recupera el objeto **Persona** y actualiza su edad a **35.** Después, guarda los cambios en la base de datos con **store.**

8. Consulta y eliminación de "Domingo":

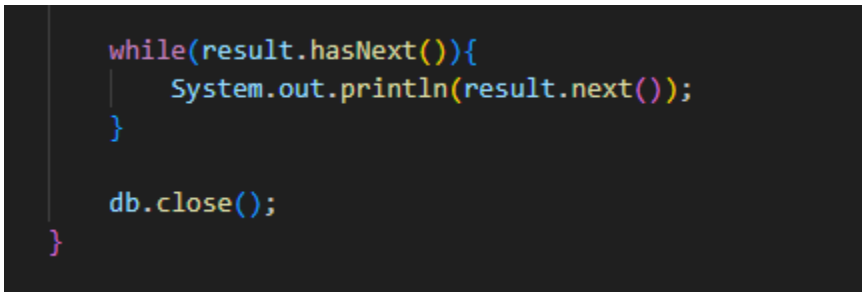
```
p = new Persona("Domingo", 0, 0, 0);
result = db.queryByExample(p);

if (result.hasNext()) {
    Persona pDel = result.next();

    db.delete(pDel);
}
```

- Cambia **p** para representar una persona llamada **"Domingo"**. Luego, realiza una consulta similar. Si "Domingo" existe, lo recupera y elimina de la base de datos usando **delete.**

9. Consulta y visualización de todas las personas:



```
    while(result.hasNext()){
        System.out.println(result.next());
    }

    db.close();
}
```

1. **while(result.hasNext()):**
    - Este es un bucle while que continúa ejecutándose mientras result.hasNext() devuelva true
    - **hasNext()** verifica si hay más elementos en el conjunto de resultados que aún no se han procesado
2. **System.out.println(result.next()):**
    - **result.next()** obtiene el siguiente objeto del conjunto de resultados
    - **System.out.println()** imprime ese objeto en la consola
3. **db.close():**
    - Esta línea es **MUY IMPORTANTE** ya que cierra la conexión con la base de datos
    - Es una buena práctica de programación siempre cerrar las conexiones a bases de datos

- Evita problemas de memoria y asegura que todos los cambios se guarden correctamente
- Si no se cierra la conexión, podría causar pérdida de datos o corrupción de la base de datos

1. **Declaración del método**

```
public static void crearPersona(){
    Persona p1 = new Persona("Fernando", 30, 1.73, 73);
    Persona p2 = new Persona("Pepe", 30, 1.75, 80);
    Persona p3 = new Persona("Alfredo", 20, 1.9, 90);
    Persona p4 = new Persona("Roberto", 35, 1.70, 70);
    Persona p5 = new Persona("Domingo", 30, 1.73, 80);

    File f = new File("personas.db4o");
```

- Es un método público y estático que no devuelve ningún valor (void)
- Se usa para crear y almacenar personas en la base de datos
- Crea 5 objetos Persona diferentes con sus respectivos datos

2. **Creación del archivo de base de datos:**

```
File f = new File("personas.db4o");
```

- Crea una referencia al archivo de base de datos DB4O

3. Apertura de la conexión:

```
ObjectContainer db = Db4oEmbedded.openFile(f.getAbsolutePath());
```

- Abre una conexión con la base de datos usando la ruta absoluta del archivo

4. Almacenamiento de los objetos:

```
db.store(p1);
db.store(p2);
db.store(p3);
db.store(p4);
db.store(p5);
```

- Guarda cada objeto Persona en la base de datos
- El método **store()** persiste los objetos en la base de datos

5. Cierre de la conexión:

```
db.close();
}
```

- Cierra la conexión con la base de datos
- Es importante para guardar los cambios y liberar recurso