## Índice de la clase JDBC

1. **Introducción a las API de Conexión a Bases de Datos** 
   * ¿Qué es una API de conexión a bases de datos?
   * JDBC como estándar para bases de datos relacionales.
   * Características principales de JDBC.
   * Ejemplo básico de flujo de trabajo JDBC.
2. **El Desfase Objeto-Relacional** 
   * ¿Qué es el desfase objeto-relacional?
   * Cómo afectan las diferencias entre el modelo de objetos y el modelo relacional.
   * Herramientas y estrategias para minimizar el desfase objeto-relacional (ORM y JDBC).
3. **Conexión a una Base de Datos con JDBC** 
   * Concepto de driver JDBC.
   * Tipos de drivers JDBC (Type 1-4).
   * Configuración de un driver JDBC.
   * Establecer una conexión con la base de datos:
     + Código básico para conectarse a una base de datos.
4. **Ejecución de Sentencias SQL en Java con JDBC**
   * Uso de la clase Statement para ejecutar sentencias SQL.
   * Tipos de sentencias: SELECT, INSERT, UPDATE, DELETE.
   * Ejemplo de ejecución de sentencias SQL en Java:
     + Conectar a la base de datos.
     + Ejecutar una consulta y mostrar los resultados.
     + Insertar, actualizar y eliminar registros.
5. **Sentencias Preparadas (PreparedStatements)** 
   * ¿Qué son las sentencias preparadas?
   * Ventajas de las sentencias preparadas:
     + Seguridad frente a ataques de inyección SQL.
     + Eficiencia y reutilización.
   * Ejemplo de uso de PreparedStatement.
6. **Manejo de Transacciones con JDBC** 
   * Concepto de transacción.
   * Uso de transacciones para garantizar la atomicidad de las operaciones.
   * Commit y rollback en JDBC.
   * Ejemplo de transacciones con JDBC:
     + Crear una transacción.
     + Ejecutar varias sentencias dentro de la transacción.
     + Confirmar o deshacer cambios.
7. **Conceptos clave** 
   * **Clave autogenerada**: cómo generar claves primarias automáticamente en bases de datos y cómo obtenerlas en JDBC.
   * **Conector**: intermediario entre una aplicación y la base de datos, proporcionando acceso mediante una API como JDBC.
   * **Desfase objeto-relacional**: diferencia entre el modelo de objetos en Java y el modelo relacional de bases de datos.
   * **Driver de JDBC**: pieza de software que permite la comunicación entre Java y una base de datos.
   * **Iterador**: mecanismo para recorrer y procesar los resultados de una consulta SQL (ResultSet en JDBC).
   * **JDBC**: Java Database Connectivity, la API de Java para conectarse y operar con bases de datos relacionales.

## Desarrollo de la Clase

### 1. Introducción a las API de Conexión a Bases de Datos

* Una **API (Application Programming Interface)** es un conjunto de reglas y protocolos que permite que diferentes aplicaciones de software se comuniquen entre sí. Esencialmente, una API define cómo las aplicaciones pueden interactuar, compartir datos y realizar acciones entre ellas, facilitando la integración y automatización de procesos entre sistemas.

### Funciones clave de una API:

1. **Interfaz**: Proporciona una serie de funciones que otros programas pueden utilizar para interactuar con un sistema o servicio.
2. **Abstracción**: Permite a los desarrolladores usar funciones de un sistema sin tener que conocer su implementación interna.
3. **Interoperabilidad**: Facilita la comunicación entre diferentes aplicaciones y plataformas, permitiendo la transferencia de datos y funciones.

### Ejemplos:

* Una API de un servicio de mapas, como Google Maps API, puede permitir que una aplicación de transporte acceda a mapas y funcionalidades de rutas sin tener que desarrollar su propio sistema de mapas.
* **JDBC (Java Database Connectivity)**: es la API estándar de Java para acceder a bases de datos relacionales. Funciona con cualquier base de datos que tenga un driver JDBC, como MySQL, PostgreSQL o Oracle.
* **Características principales de JDBC**:
  + Conexión a bases de datos relacionales.
  + Ejecución de sentencias SQL.
  + Recuperación y manipulación de datos.
  + Gestión de transacciones.

**Código Ejemplo**: Flujo de trabajo básico JDBC

Connection con = DriverManager.getConnection(url, usuario, contraseña);

Statement stmt = con.createStatement();

ResultSet rs = stmt.executeQuery("SELECT \* FROM tabla");

while (rs.next()) {

System.out.println(rs.getString(1)); // Procesa el resultado

}

*Explicación del código:*

**Connection con = DriverManager.getConnection(url, usuario, contraseña);**

Establece una conexión a la base de datos utilizando la clase DriverManager con los parámetros url, usuario, y contraseña. La url especifica la dirección de la base de datos, y el usuario y la contraseña son las credenciales para acceder a ella.

El objeto Connection con representa la conexión activa.

Statement stmt = con.createStatement();

Crea un objeto Statement que se utiliza para ejecutar consultas SQL en la base de datos.

stmt es el objeto que lanzará las consultas SQL y gestionará los resultados.

ResultSet rs = stmt.executeQuery("SELECT \* FROM tabla");

Ejecuta la consulta SQL "SELECT \* FROM tabla" en la base de datos y obtiene un ResultSet que contiene los resultados.

"SELECT \* FROM tabla" es una consulta que selecciona todas las filas y columnas de la tabla especificada (tabla).

while (rs.next()) { System.out.println(rs.getString(1)); }

Este bucle while recorre cada fila del ResultSet.

rs.next() mueve el cursor al siguiente registro y devuelve true si hay más filas.

rs.getString(1) obtiene el valor de la primera columna de la fila actual como una cadena de texto y lo imprime. Esto permite ver el contenido de cada fila en la primera columna de la tabla.

Un ejemplo de una **URL de conexión** que puedes utilizar si tienes instalado un servidor de **MySQL** localmente en tu máquina:

String url = "jdbc:mysql://localhost:3306/mi\_base\_de\_datos";

String usuario = "root";

String contraseña = "tu\_contraseña";

### Explicación de la URL:

* jdbc:mysql://: Especifica el protocolo JDBC y el tipo de base de datos (MySQL).
* localhost:3306: localhost indica que el servidor está en la máquina local, y 3306 es el puerto predeterminado de MySQL.
* mi\_base\_de\_datos: El nombre de la base de datos a la que deseas conectarte. Debe ser una base de datos existente en tu servidor MySQL.

Si tienes MySQL en un servidor remoto, cambia localhost por la **dirección IP o el nombre de dominio** del servidor. Además, asegúrate de que el usuario y la contraseña sean válidos para acceder a esa base de datos.

### 2. El Desfase Objeto-Relacional

* El **desfase objeto-relacional** surge porque el mundo de los objetos en programación (Java) es diferente al modelo relacional de las bases de datos.
* En Java, trabajamos con **objetos** mientras que las bases de datos usan **tablas** y relaciones.
* Ejemplo: Las bases de datos no tienen conceptos de herencia, y manejar estructuras complejas de objetos como listas dentro de una tabla es complicado.
* **ORM (Object-Relational Mapping)**, como Hibernate, es una solución para este desfase, pero en JDBC el desarrollador gestiona esta discrepancia manualmente.
* El **desfase objeto-relacional** (también conocido como "impedance mismatch" en inglés) se refiere a las diferencias entre el paradigma de programación orientado a objetos, utilizado en lenguajes como Java, y el modelo relacional que siguen las bases de datos tradicionales. Este desfase surge cuando intentas mapear objetos de un lenguaje de programación a las estructuras de una base de datos relacional, que trabaja con tablas, filas y columnas.

### Principales causas del desfase objeto-relacional:

1. **Estructuras de datos diferentes**:
   * En la programación orientada a objetos, los datos se organizan en clases con atributos y métodos que encapsulan tanto el estado como el comportamiento.
   * En las bases de datos relacionales, los datos se organizan en tablas con filas y columnas, donde cada fila representa un registro y cada columna un atributo.
   * El desafío es cómo convertir objetos complejos en filas y columnas de una tabla relacional, ya que los objetos pueden tener herencias, relaciones de agregación o composición que no encajan directamente en el modelo relacional.
2. **Relaciones entre objetos vs relaciones entre tablas**:
   * En el modelo orientado a objetos, las relaciones entre objetos pueden ser de herencia, composición o asociación. Estas relaciones son flexibles y dinámicas.
   * En bases de datos relacionales, las relaciones se representan mediante **claves foráneas** que conectan una tabla con otra, y no soportan de manera nativa conceptos como la herencia.
3. **Persistencia de objetos**:
   * En un sistema orientado a objetos, los objetos viven en memoria y desaparecen cuando el programa finaliza.
   * En bases de datos relacionales, los datos persisten de manera duradera en disco. La necesidad de convertir (o mapear) los objetos en tablas y luego volver a reconstruirlos para usarlos en memoria es un desafío.

### Ejemplos del desfase:

* **Herencia**: Si en Java tienes una clase Persona y una clase Empleado que hereda de Persona, en el mundo relacional no hay una forma directa de representar la herencia. Debes decidir si mapeas ambas clases en una sola tabla, o si creas dos tablas con claves foráneas para conectar las entidades.
* **Colecciones**: Un atributo de una clase puede ser una colección de objetos (como una lista o un conjunto). Sin embargo, las bases de datos no tienen un tipo de dato que represente directamente una colección, por lo que debes usar varias tablas con relaciones uno-a-muchos o muchos-a-muchos.

### Soluciones para el desfase objeto-relacional:

1. **Mapeo Objeto-Relacional (ORM)**:
   * Las herramientas ORM como **Hibernate** en Java intentan resolver este desfase proporcionando una capa de abstracción entre el código orientado a objetos y la base de datos relacional. Estas herramientas generan automáticamente el SQL necesario para mapear los objetos en tablas, facilitando la persistencia y recuperación de los datos.
2. **JDBC**:
   * Aunque JDBC no es una solución ORM, permite a los desarrolladores interactuar directamente con las bases de datos mediante SQL. Con JDBC, el desarrollador es responsable de gestionar el mapeo entre objetos y tablas, lo que requiere más trabajo manual, pero ofrece mayor control sobre las consultas.

### Conclusión:

El desfase objeto-relacional es un desafío que surge de las diferencias fundamentales entre cómo los datos se manejan en la programación orientada a objetos y en las bases de datos relacionales. Aunque existen herramientas como ORM para ayudar a reducir esta brecha, siempre será importante entender las diferencias para optimizar la interacción entre ambos mundos.

### 3. Conexión a una Base de Datos con JDBC

* **Driver de JDBC**: El puente entre Java y la base de datos. Permite que tu aplicación Java se conecte a una base de datos relacional.
* **Tipos de drivers**: Tipo 1 a 4, siendo el tipo 4 el más común actualmente (drivers puros en Java).

**Ejemplo de conexión**:

String url = "jdbc:mysql://localhost:3306/miBaseDatos";

String usuario = "root";

String contraseña = "password";

Connection con = DriverManager.getConnection(url, usuario, contraseña);

La **conexión a una base de datos con JDBC (Java Database Connectivity)** es uno de los pasos fundamentales para trabajar con bases de datos desde una aplicación Java. JDBC es una API que permite a los desarrolladores conectarse, ejecutar consultas y actualizar bases de datos relacionales utilizando SQL, proporcionando un puente entre el mundo orientado a objetos de Java y el modelo relacional de las bases de datos.

### Pasos para establecer una conexión con JDBC:

1. **Cargar el Driver JDBC**:
   * Antes de poder conectarse a una base de datos, es necesario cargar el driver JDBC que permitirá la comunicación entre Java y la base de datos específica.
   * Con JDBC 4.0 y posteriores, la carga del driver se hace automáticamente si está en el classpath, pero en versiones anteriores, se hacía manualmente con Class.forName().

Ejemplo:

Class.forName("com.mysql.cj.jdbc.Driver"); // MySQL JDBC Driver

1. **Establecer la conexión**:
   * Una vez que el driver está disponible, se puede establecer una conexión con la base de datos usando el método getConnection() de la clase DriverManager.
   * Se necesita la URL de la base de datos, junto con un nombre de usuario y una contraseña si es necesario.

Ejemplo:

String url = "jdbc:mysql://localhost:3306/miBaseDeDatos";

String usuario = "root";

String contraseña = "password";

Connection con = DriverManager.getConnection(url, usuario, contraseña);

* + La **URL** sigue el formato: jdbc:subprotocol://host:port/database.
    - jdbc: Indica que se usa JDBC.
    - subprotocol: Depende del sistema de base de datos, como mysql, postgresql, oracle.
    - host: El servidor donde se encuentra la base de datos (puede ser localhost si está en la misma máquina).
    - port: El puerto en el que escucha la base de datos (por defecto es 3306 para MySQL).
    - database: El nombre de la base de datos a la que deseas conectarte.

1. **Uso del objeto** Connection:
   * Una vez establecida la conexión, puedes utilizar el objeto Connection para crear declaraciones SQL y ejecutarlas.
   * Se pueden realizar múltiples operaciones utilizando la misma conexión, y una vez que hayas terminado de interactuar con la base de datos, debes cerrarla para liberar los recursos.

Ejemplo:

con.close(); // Cierra la conexión

1. **Control de excepciones**:
   * El manejo de excepciones es crucial cuando se trabaja con JDBC. La conexión a la base de datos puede fallar por diversas razones (credenciales incorrectas, servidor caído, etc.), y el código debe ser capaz de gestionar estas excepciones.
   * Las excepciones relacionadas con JDBC generalmente son de tipo SQLException.

Ejemplo:

try {

Connection con = DriverManager.getConnection(url, usuario, contraseña);

// Operaciones con la base de datos

} catch (SQLException e) {

e.printStackTrace();

}

### Conceptos adicionales:

1. **Pooled Connections (Conexiones en Pool)**:
   * En aplicaciones empresariales de gran escala, es costoso crear y destruir conexiones de base de datos con frecuencia. Para optimizar esto, se usa un **pool de conexiones** que permite reutilizar conexiones abiertas.
2. **Drivers JDBC**:
   * Existen cuatro tipos de drivers JDBC:
     + **Driver Tipo 1 (Bridge Driver)**: Un puente entre JDBC y ODBC. No es muy eficiente y rara vez se usa.
     + **Driver Tipo 2 (Native API Driver)**: Usa APIs nativas específicas del sistema. Más eficiente pero depende del sistema operativo.
     + **Driver Tipo 3 (Network Protocol Driver)**: Se conecta a través de un middleware que traduce las llamadas JDBC.
     + **Driver Tipo 4 (Pure Java Driver)**: Es el más común y eficiente, ya que traduce las llamadas JDBC directamente a las llamadas de protocolo de la base de datos. No requiere instalación de software adicional.
3. **Transacciones**:
   * JDBC soporta transacciones, lo que permite agrupar varias operaciones SQL para que se ejecuten de manera atómica (todas o ninguna).
   * Se puede controlar manualmente cuándo hacer **commit** (confirmar cambios) o **rollback** (deshacer cambios) con el método setAutoCommit(false) en la conexión.

Ejemplo:

con.setAutoCommit(false);

// Operaciones SQL

con.commit(); // Confirma los cambios

### Ejemplo completo de conexión y consulta con JDBC:

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.ResultSet;

import java.sql.SQLException;

import java.sql.Statement;

public class ConectarBaseDeDatos {

public static void main(String[] args) {

String url = "jdbc:mysql://localhost:3306/miBaseDeDatos";

String usuario = "root";

String contraseña = "password";

try (Connection con = DriverManager.getConnection(url, usuario, contraseña);

Statement stmt = con.createStatement()) {

String consulta = "SELECT \* FROM productos";

ResultSet rs = stmt.executeQuery(consulta);

while (rs.next()) {

System.out.println("Producto: " + rs.getString("nombre"));

}

} catch (SQLException e) {

e.printStackTrace();

}

}

}

En resumen, **JDBC** es una API poderosa y flexible para conectarse a bases de datos desde Java, que proporciona todas las herramientas necesarias para interactuar con bases de datos relacionales.

### 4. Ejecución de Sentencias SQL en Java con JDBC

* Usamos la interfaz **Statement** para ejecutar sentencias SQL:
  + **SELECT** para recuperar datos.
  + **INSERT**, **UPDATE**, **DELETE** para modificar datos.

**Ejemplo**: Ejecutar una consulta

Statement stmt = con.createStatement();

ResultSet rs = stmt.executeQuery("SELECT \* FROM productos");

while (rs.next()) {

System.out.println(rs.getString("nombre\_producto"));

}

La **ejecución de sentencias SQL en Java con JDBC** es una de las tareas fundamentales al interactuar con bases de datos. JDBC proporciona las herramientas necesarias para ejecutar sentencias SQL como SELECT, INSERT, UPDATE y DELETE directamente desde un programa Java, utilizando objetos como Statement, PreparedStatement y CallableStatement.

### Tipos de Sentencias en JDBC:

1. **Statement**:
   * Es la clase más básica que se usa para ejecutar sentencias SQL simples y directas. Un Statement se utiliza cuando no es necesario pasar parámetros a la consulta SQL.
   * **Ejemplo**: Ejecutar una consulta SQL para obtener todos los productos de una tabla.

Statement stmt = con.createStatement();

ResultSet rs = stmt.executeQuery("SELECT \* FROM productos");

while (rs.next()) {

System.out.println(rs.getString("nombre\_producto"));

}

* + **Ventajas**: Sencillo de usar.
  + **Desventajas**: No es seguro frente a ataques de inyección SQL si los datos de usuario se incluyen directamente en la consulta.

1. **PreparedStatement**:
   * Esta clase es una mejora sobre Statement y permite ejecutar sentencias SQL precompiladas, lo que mejora la eficiencia cuando una misma consulta se ejecuta varias veces.
   * Lo más importante es que los **PreparedStatements** son seguros frente a **inyecciones SQL**, ya que los parámetros se pasan de forma segura y no directamente en la cadena SQL.

**Ejemplo**:

PreparedStatement pstmt = con.prepareStatement("SELECT \* FROM productos WHERE precio > ?");

pstmt.setDouble(1, 50.0); // Establece el parámetro

ResultSet rs = pstmt.executeQuery();

while (rs.next()) {

System.out.println(rs.getString("nombre\_producto"));

}

* + **Ventajas**: Seguridad frente a inyección SQL y mejora de rendimiento con consultas repetidas.
  + **Desventajas**: Ligeramente más complejo de usar que Statement.

1. **CallableStatement**:
   * Se utiliza para ejecutar procedimientos almacenados en la base de datos, que son bloques de código SQL almacenados en el servidor de la base de datos.
   * Los **procedimientos almacenados** son útiles para realizar operaciones complejas en el servidor sin necesidad de pasar todo el código SQL desde la aplicación.

**Ejemplo**:

CallableStatement cstmt = con.prepareCall("{CALL obtener\_productos(?, ?)}");

cstmt.setInt(1, 50);

cstmt.registerOutParameter(2, Types.INTEGER);

cstmt.execute();

int resultado = cstmt.getInt(2);

* + **Ventajas**: Ideal para operaciones complejas y para reducir el tráfico de red.
  + **Desventajas**: Depende de que el procedimiento esté predefinido en la base de datos.

### ResultSet: Obteniendo Resultados de las Consultas

Cuando se ejecutan consultas SELECT con JDBC, el resultado se devuelve en un **ResultSet**, que es un conjunto de filas que puedes recorrer para obtener los datos.

* **Navegación por el ResultSet**: Utiliza el método next() para moverte de una fila a la siguiente, y métodos como getString(), getInt(), etc., para obtener los valores de las columnas.

ResultSet rs = stmt.executeQuery("SELECT \* FROM productos");

while (rs.next()) {

System.out.println(rs.getString("nombre\_producto"));

System.out.println(rs.getDouble("precio"));

}

* **Tipos de ResultSet**: Hay diferentes tipos de ResultSet, como:
  + **TYPE\_FORWARD\_ONLY** (por defecto): Solo permite avanzar una fila a la vez.
  + **TYPE\_SCROLL\_INSENSITIVE**: Permite moverse hacia adelante y hacia atrás en el ResultSet.
  + **TYPE\_SCROLL\_SENSITIVE**: Igual que el anterior, pero sensible a cambios realizados en la base de datos mientras se navega por el conjunto de resultados.

### Actualización de Datos con JDBC:

Las sentencias **INSERT**, **UPDATE** y **DELETE** también se ejecutan usando Statement o PreparedStatement, pero en lugar de devolver un ResultSet, se devuelve un valor entero que indica cuántas filas fueron afectadas.

PreparedStatement pstmt = con.prepareStatement("UPDATE productos SET precio = ? WHERE id = ?");

pstmt.setDouble(1, 99.99);

pstmt.setInt(2, 1);

int filasAfectadas = pstmt.executeUpdate();

System.out.println("Filas actualizadas: " + filasAfectadas);

### Transacciones con JDBC:

En JDBC, puedes manejar transacciones de forma manual si configuras la conexión en **modo manual** (desactivando el auto-commit). Esto te permite agrupar múltiples operaciones SQL dentro de una misma transacción, y hacer que se confirmen o se deshagan como un todo.

* **Inicio de la transacción**:

con.setAutoCommit(false); // Desactiva el auto-commit

* **Commit y rollback**:

try {

// Ejecutar varias operaciones SQL

con.commit(); // Confirmar la transacción

} catch (SQLException e) {

con.rollback(); // Deshacer los cambios si hay un error

}

### Buenas Prácticas:

1. **Cerrar recursos**: Siempre se deben cerrar los recursos (conexiones, statements, result sets) para evitar fugas de memoria. Puedes usar bloques try-with-resources para gestionar esto automáticamente:

try (Connection con = DriverManager.getConnection(url, usuario, contraseña);

Statement stmt = con.createStatement()) {

// Operaciones SQL

} catch (SQLException e) {

e.printStackTrace();

}

1. **Uso de PreparedStatements**: Siempre que sea posible, utiliza **PreparedStatements** para evitar vulnerabilidades de inyección SQL y mejorar el rendimiento en consultas repetidas.

### Conclusión:

JDBC ofrece una API flexible y poderosa para interactuar con bases de datos desde Java, permitiendo ejecutar desde consultas simples hasta transacciones complejas. Aunque el uso de Statement es sencillo, PreparedStatement y CallableStatement ofrecen más seguridad y eficiencia, especialmente en aplicaciones con acceso frecuente a bases de datos.

### 5. Sentencias Preparadas (PreparedStatements)

* **PreparedStatement**: mejora las consultas SQL al evitar ataques de inyección SQL y mejorar la eficiencia de las consultas repetidas.

**Ejemplo**:

PreparedStatement pstmt = con.prepareStatement("INSERT INTO usuarios (nombre, email) VALUES (?, ?)");

pstmt.setString(1, "Juan");

pstmt.setString(2, "juan@gmail.com");

pstmt.executeUpdate();

Las **Sentencias Preparadas (PreparedStatements)** en JDBC son una herramienta clave para ejecutar consultas SQL de manera más eficiente y segura. A diferencia de las sentencias básicas (Statement), las **PreparedStatements** permiten precompilar las consultas y reutilizarlas con diferentes valores, lo que mejora el rendimiento, especialmente en operaciones repetitivas. Además, las **PreparedStatements** previenen vulnerabilidades como la **inyección de SQL**.

### Características principales de las Sentencias Preparadas:

1. **Precompilación**:
   * Las sentencias SQL que se usan en un **PreparedStatement** son precompiladas por la base de datos, lo que significa que el servidor puede optimizar la ejecución de las consultas cuando se ejecutan varias veces con diferentes parámetros.
   * Esto reduce el tiempo de procesamiento en la base de datos, especialmente en operaciones repetitivas como las inserciones masivas o las actualizaciones frecuentes.

**Ejemplo de código**:

String sql = "INSERT INTO usuarios (nombre, email) VALUES (?, ?)";

PreparedStatement pstmt = con.prepareStatement(sql);

pstmt.setString(1, "Juan");

pstmt.setString(2, "juan@example.com");

pstmt.executeUpdate();

1. **Protección contra inyección SQL**:
   * Una de las mayores ventajas de usar **PreparedStatements** es que los valores de los parámetros se procesan de manera separada a la consulta SQL. Esto significa que los datos del usuario no se combinan directamente con la consulta, lo que impide que se introduzcan comandos maliciosos.
   * Esto soluciona el problema de la **inyección SQL**, donde un atacante puede manipular las consultas al incluir código SQL en los datos de entrada.

**Ejemplo de vulnerabilidad de inyección SQL (sin PreparedStatement)**:

String sql = "SELECT \* FROM usuarios WHERE nombre = '" + nombreUsuario + "'";

Statement stmt = con.createStatement();

ResultSet rs = stmt.executeQuery(sql);

Si nombreUsuario contiene algo como "' OR '1'='1", la consulta SQL sería manipulada para devolver todos los registros. En cambio, con **PreparedStatements**, este tipo de ataque se mitiga:

String sql = "SELECT \* FROM usuarios WHERE nombre = ?";

PreparedStatement pstmt = con.prepareStatement(sql);

pstmt.setString(1, nombreUsuario); // El valor es procesado de forma segura

ResultSet rs = pstmt.executeQuery();

1. **Reutilización y eficiencia**:
   * Las **PreparedStatements** pueden reutilizarse varias veces con diferentes valores, lo que las hace ideales para operaciones repetitivas como las inserciones masivas.
   * La precompilación de la consulta y el uso de parámetros en lugar de concatenación de cadenas de texto mejora el rendimiento y la escalabilidad de la aplicación.
2. **Parámetros en las sentencias**:
   * Los valores en una **PreparedStatement** se especifican utilizando marcadores de posición (?) dentro de la consulta SQL. Estos marcadores de posición son luego reemplazados con los valores correspondientes mediante métodos como setString(), setInt(), setDouble(), etc.
   * El uso de estos marcadores hace que el código sea más claro y facilita la separación entre la lógica de la consulta y los datos.

**Ejemplo con varios parámetros**:

String sql = "UPDATE productos SET precio = ? WHERE id = ?";

PreparedStatement pstmt = con.prepareStatement(sql);

pstmt.setDouble(1, 19.99); // Establece el nuevo precio

pstmt.setInt(2, 101); // Establece el ID del producto

int filasActualizadas = pstmt.executeUpdate();

1. **Trabajando con grandes cantidades de datos**:
   * Las **PreparedStatements** son útiles para trabajar con **Batch Processing** (procesamiento por lotes). JDBC permite ejecutar varias sentencias SQL en un solo lote para reducir el número de llamadas al servidor de base de datos.
   * Esto es especialmente útil cuando necesitas insertar o actualizar grandes cantidades de datos, ya que puedes ejecutar múltiples operaciones a la vez sin tener que enviar una consulta individual para cada fila.

**Ejemplo de procesamiento por lotes**:

PreparedStatement pstmt = con.prepareStatement("INSERT INTO productos (nombre, precio) VALUES (?, ?)");

for (int i = 0; i < 1000; i++) {

pstmt.setString(1, "Producto " + i);

pstmt.setDouble(2, i \* 10.0);

pstmt.addBatch(); // Añadir al lote

}

pstmt.executeBatch(); // Ejecuta todas las inserciones en un solo lote

### Ventajas del uso de PreparedStatements:

1. **Seguridad**: Los parámetros se tratan de manera segura, protegiendo la aplicación contra inyecciones de SQL.
2. **Eficiencia**: Las consultas se precompilan, lo que reduce el tiempo de ejecución en consultas repetitivas.
3. **Claridad**: El uso de parámetros y marcadores de posición mejora la legibilidad y el mantenimiento del código.
4. **Procesamiento por lotes**: Facilita la inserción o actualización de grandes volúmenes de datos de manera eficiente.

### Buenas Prácticas:

* **Cerrar siempre los recursos**: Es importante cerrar las instancias de PreparedStatement y ResultSet para liberar los recursos de la base de datos.

try (PreparedStatement pstmt = con.prepareStatement(sql)) {

// Código para ejecutar la consulta

} catch (SQLException e) {

e.printStackTrace();

}

* **Usar** PreparedStatement **en lugar de** Statement siempre que haya datos de usuario involucrados en las consultas, para garantizar seguridad y evitar inyección SQL.

### Conclusión:

Las **Sentencias Preparadas (PreparedStatements)** son una herramienta esencial en JDBC para escribir consultas SQL eficientes, reutilizables y seguras. Ofrecen un nivel de protección importante contra ataques de inyección SQL y optimizan la ejecución de consultas repetitivas o con grandes volúmenes de datos. Por estas razones, es recomendable preferir **PreparedStatements** sobre el uso de Statement en casi todos los casos, especialmente cuando se manejen entradas de usuario.

### 6. Manejo de Transacciones con JDBC

* **Transacciones**: Permiten agrupar varias operaciones SQL que deben ejecutarse de manera atómica (todas o ninguna).

**Ejemplo**:

con.setAutoCommit(false);

try {

stmt.executeUpdate("INSERT INTO pedidos (id\_cliente, total) VALUES (1, 100)");

stmt.executeUpdate("INSERT INTO detalles\_pedido (id\_pedido, producto) VALUES (1, 'Laptop')");

con.commit(); // Confirma la transacción

} catch (SQLException e) {

con.rollback(); // Deshace los cambios en caso de error

}

El **manejo de transacciones** en JDBC es crucial para asegurar que múltiples operaciones SQL se ejecuten de manera atómica, es decir, todas o ninguna. Las transacciones permiten agrupar varias sentencias SQL en una sola unidad de trabajo, lo que es importante para garantizar la consistencia de los datos en la base de datos, especialmente en sistemas críticos donde múltiples usuarios o procesos pueden estar modificando datos al mismo tiempo.

### ¿Qué es una transacción?

Una **transacción** es un conjunto de operaciones que deben ejecutarse en conjunto. Si una de ellas falla, todas las demás deben revertirse para mantener la integridad de la base de datos. Esto se controla a través de las propiedades **ACID** (Atomicidad, Consistencia, Aislamiento y Durabilidad):

1. **Atomicidad**: Todas las operaciones de la transacción deben completarse con éxito. Si alguna falla, ninguna de las operaciones tendrá efecto.
2. **Consistencia**: Una transacción debe llevar la base de datos de un estado válido a otro estado válido.
3. **Aislamiento**: Los cambios realizados por una transacción no deben ser visibles para otras transacciones hasta que se complete.
4. **Durabilidad**: Una vez que una transacción se completa con éxito, sus efectos deben ser permanentes, incluso si ocurre una falla en el sistema.

### Modo de transacción en JDBC

Por defecto, JDBC está en **auto-commit**. Esto significa que cada instrucción SQL que ejecutas se confirma inmediatamente después de su ejecución. Si deseas agrupar varias operaciones en una transacción, debes desactivar el auto-commit.

#### Desactivando el auto-commit:

Connection con = DriverManager.getConnection(url, usuario, contraseña);

con.setAutoCommit(false); // Desactiva el auto-commit

Esto permite que varias operaciones SQL se realicen en una misma transacción.

#### Ejecutando operaciones en una transacción:

Después de desactivar el auto-commit, puedes ejecutar varias operaciones SQL y luego usar **commit** o **rollback** para confirmar o deshacer los cambios.

#### Ejemplo de manejo de transacciones:

try {

con.setAutoCommit(false); // Inicia la transacción

// Ejecución de la primera sentencia SQL

Statement stmt = con.createStatement();

stmt.executeUpdate("INSERT INTO pedidos (id\_cliente, total) VALUES (1, 100)");

// Ejecución de la segunda sentencia SQL

stmt.executeUpdate("INSERT INTO detalles\_pedido (id\_pedido, producto) VALUES (1, 'Laptop')");

// Si ambas sentencias se ejecutan correctamente, confirma la transacción

con.commit();

System.out.println("Transacción completada con éxito.");

} catch (SQLException e) {

// Si ocurre un error, deshacer los cambios

con.rollback();

System.err.println("Transacción fallida. Se han revertido los cambios.");

} finally {

con.setAutoCommit(true); // Restablecer el auto-commit

}

### Importancia del ****Commit**** y ****Rollback****:

* **Commit**: Si todas las operaciones en la transacción se ejecutan correctamente, se llama a commit(), lo que hace que los cambios sean permanentes en la base de datos.
* **Rollback**: Si ocurre algún error durante la ejecución de la transacción, puedes llamar a rollback(), lo que deshará todos los cambios realizados hasta ese momento en la transacción.

### Manejo de transacciones en sistemas complejos

En aplicaciones más avanzadas, como sistemas bancarios o sistemas de ventas en línea, el manejo de transacciones es crítico para garantizar que las operaciones críticas, como transferencias de dinero o actualizaciones de inventario, se realicen correctamente. Las transacciones aseguran que los datos permanezcan consistentes incluso si una operación intermedia falla, evitando errores como la duplicación de registros o inconsistencias entre tablas relacionadas.

### Aislamiento de transacciones

El aislamiento de transacciones es importante cuando varias transacciones se ejecutan de forma concurrente. JDBC soporta varios niveles de aislamiento, que puedes configurar para evitar problemas como:

* **Lectura sucia**: Cuando una transacción ve cambios no confirmados de otra transacción.
* **Lectura no repetible**: Cuando una transacción ve cambios en los datos que se leyeron anteriormente dentro de la misma transacción.
* **Lectura fantasma**: Cuando una transacción ve filas nuevas que se insertaron por otra transacción después de que la transacción actual comenzó.

Puedes establecer el nivel de aislamiento con el siguiente método:

con.setTransactionIsolation(Connection.TRANSACTION\_SERIALIZABLE);

Los niveles de aislamiento disponibles son:

* **TRANSACTION\_READ\_UNCOMMITTED**: Permite leer datos no confirmados por otras transacciones (riesgo de lectura sucia).
* **TRANSACTION\_READ\_COMMITTED**: Solo permite leer datos confirmados (evita lectura sucia, pero puede tener problemas de lectura no repetible).
* **TRANSACTION\_REPEATABLE\_READ**: Evita la lectura no repetible (pueden ocurrir lecturas fantasma).
* **TRANSACTION\_SERIALIZABLE**: Es el nivel más alto de aislamiento y evita tanto lecturas no repetibles como fantasmas, pero es el más costoso en términos de rendimiento.

### Buenas prácticas en el manejo de transacciones:

1. **Desactivar auto-commit solo cuando sea necesario**: Mantén el auto-commit desactivado solo durante la transacción y reactívalo después para evitar problemas de concurrencia.
2. **Siempre utilizar commit o rollback**: Asegúrate de usar commit() para confirmar los cambios, y en caso de errores, usa rollback() para deshacer cualquier cambio no deseado.
3. **Tratar las excepciones adecuadamente**: Las excepciones de SQL pueden ocurrir durante cualquier operación de la transacción, y es importante capturarlas para poder reaccionar correctamente y proteger la integridad de los datos.
4. **Cerrar correctamente los recursos**: Siempre utiliza bloques try-with-resources o asegúrate de cerrar manualmente los recursos (conexión, sentencias, result sets) al finalizar una transacción.

### Conclusión:

El manejo de transacciones con JDBC es una habilidad esencial cuando se trabaja con bases de datos en aplicaciones críticas. Permite agrupar operaciones SQL de forma atómica, asegurando la consistencia y durabilidad de los datos. Usar transacciones correctamente también mejora la seguridad y previene errores comunes en sistemas con múltiples usuarios o procesos concurrentes que acceden a la base de datos.

### 7. Conceptos Clave

El **manejo de transacciones** en JDBC es crucial para asegurar que múltiples operaciones SQL se ejecuten de manera atómica, es decir, todas o ninguna. Las transacciones permiten agrupar varias sentencias SQL en una sola unidad de trabajo, lo que es importante para garantizar la consistencia de los datos en la base de datos, especialmente en sistemas críticos donde múltiples usuarios o procesos pueden estar modificando datos al mismo tiempo.

### ¿Qué es una transacción?

Una **transacción** es un conjunto de operaciones que deben ejecutarse en conjunto. Si una de ellas falla, todas las demás deben revertirse para mantener la integridad de la base de datos. Esto se controla a través de las propiedades **ACID** (Atomicidad, Consistencia, Aislamiento y Durabilidad):

1. **Atomicidad**: Todas las operaciones de la transacción deben completarse con éxito. Si alguna falla, ninguna de las operaciones tendrá efecto.
2. **Consistencia**: Una transacción debe llevar la base de datos de un estado válido a otro estado válido.
3. **Aislamiento**: Los cambios realizados por una transacción no deben ser visibles para otras transacciones hasta que se complete.
4. **Durabilidad**: Una vez que una transacción se completa con éxito, sus efectos deben ser permanentes, incluso si ocurre una falla en el sistema.

### Modo de transacción en JDBC

Por defecto, JDBC está en **auto-commit**. Esto significa que cada instrucción SQL que ejecutas se confirma inmediatamente después de su ejecución. Si deseas agrupar varias operaciones en una transacción, debes desactivar el auto-commit.

#### Desactivando el auto-commit:

Connection con = DriverManager.getConnection(url, usuario, contraseña);

con.setAutoCommit(false); // Desactiva el auto-commit

Esto permite que varias operaciones SQL se realicen en una misma transacción.

#### Ejecutando operaciones en una transacción:

Después de desactivar el auto-commit, puedes ejecutar varias operaciones SQL y luego usar **commit** o **rollback** para confirmar o deshacer los cambios.

#### Ejemplo de manejo de transacciones:

try {

con.setAutoCommit(false); // Inicia la transacción

// Ejecución de la primera sentencia SQL

Statement stmt = con.createStatement();

stmt.executeUpdate("INSERT INTO pedidos (id\_cliente, total) VALUES (1, 100)");

// Ejecución de la segunda sentencia SQL

stmt.executeUpdate("INSERT INTO detalles\_pedido (id\_pedido, producto) VALUES (1, 'Laptop')");

// Si ambas sentencias se ejecutan correctamente, confirma la transacción

con.commit();

System.out.println("Transacción completada con éxito.");

} catch (SQLException e) {

// Si ocurre un error, deshacer los cambios

con.rollback();

System.err.println("Transacción fallida. Se han revertido los cambios.");

} finally {

con.setAutoCommit(true); // Restablecer el auto-commit

}

### Importancia del ****Commit**** y ****Rollback****:

* **Commit**: Si todas las operaciones en la transacción se ejecutan correctamente, se llama a commit(), lo que hace que los cambios sean permanentes en la base de datos.
* **Rollback**: Si ocurre algún error durante la ejecución de la transacción, puedes llamar a rollback(), lo que deshará todos los cambios realizados hasta ese momento en la transacción.

### Manejo de transacciones en sistemas complejos

En aplicaciones más avanzadas, como sistemas bancarios o sistemas de ventas en línea, el manejo de transacciones es crítico para garantizar que las operaciones críticas, como transferencias de dinero o actualizaciones de inventario, se realicen correctamente. Las transacciones aseguran que los datos permanezcan consistentes incluso si una operación intermedia falla, evitando errores como la duplicación de registros o inconsistencias entre tablas relacionadas.

### Aislamiento de transacciones

El aislamiento de transacciones es importante cuando varias transacciones se ejecutan de forma concurrente. JDBC soporta varios niveles de aislamiento, que puedes configurar para evitar problemas como:

* **Lectura sucia**: Cuando una transacción ve cambios no confirmados de otra transacción.
* **Lectura no repetible**: Cuando una transacción ve cambios en los datos que se leyeron anteriormente dentro de la misma transacción.
* **Lectura fantasma**: Cuando una transacción ve filas nuevas que se insertaron por otra transacción después de que la transacción actual comenzó.

Puedes establecer el nivel de aislamiento con el siguiente método:

con.setTransactionIsolation(Connection.TRANSACTION\_SERIALIZABLE);

Los niveles de aislamiento disponibles son:

* **TRANSACTION\_READ\_UNCOMMITTED**: Permite leer datos no confirmados por otras transacciones (riesgo de lectura sucia).
* **TRANSACTION\_READ\_COMMITTED**: Solo permite leer datos confirmados (evita lectura sucia, pero puede tener problemas de lectura no repetible).
* **TRANSACTION\_REPEATABLE\_READ**: Evita la lectura no repetible (pueden ocurrir lecturas fantasma).
* **TRANSACTION\_SERIALIZABLE**: Es el nivel más alto de aislamiento y evita tanto lecturas no repetibles como fantasmas, pero es el más costoso en términos de rendimiento.

### Buenas prácticas en el manejo de transacciones:

1. **Desactivar auto-commit solo cuando sea necesario**: Mantén el auto-commit desactivado solo durante la transacción y reactívalo después para evitar problemas de concurrencia.
2. **Siempre utilizar commit o rollback**: Asegúrate de usar commit() para confirmar los cambios, y en caso de errores, usa rollback() para deshacer cualquier cambio no deseado.
3. **Tratar las excepciones adecuadamente**: Las excepciones de SQL pueden ocurrir durante cualquier operación de la transacción, y es importante capturarlas para poder reaccionar correctamente y proteger la integridad de los datos.
4. **Cerrar correctamente los recursos**: Siempre utiliza bloques try-with-resources o asegúrate de cerrar manualmente los recursos (conexión, sentencias, result sets) al finalizar una transacción.

### Conclusión:

El manejo de transacciones con JDBC es una habilidad esencial cuando se trabaja con bases de datos en aplicaciones críticas. Permite agrupar operaciones SQL de forma atómica, asegurando la consistencia y durabilidad de los datos. Usar transacciones correctamente también mejora la seguridad y previene errores comunes en sistemas con múltiples usuarios o procesos concurrentes que acceden a la base de datos.

4o

##### Tú dijiste:

cuentame algo mas de estos 6 conceptos clave: • Clave autogenerada: • Conector: • Desfase objeto-relacional: • Driver de JDBC: • Iterador: • JDBC:

###### ChatGPT Plus

ChatGPT

Aquí tienes una explicación más detallada de los seis conceptos clave relacionados con bases de datos y JDBC:

### 1. ****Clave autogenerada****:

Una **clave autogenerada** (o **clave primaria autoincremental**) es un campo de una tabla en una base de datos que genera automáticamente un valor único cada vez que se inserta un nuevo registro. Este valor se utiliza como identificador único del registro, sin necesidad de que el usuario lo proporcione manualmente.

* **Ejemplo en SQL (MySQL)**:

sql

CREATE TABLE usuarios (

id INT AUTO\_INCREMENT,

nombre VARCHAR(255),

PRIMARY KEY (id)

);

En JDBC, al insertar un registro con una clave autogenerada, puedes obtener ese valor utilizando Statement.RETURN\_GENERATED\_KEYS después de ejecutar una sentencia INSERT.

**Ejemplo en Java**:

PreparedStatement pstmt = con.prepareStatement("INSERT INTO usuarios (nombre) VALUES (?)", Statement.RETURN\_GENERATED\_KEYS);

pstmt.setString(1, "Juan");

pstmt.executeUpdate();

ResultSet rs = pstmt.getGeneratedKeys();

if (rs.next()) {

int idGenerado = rs.getInt(1);

System.out.println("ID generado: " + idGenerado);

}

### 2. ****Conector****:

Un **conector** es un componente de software que actúa como intermediario entre una aplicación y una base de datos. Permite que la aplicación se comunique con la base de datos a través de una API como JDBC. El conector traduce las llamadas de la API en comandos que la base de datos puede entender y ejecutar.

* **Ejemplo**: El conector JDBC de MySQL (MySQL Connector/J) es un ejemplo típico de conector que permite que las aplicaciones Java se conecten a bases de datos MySQL.

### 3. ****Desfase objeto-relacional****:

El **desfase objeto-relacional** (O/R impedance mismatch) se refiere a la diferencia entre el modelo de datos de un sistema orientado a objetos, como Java, y el modelo relacional de una base de datos.

* En los lenguajes de programación orientados a objetos, los datos se organizan en clases, objetos y relaciones entre estos objetos.
* En las bases de datos relacionales, los datos se organizan en tablas, filas y columnas.

Este desfase surge cuando intentas mapear clases y objetos a tablas y relaciones entre tablas. El desafío es cómo manejar conceptos como la herencia, las asociaciones o las composiciones dentro del modelo relacional. Una solución común es usar herramientas de **Mapeo Objeto-Relacional (ORM)**, como **Hibernate**, para facilitar la conversión entre los dos modelos.

### 4. ****Driver de JDBC****:

Un **driver JDBC** es una implementación específica de la interfaz JDBC que permite a Java conectarse a una base de datos particular. El driver traduce las llamadas JDBC hechas en Java en instrucciones SQL que la base de datos puede ejecutar.

* **Tipos de drivers JDBC**:
  1. **Type 1: JDBC-ODBC Bridge**: Usa ODBC como puente para conectarse a la base de datos (obsoleto en versiones recientes de Java).
  2. **Type 2: Native-API**: Usa bibliotecas nativas para conectarse a la base de datos, pero depende del sistema operativo.
  3. **Type 3: Network Protocol Driver**: Usa un middleware para traducir las llamadas JDBC a instrucciones SQL.
  4. **Type 4: Pure Java Driver**: Es el más común y eficiente, ya que traduce directamente las llamadas JDBC en el protocolo de la base de datos (ej., MySQL, PostgreSQL).
* **Ejemplo**: El driver JDBC de MySQL se carga automáticamente si está en el classpath de la aplicación, permitiendo la conexión a bases de datos MySQL.

### 5. ****Iterador****:

Un **iterador** es un objeto que permite recorrer una colección de elementos secuencialmente, sin necesidad de conocer la estructura interna de esa colección. En el contexto de JDBC, un **ResultSet** actúa como un iterador para recorrer los resultados de una consulta SQL.

* **Ejemplo**:

ResultSet rs = stmt.executeQuery("SELECT \* FROM productos");

while (rs.next()) {

System.out.println(rs.getString("nombre\_producto"));

}

Aquí, el método next() se usa para mover el cursor al siguiente registro en el ResultSet.

### 6. ****JDBC****:

**JDBC (Java Database Connectivity)** es una API de Java que permite a las aplicaciones conectarse y operar con bases de datos relacionales. JDBC proporciona un conjunto de clases e interfaces para interactuar con bases de datos mediante el uso de SQL desde programas Java.

* **Funciones principales de JDBC**:
  + Conectar a una base de datos.
  + Ejecutar consultas SQL (como SELECT, INSERT, UPDATE, DELETE).
  + Recuperar y procesar los resultados de las consultas (ResultSet).
  + Manejar transacciones para agrupar múltiples operaciones SQL en una sola operación atómica.

**Ejemplo básico de JDBC**:

Connection con = DriverManager.getConnection(url, usuario, contraseña);

Statement stmt = con.createStatement();

ResultSet rs = stmt.executeQuery("SELECT \* FROM usuarios");

while (rs.next()) {

System.out.println(rs.getString("nombre"));

}

con.close();

### Conclusión:

Cada uno de estos conceptos desempeña un papel clave en la conexión y manipulación de bases de datos desde aplicaciones Java mediante JDBC. Desde la creación de claves autogeneradas hasta el manejo del desfase objeto-relacional, estos fundamentos permiten a los desarrolladores construir aplicaciones robustas y eficientes con soporte para bases de datos relacionales.

**RESUMEN**

## . ****Introducción a las API de Conexión a Bases de Datos****

### Definición de API

Una **API** (Application Programming Interface) es un conjunto de reglas y protocolos que permiten que diferentes aplicaciones de software se comuniquen entre sí. Las API facilitan la integración y automatización de procesos entre sistemas.

### Funciones clave de una API:

* **Interfaz**: Permite que otros programas interactúen con un sistema o servicio.
* **Abstracción**: Facilita el uso de funciones sin conocer su implementación interna.
* **Interoperabilidad**: Permite la comunicación entre diferentes aplicaciones y plataformas.

### Ejemplo:

Un API de mapas, como Google Maps API, permite que una aplicación de transporte acceda a mapas sin necesidad de desarrollar un sistema de mapas propio.

### JDBC (Java Database Connectivity)

**JDBC** es la API estándar de Java para acceder a bases de datos relacionales, como MySQL, PostgreSQL u Oracle.

#### Características principales:

* Conexión a bases de datos relacionales.
* Ejecución de sentencias SQL.
* Recuperación y manipulación de datos.
* Gestión de transacciones.

**Ejemplo básico**:

Connection con = DriverManager.getConnection(url, usuario, contraseña);

Statement stmt = con.createStatement();

ResultSet rs = stmt.executeQuery("SELECT \* FROM tabla");

while (rs.next()) {

System.out.println(rs.getString(1)); // Procesa el resultado

}

## 2. ****El Desfase Objeto-Relacional****

El **desfase objeto-relacional** surge de las diferencias entre los objetos en la programación orientada a objetos y las tablas en las bases de datos relacionales.

### Principales causas:

1. **Estructuras de datos diferentes**: En los objetos se organizan en clases con atributos y métodos, mientras que en las bases de datos relacionales los datos se organizan en tablas.
2. **Relaciones entre objetos vs. tablas**: Las relaciones en objetos pueden ser de herencia, composición o asociación, mientras que en bases de datos, las relaciones se representan mediante claves foráneas.
3. **Persistencia de objetos**: Los objetos en la programación viven en memoria temporal, mientras que las bases de datos almacenan los datos de manera persistente.

### Ejemplo:

Una clase Persona en Java no se mapea directamente a una tabla en bases de datos. Puede ser necesario utilizar múltiples tablas o claves foráneas.

### Soluciones:

* **ORM (Object-Relational Mapping)**: Herramientas como Hibernate facilitan el mapeo de objetos a tablas de bases de datos.
* **JDBC**: Permite un control más detallado, pero requiere mayor esfuerzo manual.

## 3. ****Conexión a una Base de Datos con JDBC****

### Pasos para establecer una conexión con JDBC:

1. **Cargar el Driver JDBC**: Con JDBC 4.0 o posteriores, el driver se carga automáticamente. En versiones anteriores, se utiliza Class.forName().

**Ejemplo**:

Class.forName("com.mysql.cj.jdbc.Driver");

1. **Establecer la conexión**:

String url = "jdbc:mysql://localhost:3306/miBaseDeDatos";

Connection con = DriverManager.getConnection(url, usuario, contraseña);

1. **Uso del objeto** Connection: Una vez establecida la conexión, puedes ejecutar sentencias SQL y cerrar la conexión cuando termines.

con.close(); // Cierra la conexión

1. **Control de excepciones**: Las conexiones fallidas lanzan SQLException, que debe gestionarse.

try {

Connection con = DriverManager.getConnection(url, usuario, contraseña);

} catch (SQLException e) {

e.printStackTrace();

}

### Drivers JDBC:

Existen cuatro tipos de drivers JDBC:

1. **Type 1 (Bridge Driver)**: Un puente entre JDBC y ODBC.
2. **Type 2 (Native API Driver)**: Usa APIs nativas del sistema.
3. **Type 3 (Network Protocol Driver)**: Se conecta mediante un middleware.
4. **Type 4 (Pure Java Driver)**: Traduce directamente las llamadas JDBC al protocolo de la base de datos.

## 4. ****Ejecución de Sentencias SQL en Java con JDBC****

### Tipos de sentencias SQL en JDBC:

1. **Statement**: Se utiliza para ejecutar sentencias SQL simples.

Statement stmt = con.createStatement();

ResultSet rs = stmt.executeQuery("SELECT \* FROM productos");

1. **PreparedStatement**: Más eficiente y seguro frente a inyecciones SQL.

PreparedStatement pstmt = con.prepareStatement("SELECT \* FROM productos WHERE precio > ?");

pstmt.setDouble(1, 50.0);

ResultSet rs = pstmt.executeQuery();

1. **CallableStatement**: Se utiliza para ejecutar procedimientos almacenados en la base de datos.

## 5. ****Sentencias Preparadas (PreparedStatements)****

### Características:

* **Precompilación**: Las consultas SQL se precompilan, lo que mejora la eficiencia y permite reutilización.
* **Seguridad frente a inyección SQL**: Los parámetros se procesan de manera separada a la consulta.

**Ejemplo**:

PreparedStatement pstmt = con.prepareStatement("INSERT INTO usuarios (nombre, email) VALUES (?, ?)");

pstmt.setString(1, "Juan");

pstmt.setString(2, "juan@gmail.com");

pstmt.executeUpdate();

## 6. ****Manejo de Transacciones con JDBC****

Las **transacciones** en JDBC permiten agrupar múltiples operaciones SQL en una sola unidad atómica.

### Propiedades ACID:

1. **Atomicidad**: Todas las operaciones deben completarse con éxito.
2. **Consistencia**: Mantener la integridad de la base de datos.
3. **Aislamiento**: Los cambios no deben ser visibles hasta que la transacción se confirme.
4. **Durabilidad**: Los cambios persisten incluso después de fallos.

### Ejemplo de manejo de transacciones:

con.setAutoCommit(false); // Inicia la transacción

try {

Statement stmt = con.createStatement();

stmt.executeUpdate("INSERT INTO pedidos (id\_cliente, total) VALUES (1, 100)");

stmt.executeUpdate("INSERT INTO detalles\_pedido (id\_pedido, producto) VALUES (1, 'Laptop')");

con.commit(); // Confirma la transacción

} catch (SQLException e) {

con.rollback(); // Deshace los cambios si hay un error

}

## 7. ****Conceptos Clave****

1. **Clave Autogenerada**: Campo que genera un valor único automáticamente.
2. **Conector**: Intermediario entre una aplicación y la base de datos.
3. **Desfase Objeto-Relacional**: Diferencias entre el modelo orientado a objetos y el relacional.
4. **Driver JDBC**: Software que permite la comunicación entre Java y una base de datos.
5. **Iterador**: Recorre los resultados de una consulta SQL (ej., ResultSet).
6. **JDBC**: API de Java para interactuar con bases de datos relacionales.

**Conclusión**: JDBC es una herramienta poderosa para conectar aplicaciones Java con bases de datos relacionales, permitiendo un control total sobre las operaciones SQL y la gestión de transacciones de manera segura y eficiente.