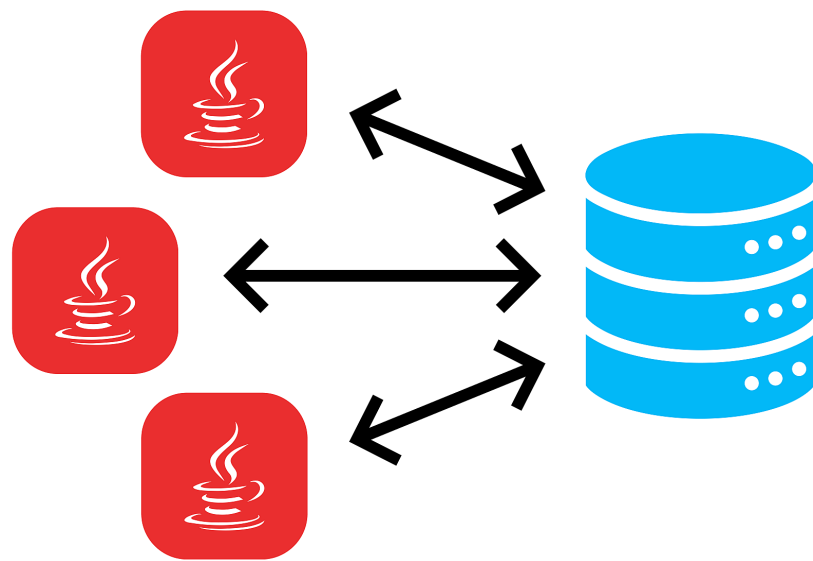


UD10: Acceso a Bases de Datos Relacionales



1. Introducción

- 1. 1. Conexión a las BBDD: conectores

2. JDBC

- 2. 1. Funciones del JDBC
- 2. 2. Drivers JDBC
- 2. 3. Instalación controlador
- 2. 4. Carga del controlador JDBC y conexión con la BD
 - 2. 4. 1. Conexión alternativa mediante `Driver`

3. Patrones de diseño aplicables

- 3. 1. Patrón `Singleton`
- 3. 2. Patrón `Pool Object` (DataBase Pool Connection `DBPC`)
- 3. 3. Patrón `DAO`

4. Acceso a BBDD

- 4. 1. Cargar el `Driver`
- 4. 2. Clase `DriverManager`
- 4. 3. Clase `Connection`
- 4. 4. Clase `Statement`
- 4. 5. Clase `ResultSet`

5. Navegabilidad y concurrencia

6. Consultas (`Query`)

- 6. 1. Navegación de un `ResultSet`
- 6. 2. Obteniendo datos del `ResultSet`
- 6. 3. Tipos de datos y conversiones
- 6. 4. Sentencias que no devuelven datos
- 6. 5. Asegurar la liberación de recursos

7. Modificación (`update`)

8. Inserción (`insert`)

9. Borrado (`delete`)

10. Sentencias predefinidas

- 10. 1. Ventajas de `PreparedStatement` desde el Punto de Vista de Seguridad

11. Píldoras informáticas relacionadas

12. Fuentes de información

1. Introducción

Hoy en día, la mayoría de aplicaciones informáticas necesitan almacenar y gestionar gran cantidad de datos. Esos datos, se suelen guardar en **bases de datos relacionales**, ya que éstas son las más extendidas actualmente.

Las bases de datos relacionales permiten organizar los datos en **tablas** y esas tablas y datos se relacionan mediante campos clave. Además se trabaja con el lenguaje estándar conocido como **SQL**, para poder realizar las consultas que deseemos a la base de datos.

Una base de datos relacional se puede definir de una manera simple como aquella que presenta la información en tablas con filas y columnas.

Una tabla es una serie de **filas** y **columnas**, en la que cada fila es un **registro** y cada columna es un **campo**. Un campo representa un dato de los elementos almacenados en la tabla (*NSS, nombre, etc.*). Cada registro representa un elemento de la tabla (el equipo Real Madrid, el equipo Real Murcia, etc.)

No se permite que pueda aparecer dos o más veces el mismo registro, por lo que uno o más campos de la tabla forman lo que se conoce como **clave primaria** (atributo que se elige como identificador en una tabla, de manera que no haya dos registros iguales, sino que se diferencien al menos en esa clave. Por ejemplo, en el caso de una tabla que guarda datos de personas, el número de la seguridad social, podría elegirse como clave primaria, pues sabemos que aunque haya dos personas llamadas, por ejemplo, Juan Pérez Pérez, estamos seguros de que su número de seguridad social será distinto).

El sistema gestor de bases de datos, en inglés conocido como: **Database Management System (DBMS)**, gestiona el modo en que los datos se almacenan, mantienen y recuperan.

En el caso de una base de datos relacional, el sistema gestor de base de datos se denomina: **Relational Database Management System (RDBMS)**.

Tradicionalmente, la programación de bases de datos ha sido como una Torre de Babel: gran cantidad de productos de bases de datos en el mercado, y cada uno "hablando" en su lenguaje privado con las aplicaciones.

Java, mediante **JDBC** (Java Database Connectivity, API que permite la ejecución de operaciones sobre bases de datos desde el lenguaje de programación Java, independientemente del sistema operativo donde se ejecute o de la base de datos a la cual se accede), permite simplificar el acceso a base de datos, proporcionando un lenguaje mediante el cual las aplicaciones pueden comunicarse con motores de bases de datos. Sun desarrolló este API para el acceso a bases de datos, con tres objetivos principales en mente:

- Ser un API con soporte de SQL: poder construir sentencias SQL e insertarlas dentro de llamadas al API de Java,
- Aprovechar la experiencia de los APIs de bases de datos existentes,
- Ser sencillo.

Ampliación

Qué es una API?

API: Application Programming Interface (Interfaz de Programación de Aplicaciones). Conjunto de reglas y protocolos que permite a diferentes aplicaciones o sistemas comunicarse entre sí. En pocas palabras, actúa como un intermediario que permite que dos programas informáticos se comuniquen y compartan datos entre ellos de manera segura y eficiente. Las APIs se utilizan comúnmente en el desarrollo de software para permitir la integración de diferentes sistemas, la creación de aplicaciones de terceros y la automatización de procesos.

Un ejemplo sencillo de API podría ser el servicio de pronóstico del tiempo proporcionado por una compañía meteorológica. Imagina que tienes una aplicación de clima en tu teléfono. Esta aplicación necesita mostrar el pronóstico del tiempo actualizado, pero no tiene la capacidad de predecir el clima por sí misma.

En lugar de eso, la aplicación utiliza una API proporcionada por una empresa meteorológica. Esta API permite que la aplicación envíe una solicitud con la ubicación actual del usuario y, a cambio, recibe datos sobre el clima en esa ubicación. La API proporciona estos datos en un formato estructurado, como JSON o XML, que la aplicación puede interpretar y mostrar de manera comprensible para el usuario.

En resumen, la aplicación de clima utiliza la API de la empresa meteorológica para obtener datos actualizados sobre el pronóstico del tiempo sin tener que desarrollar su propio sistema de predicción meteorológica. La API actúa como un puente entre la aplicación y los recursos de la compañía meteorológica, permitiendo que la aplicación acceda y utilice esos recursos de manera fácil y eficiente.

Ampliación

Bases de datos NoSQL vs Bases de datos relacionales

Las bases de datos NoSQL (Not Only SQL) son sistemas de gestión de bases de datos que no siguen el modelo relacional tradicional. Están diseñadas para manejar datos no estructurados o semiestructurados y para proporcionar flexibilidad en la forma en que se almacenan y acceden a los datos. Existen diferentes tipos:

- **Documentales:** Almacenan datos en documentos similares a JSON, como [MongoDB](#).
- **Clave-Valor:** Almacenan pares de clave-valor, como [Redis](#).
- **Columnar:** Almacenan datos en columnas en lugar de filas, como [Cassandra](#).
- **Grafos:** Almacenan datos en estructuras de grafos, como [Neo4j](#).

Entre sus ventajas, podemos encontrar las siguientes:

- **Escalabilidad Horizontal:** Pueden manejar grandes volúmenes de datos mediante la distribución de la carga en múltiples servidores.
- **Flexibilidad:** No requieren un esquema fijo, lo que permite manejar datos heterogéneos y evolucionar el modelo de datos fácilmente.
- **Alto Rendimiento:** Optimizadas para operaciones de lectura y escritura rápidas.

Diferencias con Bases de Datos Relacionales

Referente al Modelo de Datos, las **Relacionales** utilizan un modelo tabular con filas y columnas, y relaciones definidas mediante claves primarias y foráneas. Mientras que las **NoSQL** utilizan diversos modelos de datos (documentos, clave-valor, columnar, grafos) que permiten una mayor flexibilidad.

En cuanto a su Esquema, las **Relacionales** tienen esquemas rígidos y predefinidos. Cambiar el esquema puede ser complejo. Por contra, las **NoSQL** tienen esquemas dinámicos o inexistentes, permitiendo agregar nuevos campos sobre la marcha sin necesidad de modificar el esquema de la base de datos.

Las Consultas en las **Relacionales** utilizan SQL (Structured Query Language) para realizar consultas complejas y uniones entre tablas. Sin embargo, las **NoSQL** utilizan diferentes lenguajes de consulta o API específicos para el tipo de base de datos, que pueden ser menos complejos y más rápidos para ciertas operaciones.

Si hablamos de Escalabilidad, las **Relacionales** tienen escalabilidad vertical (mejorar el hardware del servidor). Pero las **NoSQL** cuentan con escalabilidad horizontal (añadir más servidores al clúster).

Por último, referente a la Integridad y Transacciones, las **Relacionales** cuentan con una fuerte consistencia y soporte robusto para transacciones ACID (Atomicidad, Consistencia, Aislamiento, Durabilidad). Por contra, las **NoSQL** en muchos casos, priorizan la disponibilidad y la partición tolerante sobre la consistencia (sistema BASE: Basic Availability, Soft state, Eventual consistency).

1.1. Conexión a las BBDD: conectores

Dejemos de momento de lado el desfase Objeto-Relacional y centrémonos ahora en el acceso a Base de Datos Relacionales desde los lenguajes de programación. Lo razonaremos en general y lo aplicaremos a Java.

Desde la década de los 80 que existen a pleno rendimiento las bases de datos relacionales. Casi todos los Sistemas Gestores de Bases de Datos (excepto los más pequeños como Access, SQLite o Base de LibreOffice) utilizan la arquitectura cliente-servidor. Esto significa que hay un ordenador central donde está instalado el Sistema Gestor de Bases de Datos Relacional que actúa como servidor, y habrá muchos clientes que se conectarán al servidor haciendo peticiones sobre la Base de Datos.

Los Sistemas Gestores de Bases de Datos inicialmente disponían de lenguajes de programación propios para poder hacer los accesos desde los clientes. Era muy consistente, pero a base de ser muy poco operativo:

- La empresa desarrolladora del SGBD debían mantener un lenguaje de programación, que resultaba necesariamente muy costoso, si no querían que quedara desfasado.
- Las empresas usuarias del SGBD, que se conectaban como clientes, se encontraban muy ligadas al servidor para tener que utilizar el lenguaje de programación para acceder al servidor, lo que no siempre se ajustaba a sus necesidades. Además, el plantearse cambiar de servidor, significaba que había que rehacer todos los programas, y por tanto una tarea de muchísima envergadura.

Para poder ser más operativos, había que desvincular los lenguajes de programación de los Sistemas Gestores de Bases de Datos utilizando unos estándares de conexión.

2. JDBC

Java puede conectarse con distintos SGBD y en diferentes sistemas operativos. Independientemente del método en que se almacenen los datos debe existir siempre un **mediador** entre la aplicación y el sistema de base de datos y en Java esa función la realiza **JDBC**.

Importante

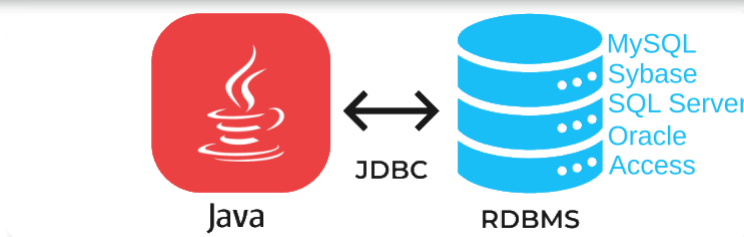
Para la conexión a las bases de datos utilizaremos la API estándar de JAVA denominada **JDBC** (*Java Data Base Connectivity*).

JDBC es un API incluido dentro del lenguaje Java para el acceso a bases de datos. Consiste en un conjunto de clases e interfaces escritas en Java que ofrecen un completo API para la programación con bases de datos, por lo tanto es la única solución 100% Java que permite el acceso a bases de datos.

JDBC es una especificación formada por una colección de interfaces y clases abstractas, que todos los fabricantes de drivers deben implementar si quieren realizar una implementación de su driver 100% Java y compatible con JDBC (JDBC-compliant driver). Debido a que JDBC está escrito completamente en Java también posee la ventaja de ser independiente de la plataforma.

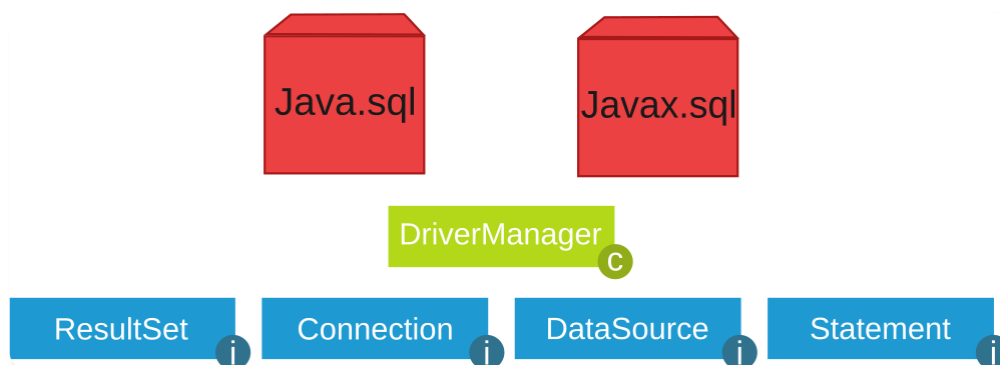
Importante

No será necesario escribir un programa para cada tipo de base de datos, una misma aplicación escrita utilizando JDBC podrá manejar bases de datos Oracle, Sybase, SQL Server, etc.



Además podrá ejecutarse en cualquier sistema operativo que posea una Máquina Virtual de Java, es decir, serán aplicaciones completamente independientes de la plataforma. Otras API'S que se suelen utilizar bastante para el acceso a bases de datos son DAO (Data Access Objects) y RDO (Remote Data Objects), y ADO (ActiveX Data Objects), pero el problema que ofrecen estas soluciones es que sólo son para plataformas Windows.

JDBC tiene sus clases en el paquete `java.sql` y otras extensiones en el paquete `javax.sql`.



2.1. Funciones del JDBC

Básicamente el API JDBC hace posible la realización de las siguientes tareas:

- Establecer una conexión con una base de datos.
- Enviar sentencias SQL.
- Manipular datos.
- Procesar los resultados de la ejecución de las sentencias.

2.2. Drivers JDBC

Los drivers nos permiten conectarnos con una base de datos determinada. Existen **cuatro tipos de drivers JDBC**, cada tipo presenta una filosofía de trabajo diferente. A continuación se pasa a comentar cada uno de los drivers:

- **JDBC-ODBC bridge plus ODBC driver** (tipo 1): permite al programador acceder a fuentes de datos ODBC existentes mediante JDBC. El JDBC-ODBC Bridge (puente JDBC-ODBC) implementa operaciones JDBC traduciéndolas a operaciones ODBC, se encuentra dentro del paquete `sun.jdbc.odbc` y contiene librerías nativas para acceder a ODBC.

Al ser usuario de ODBC depende de las dll de ODBC y eso limita la cantidad de plataformas en donde se puede ejecutar la aplicación.

- **Native-API partly-Java driver** (tipo 2): son similares a los drivers de tipo1, en tanto en cuanto también necesitan una configuración en la máquina cliente. Este tipo de driver convierte llamadas JDBC a llamadas de Oracle, Sybase, Informix, DB2 u otros SGBD. Tampoco se pueden utilizar dentro de applets al poseer código nativo.
- **JDBC-Net pure Java driver** (tipo 3): Estos controladores están escritos en Java y se encargan de convertir las llamadas JDBC a un protocolo independiente de la base de datos y en la aplicación servidora utilizan las funciones nativas del sistema de gestión de base de datos mediante el uso de una biblioteca JDBC en el servidor. La ventaja de esta opción es la portabilidad.
- **JDBC de Java cliente** (tipo 4): Estos controladores están escritos en Java y se encargan de convertir las llamadas JDBC a un protocolo independiente de la base de datos y en la aplicación servidora utilizan las funciones nativas del sistema de gestión de base de datos sin necesidad de bibliotecas. La ventaja de esta opción es la portabilidad. Son como los drivers de tipo 3 pero sin la figura del intermediario y tampoco requieren ninguna configuración en la máquina cliente. Los drivers de tipo 4 se pueden utilizar para servidores Web de tamaño pequeño y medio, así como para intranets.

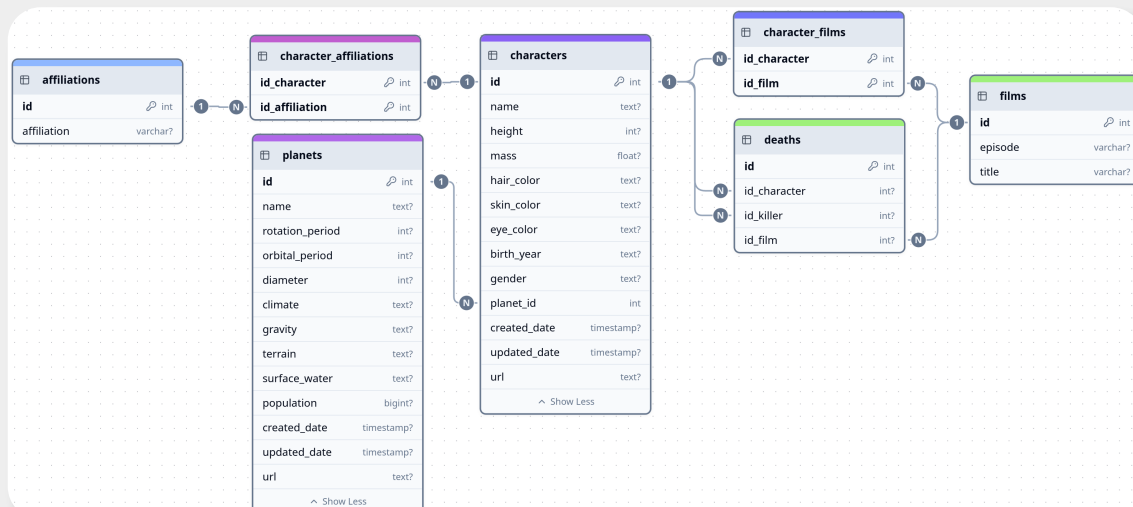
2.3. Instalación controlador

Consulta en el taller [UD10 T1 Conectores ES](#) indicaciones sobre como instalar el controlador que necesites según tu IDE, sistema operativo y gestor de BBDD.

Importante Base de datos de pruebas "starwars" y ""

Todo el código fuente que veras a continuación se basa en el taller [UD10 T2 AWS IntelliJ ES](#), que crea la BBDD "StarWars" en un servidor **MariaDB** alojado en la nube de **Amazon**. Para replicar las pruebas necesitaras la dirección de tu instancia de BBDD (algo parecido a `databaseamp.cipxbdkxiaqy.us-east-1.rds.amazonaws.com`), el puerto (por defecto 3306), el usuario (`admin`) y la contraseña (`123456Ab$`). Asegurate de haber completado primero el taller para poder probar el código.

Aquí te dejo el diagrama de la BD. Deberías familiarizarte con él antes de seguir.



2.4. Carga del controlador JDBC y conexión con la BD

El primer paso para conectarnos a una base de datos mediante JDBC es cargar el controlador apropiado. Estos controladores se distribuyen en un archivo `.jar` que provee el fabricante del SGBD y deben estar accesibles por la aplicación.

Para cargar el controlador (MySQL) se usan las siguientes sentencias:

```

1 package es.martinezpenya.ejemplos.UD10._02_JDBC;
2
3 import java.sql.Connection;
4 import java.sql.DriverManager;
5 import java.sql.SQLException;
6
7 public class UD10_01_ConectarMySQL {
8     public static void main(String[] args) {
9         try {
10             // Dependiendo de a qué tipo de SGBD queramos conectar cargaremos un
            // controlador u otro
11             // Intentar cargar el driver de MySQL
12             Class<?> c = Class.forName("com.mysql.cj.jdbc.Driver");
13             System.out.println("Cargado " + c.getName());
14
15             //Definir la url de conexión y los parámetros de usuario y contraseña
16             String host = "jdbc:mysql://databasedmp.cipxbdkxiagy.us-east-
17             1.rds.amazonaws.com:3306/starwars";
18             String username = "admin";
19             String password = "123456Ab$";
20             Connection con = DriverManager.getConnection(host, username, password);
21
22             System.out.println("Conexión completada");
23             //cerrar la conexión
24             con.close();
25         } catch (ClassNotFoundException cnfe) {
26             System.out.println(cnfe.getMessage());
27         } catch (SQLException ex) {
28             System.out.println("ERROR al conectar: " + ex.getMessage());
29         }
30     }
31 }
```

Observamos las siguientes cuestiones:

- Como ya hemos comentado alguna vez, la sentencia `Class.forName()` no sería necesaria en muchas aplicaciones. Pero nos asegura que hemos cargado el driver, y por tanto el `DriverManager` la sabrá manejar
- El `DriverManager` es capaz de encontrar el driver adecuado a través de la url proporcionada (sobre todo si el driver está cargado en memoria), y es quien nos proporciona el objeto `Connection` por medio del método `getConnection()`. Hay otra manera de obtener el `Connection` por medio del objeto `Driver`, como veremos más adelante, pero también será pasando indirectamente por `DriverManager`.
- Si no se encuentra la clase del driver (por no tenerlo en las librerías del proyecto, o haber escrito mal su nombre) se producirá la excepción `ClassNotFoundException`. Es conveniente tratarla con `try ... catch`.
- Si no se puede establecer la conexión por alguna razón se producirá la excepción `SQLException`. Al igual que en el caso anterior, es conveniente tratarla con `try ... catch`.
- El objeto `Connection` mantendrá una conexión con la Base de Datos desde el momento de la creación hasta el momento de cerrarla con `close()`. Es muy importante cerrar la conexión, no sólo para liberar la memoria de nuestro ordenador (que al cerrar la aplicación liberaría), sino sobre todo para cerrar la sesión abierta en el Servidor de Bases de Datos.

2.4.1. Conexión alternativa mediante `Driver`

Una manera de conectar alternativa a las anteriores es utilizando el objeto `Driver`. La clase `java.sql.Driver` pertenece a la **API JDBC**, pero no es instanciable, y tan sólo es una interfaz, para que las clases `Driver` de los contenedores hereden de ella e implementen la manera exacta de acceder al SGBD correspondiente. Como no es instanciable (no podemos hacer `new Driver()`) la manera de crearlo es a través del método `getDriver()` del `DriverManager`, que seleccionará el driver adecuado a partir de la url. Ya sólo quedarán definir algunas propiedades, como el usuario y la contraseña, y obtener el `Connection` por medio del método `connect()`.

La manera de conectar a través de un objeto `Driver` es más larga, pero más completa ya que se podrían especificar más cosas. Y quizás ayude a entender el montaje de los controladores de los diferentes SGBD en Java.

```

1  package es.martinezpenya.ejemplos.UD10._02_JDBC;
2
3  import java.sql.Connection;
4  import java.sql.Driver;
5  import java.sql.DriverManager;
6  import java.sql.SQLException;
7  import java.util.Properties;
8
9  public class UD10_02_ConectarMySQLDriver {
10     public static void main(String[] args) {
11         //Definir la url de conexión y los parámetros de usuario y contraseña
12         String url = "jdbc:mysql://databasedmp.cipxbdkxiaqy.us-east-1.rds.amazonaws.com:3306/starwars";
13         String username = "admin";
14         String password = "123456Ab$";
15
16         try {
17             Driver driver = DriverManager.getDriver(url);
18             //Establecer las propiedades de la conexión
19             Properties properties = new Properties();
20             properties.setProperty("user", username);
21             properties.setProperty("password", password);
22             //Conectar
23             Connection con = driver.connect(url, properties);
24             System.out.println("Conexión completada a través de Driver");
25             //Cerrar la conexión
26             con.close();
27         } catch (SQLException ex) {
28             System.out.println("ERROR al conectar: " + ex.getMessage());
29         }
30     }
31 }

```


3. Patrones de diseño aplicables

3.1. Patrón Singleton

El patrón `Singleton` es un patrón de diseño que asegura que una clase tenga **una única instancia** durante toda la ejecución del programa, y proporciona un **punto de acceso global** a esa instancia. Esto es especialmente útil cuando queremos controlar el acceso a recursos compartidos, como una conexión a una base de datos.

¿Por qué usar `Singleton` en el acceso a bases de datos?

Cuando trabajamos con bases de datos relacionales, es común que necesitemos una **única conexión** o un **pool de conexiones** para interactuar con la base de datos. Si no controlamos cómo se crean estas conexiones, podríamos terminar con múltiples instancias de conexiones abiertas, lo que puede llevar a:

1. **Consumo excesivo de recursos:** Cada conexión consume memoria y recursos del sistema.
2. **Problemas de consistencia:** Si hay múltiples conexiones, podríamos tener inconsistencias en los datos.
3. **Dificultad para gestionar:** Sería complicado cerrar y controlar todas las conexiones abiertas.

Vamos a implementar una clase `DatabaseConnection` que siga el patrón Singleton. Esta clase se encargará de gestionar la conexión a la base de datos.

```

1  package es.martinezpenya.ejemplos.UD10._03_Patrones._01_Singleton;
2
3  import java.sql.Connection;
4  import java.sql.DriverManager;
5  import java.sql.SQLException;
6
7  public class DatabaseConnection {
8
9      // 1. Declaramos una instancia estática y privada de la clase
10     private static DatabaseConnection instance;
11     private Connection connection;
12
13     // 2. Hacemos el constructor privado para evitar que se creen instancias desde fuera
14     private DatabaseConnection() {
15         try {
16             // Establecemos la conexión a la base de datos
17             String url = "jdbc:mysql://databasedmp.cipxbdkxiaqy.us-east-
18 1.rds.amazonaws.com:3306/starwars";
19             String user = "admin";
20             String password = "123456Ab$";
21             this.connection = DriverManager.getConnection(url, user, password);
22             System.out.println("Conexión a la base de datos establecida.");
23         } catch (SQLException e) {
24             System.out.println("Error al conectar a la base de datos: " + e.getMessage());
25         }
26     }
27
28     // 3. Método estático para obtener la instancia única (Singleton)
29     public static DatabaseConnection getInstance() {
30         if (instance == null) {
31             instance = new DatabaseConnection();
32         }
33         return instance;
34     }
35
36     // 4. Método para obtener la conexión
37     public Connection getConnection() {
38         return connection;
39     }
40
41     // 5. Método para cerrar la conexión (opcional)
42     public void closeConnection() {
43         if (connection != null) {
44             try {
45                 connection.close();
46                 System.out.println("Conexión cerrada.");
47             } catch (SQLException e) {
48                 System.out.println("Error al cerrar la conexión: " + e.getMessage());
49             }
50         }
51     }
52 }

```

```

49     }
50 }
51 }

```

Explicación paso a paso

1. **Instancia única:** La variable `instance` es estática y privada, lo que significa que pertenece a la clase y no puede ser accedida directamente desde fuera. Esto asegura que solo exista una instancia de `DatabaseConnection`.
2. **Constructor privado:** Al hacer el constructor privado, evitamos que otras clases puedan crear instancias de `DatabaseConnection` usando `new DatabaseConnection()`. Esto garantiza que la única forma de obtener la instancia sea a través del método `getInstance()`.
3. **Método `getInstance()`:** Este método es estático y se encarga de crear la instancia única si no existe, o devolver la existente si ya fue creada. Esto asegura que siempre trabajemos con la misma conexión.
4. **Conexión a la base de datos:** La conexión se establece en el constructor privado, y podemos acceder a ella a través del método `getConnection()`.
5. **Cerrar la conexión:** Aunque no es estrictamente necesario, es buena práctica proporcionar un método para cerrar la conexión cuando ya no la necesitamos.

¿Cómo usamos esta clase?

Aquí tienes un ejemplo de cómo usar la clase `DatabaseConnection` en tu aplicación:

```

1 package es.martinezpenya.ejemplos.UD10._03_Patrones._01_Singleton;
2
3 import java.sql.Connection;
4
5 public class DatabaseConnectionTest {
6     public class Main {
7         public static void main(String[] args) {
8             // Obtenemos la instancia única de DatabaseConnection
9             DatabaseConnection dbConnection = DatabaseConnection.getInstance();
10
11             // Usamos la conexión para interactuar con la base de datos
12             Connection connection = dbConnection.getConnection();
13             // Aquí podríamos ejecutar consultas SQL, por ejemplo:
14             // Statement statement = connection.createStatement();
15             // ResultSet resultSet = statement.executeQuery("SELECT * FROM usuarios");
16
17             // Cerramos la conexión cuando ya no la necesitamos
18             dbConnection.closeConnection();
19         }
20     }
21 }

```

3.2. Patrón Pool Object (DataBase Pool Connection DBPC)

El patrón Pool de Objetos es un patrón de diseño que tiene como objetivo **reutilizar objetos** que son costosos de crear, como conexiones a bases de datos, hilos, o sockets. En lugar de crear y destruir estos objetos repetidamente, el patrón mantiene un **conjunto de objetos ya creados** (un "pool") que pueden ser prestados y devueltos cuando ya no se necesitan.

¿Por qué usar un Pool de Objetos en el acceso a bases de datos?

Cuando trabajamos con bases de datos, establecer una nueva conexión es una operación costosa en términos de tiempo y recursos. Si abrimos y cerramos conexiones constantemente, el rendimiento de la aplicación se verá afectado. Aquí es donde entra en juego el **Database Connection Pooling (DBCP)**.

Un **Database Connection Pool** es una implementación específica del patrón Pool de Objetos que se encarga de gestionar múltiples conexiones a la base de datos. En lugar de crear una nueva conexión cada vez que se necesita, el pool:

1. **Mantiene un conjunto de conexiones abiertas** y listas para ser usadas.
2. **Presta una conexión** cuando un componente de la aplicación la solicita.
3. **Recupera la conexión** cuando el componente termina de usarla, permitiendo que sea reutilizada por otros componentes.

¿Qué es HikariCP?

HikariCP es una librería de conexión a bases de datos que implementa un **Database Connection Pool**. Es conocida por ser extremadamente rápida, ligera y eficiente en comparación con otras soluciones como Apache DBCP o C3P0. HikariCP se ha convertido en el estándar de facto para aplicaciones Java que requieren alta concurrencia y rendimiento.

Además en IntelliJ con maven es muy sencillo, ya que solo debemos agregar esta dependencia al `pom.xml`:

```
1 <!-- HikariCP -->
2 <!-- https://mvnrepository.com/artifact/com.zaxxer/HikariCP -->
3 <dependency>
4     <groupId>com.zaxxer</groupId>
5     <artifactId>HikariCP</artifactId>
6     <version>6.2.1</version>
7 </dependency>
```

Ventajas de usar un pool de conexiones:

- **Mejora del rendimiento:** Al reutilizar conexiones existentes, se evita el costo de crear y cerrar conexiones repetidamente.
- **Control de recursos:** El pool limita el número máximo de conexiones, evitando que la base de datos se sature.
- **Escalabilidad:** Permite que múltiples hilos o componentes de la aplicación accedan a la base de datos de manera concurrente sin problemas.
- **Manejo de errores:** El pool puede manejar conexiones caídas o inválidas, reemplazándolas automáticamente.

```
1 package es.martinezpenya.ejemplos.UD10._03_Patrones._02_PoolObject;
2
3 import com.zaxxer.hikari.HikariConfig;
4 import com.zaxxer.hikari.HikariDataSource;
5 import java.sql.Connection;
6 import java.sql.SQLException;
7
8 public class HikariCPSingleton {
9
10     private static HikariDataSource dataSource;
11
12     // Configuración del pool de conexiones
13     static {
14         HikariConfig config = new HikariConfig();
15         config.setJdbcUrl("jdbc:mariadb://databasedmp.cipxbdkxiagy.us-east-
16 1.rds.amazonaws.com:3306/starwars"); // URL de la base de datos
17         config.setUsername("admin"); // Usuario de la base de datos
18         config.setPassword("123456Ab$"); // Contraseña de la base de datos
19         config.setMaximumPoolSize(10); // Número máximo de conexiones en el pool
20         config.setMinimumIdle(2); // Número mínimo de conexiones inactivas en el pool
21         config.setIdleTimeout(30000); // Tiempo de espera para conexiones inactivas (en
22 milisegundos)
23         config.setMaxLifetime(1800000); // Tiempo máximo de vida de una conexión (en
24 milisegundos)
25         config.setConnectionTimeout(30000); // Tiempo de espera para obtener una conexión
26         (en milisegundos)
27
28         dataSource = new HikariDataSource(config);
29     }
30
31     // Método para obtener una conexión del pool
32     public static Connection getConnection() throws SQLException {
33         return dataSource.getConnection();
34     }
35
36     // Método para cerrar el pool de conexiones (opcional)
37     public static void closeDataSource() {
38         if (dataSource != null) {
39             dataSource.close();
40         }
41     }
42 }
```

¿Cómo usamos esta clase?

Aquí tienes un ejemplo de cómo usar la clase `HikariCPSingleton` en tu aplicación:

```

1 package es.martinezpenya.ejemplos.UD10._03_Patrones._02_PoolObject;
2
3 import java.sql.Connection;
4 import java.sql.SQLException;
5
6 public class HikariCPSingletonTest {
7
8     private static Connection connection;
9
10    public static void main(String[] args) {
11        try {
12            connection = HikariCPSingleton.getConnection();
13            // Hacemos algo con la conexión
14        } catch (SQLException e) {
15            System.out.println("ERROR al conectar: " + e.getMessage());
16        }
17    }
18 }

```

Importante

Este último Patrón `Pool Object` será el que usaremos para el resto de ejemplos de la unidad, y el que te recomiendo que uses. Si no te convence mi opinión, puedes ver el vídeo de [makigas](#) que tienes más abajo.

3.3. Patrón DAO

Proporciona una abstracción para las operaciones CRUD (Create, Read, Update, Delete) con la base de datos. Este patrón separa la lógica de negocio de la lógica de acceso a datos, facilitando el mantenimiento y la escalabilidad del código.

El patrón DAO se utiliza para encapsular todo el acceso a la base de datos en una clase separada. Esto permite que la lógica de negocio interactúe con la base de datos a través de métodos definidos en el DAO, sin preocuparse por los detalles de la implementación de la base de datos.

Ejemplo para una clase `Film` sencilla

```

1 package es.martinezpenya.ejemplos.UD10._03_Patrones._03_DAO;
2
3 public class Film {
4     private int id;
5     private String episode;
6     private String title;
7
8     // Getters y setters
9     public int getId() {
10        return id;
11    }
12
13    public void setId(int id) {
14        this.id = id;
15    }
16
17    public String getEpisode() {
18        return episode;
19    }
20
21    public void setEpisode(String episode) {
22        this.episode = episode;
23    }
24
25    public String getTitle() {
26        return title;
27    }
28
29    public void setTitle(String title) {
30        this.title = title;
31    }
32 }

```

32 | }

La interfaz de **FilmDAO** :

```

1 package es.martinezpenya.ejemplos.UD10._03_Patrones._02_DAO;
2
3 public interface FilmDAO {
4     void agregarFilm(Film film);           // Crear una nueva película
5     Film obtenerFilm(int id);              // Obtener una película por su ID
6     void actualizarFilm(Film film);        // Actualizar una película existente
7     void eliminarFilm(int id);            // Eliminar una película por su ID
8 }

```

Ampliación

Dejo aquí una cuestión en el aire para que reflexiones, si todas las entidades van a usar una Interfaz similar... que solución propones para definir una única Interfaz DAO que sirva para todas las entidades? (pista: Genéricos)

Y por último la implementación de la interfaz en **FilmDAOImplementado** :

```

1 package es.martinezpenya.ejemplos.UD10._03_Patrones._03_DAO;
2
3 import es.martinezpenya.ejemplos.UD10._03_Patrones._02_PoolObject.HikariCPSingleton;
4
5 import java.sql.Connection;
6 import java.sql.PreparedStatement;
7 import java.sql.ResultSet;
8 import java.sql.SQLException;
9
10 public class FilmDAOImplementado implements FilmDAO {
11
12     private Connection connection;
13
14     public FilmDAOImplementado() {
15         try {
16             connection = HikariCPSingleton.getConnection();
17         } catch (SQLException e) {
18             System.out.println("ERROR al conectar: " + e.getMessage());
19         }
20     }
21
22     @Override
23     public void agregarFilm(Film film) {
24         try {
25             PreparedStatement ps = connection.prepareStatement("INSERT INTO films
26 (id, episode, title) VALUES (?, ?, ?)");
27             ps.setInt(1, film.getId());
28             ps.setString(2, film.getEpisode());
29             ps.setString(3, film.getTitle());
30             ps.executeUpdate();
31         } catch (SQLException e) {
32             e.printStackTrace();
33         }
34     }
35
36     @Override
37     public Film obtenerFilm(int id) {
38         Film film = null;
39         try {
40             PreparedStatement ps = connection.prepareStatement("SELECT * FROM films
41 WHERE id = ?");
42             ps.setInt(1, id);
43             ResultSet rs = ps.executeQuery();
44             if (rs.next()) {
45                 film = new Film();
46                 film.setId(rs.getInt("id"));
47                 film.setEpisode(rs.getString("episode"));
48                 film.setTitle(rs.getString("title"));
49             }
50         } catch (SQLException e) {
51             e.printStackTrace();
52         }
53     }
54 }

```

```

50     }
51     return film;
52 }
53
54 @Override
55 public void actualizarFilm(Film film) {
56     try {
57         PreparedStatement ps = connection.prepareStatement("UPDATE films SET
episode = ?, title = ? WHERE id = ?");
58         ps.setString(1, film.getEpisode());
59         ps.setString(2, film.getTitle());
60         ps.setInt(3, film.getId());
61         ps.executeUpdate();
62     } catch (SQLException e) {
63         e.printStackTrace();
64     }
65 }
66
67 @Override
68 public void eliminarFilm(int id) {
69     try {
70         PreparedStatement ps = connection.prepareStatement("DELETE FROM films
WHERE id = ?");
71         ps.setInt(1, id);
72         ps.executeUpdate();
73     } catch (SQLException e) {
74         e.printStackTrace();
75     }
76 }
77 }

```

Entre las ventajas de usar el patrón encontramos:

- **Separación de responsabilidades:** La lógica de acceso a datos se separa de la lógica de negocio.
- **Reutilización de código:** Los métodos de acceso a datos pueden ser reutilizados por diferentes partes de la aplicación.
- **Facilidad de mantenimiento:** Cambiar la implementación de la base de datos no afecta la lógica de negocio.

Con la siguiente clase `TestFilmDAOImplementado.java` podemos probar los métodos implementados:

```

1  package es.martinezpenya.ejemplos.UD10._03_Patrones._03_DAO;
2
3  public class TestFilmDAOImplementado {
4
5      public static void main(String[] args) {
6          // Crear una instancia de FilmDAOImplementado
7          FilmDAOImplementado filmDAO = new FilmDAOImplementado();
8
9          // Crear un film ficticio para el episodio 7
10         Film film = new Film();
11         film.setId(7); // Cambiamos el episodio a 7
12         film.setEpisode("VII");
13         film.setTitle("The Force Awakens"); // Cambiamos el título
14
15         // 1. Insertar el film en la base de datos
16         System.out.println("Insertando el film...");
17         filmDAO.agregarFilm(film);
18         System.out.println("Film insertado: " + film.getTitle());
19
20         // 2. Recuperar el film recién insertado (asumiendo que el ID es 1)
21         System.out.println("\nRecuperando el film con ID 7...");
22         Film filmRecuperado = filmDAO.obtenerFilm(7);
23         if (filmRecuperado != null) {
24             System.out.println("Film recuperado: " + filmRecuperado.getTitle() + " (" +
filmRecuperado.getEpisode() + ")");
25         } else {
26             System.out.println("No se encontró el film con ID 7.");
27         }
28
29         // 3. Modificar el film recuperado
30         if (filmRecuperado != null) {

```

```

31         System.out.println("\nModificando el film...");
32         filmRecuperado.setEpisode("Episode VII"); // Cambiamos el título para agregar
    más detalle
33         filmDAO.actualizarFilm(filmRecuperado);
34         System.out.println("Film modificado: " + filmRecuperado.getTitle());
35
36         // Verificar la modificación
37         System.out.println("\nRecuperando el film modificado...");
38         Film filmModificado = filmDAO.obtenerFilm(filmRecuperado.getId());
39         if (filmModificado != null) {
40             System.out.println("Film modificado recuperado: " +
    filmModificado.getTitle() + " (" + filmModificado.getEpisode() + ")");
41         } else {
42             System.out.println("No se encontró el film modificado.");
43         }
44     }
45
46     // 4. Eliminar el film
47     if (filmRecuperado != null) {
48         System.out.println("\nEliminando el film con ID " + filmRecuperado.getId() +
    "...");
49         filmDAO.eliminarFilm(filmRecuperado.getId());
50         System.out.println("Film eliminado.");
51
52         // Verificar la eliminación
53         System.out.println("\nIntentando recuperar el film eliminado...");
54         Film filmEliminado = filmDAO.obtenerFilm(filmRecuperado.getId());
55         if (filmEliminado == null) {
56             System.out.println("El film fue eliminado correctamente.");
57         } else {
58             System.out.println("El film no fue eliminado.");
59         }
60     }
61 }
62 }

```

El resultado será:

```

1  Insertando el film...
2  Film insertado: The Force Awakens
3
4  Recuperando el film con ID 7...
5  Film recuperado: The Force Awakens (VII)
6
7  Modificando el film...
8  Film modificado: The Force Awakens
9
10 Recuperando el film modificado...
11 Film modificado recuperado: The Force Awakens (Episode VII)
12
13 Eliminando el film con ID 7...
14 Film eliminado.
15
16 Intentando recuperar el film eliminado...
17 El film fue eliminado correctamente.

```

4. Acceso a BBDD

En este apartado se ofrece una introducción a los aspectos fundamentales del acceso a bases de datos mediante código Java. En los siguientes apartados se explicarán algunos aspectos en mayor detalle, sobre todo los relacionados con las clases `Statement` y `ResultSet`.

4.1. Cargar el `Driver`

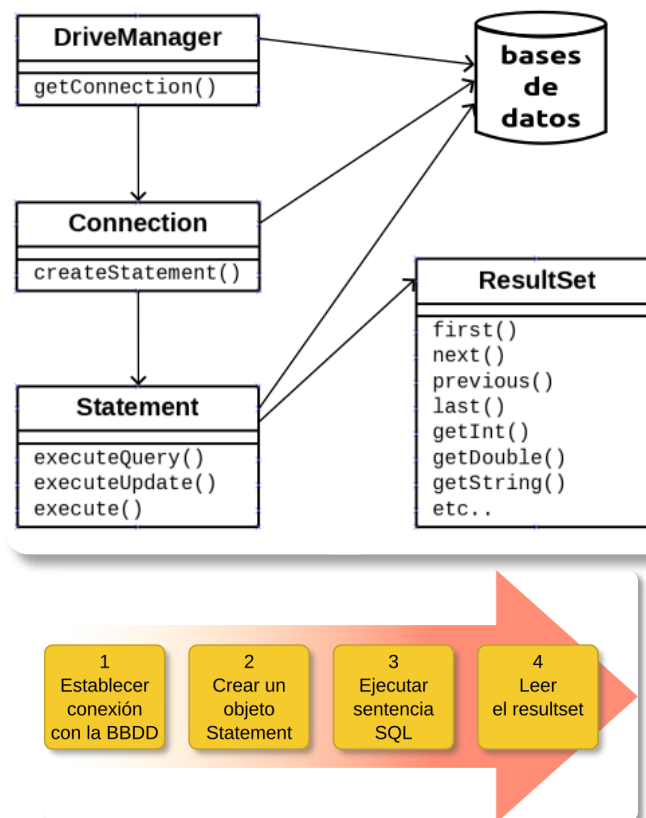
En un proyecto Java que realice conexiones a bases de datos es necesario, antes que nada, utilizar `Class.forName(...)` para cargar dinámicamente el `Driver` que vamos a utilizar. Esto solo es necesario hacerlo una vez en nuestro programa. Puede lanzar excepciones por lo que es necesario utilizar un bloque `try-catch`.

```
1  try {
2      Class<?> c = Class.forName("com.mysql.cj.jdbc.Driver");
3      System.out.println("Cargado " + c.getName());
4  } catch (Exception e) {
5      // manejamos el error
6  }
```

Hay que tener en cuenta que las clases y métodos utilizados para conectarse a una base de datos (explicados más adelante) funcionan con todos los drivers disponibles para Java (JDBC es solo uno, hay muchos más). Esto es posible ya que el estándar de Java solo los define como interfaces (*interface*) y cada librería driver los implementa (define las clases y su código). Por ello es necesario utilizar `Class.forName(...)` para indicarle a Java qué driver vamos a utilizar.

Este nivel de abstracción facilita el desarrollo de proyectos ya que si necesitáramos utilizar otro sistema de base de datos (que no fuera *MySQL*) solo necesitaríamos cambiar la línea de código que carga el driver y poco más. Si cada sistema de base de datos necesitara que utilizáramos distintas clases y métodos todo sería mucho más complicado.

Las cuatro clases fundamentales que toda aplicación Java necesita para conectarse a una base de datos y ejecutar sentencias son: `DriverManager`, `Connection`, `Statement` y `ResultSet`.



4.2. Clase DriverManager

Paso 1: Establecer conexión con la BBDD

```

1  /* Para MySQL:
2      jdbc --> driver
3      mysql --> protocolo driver
4      localhost --> dirección del servidor
5      3306 --> puerto
6      gestionPedidos --> nombre de la BD
7  */
8  jdbc:mysql:localhost:3306/gestionPedidos
9
10 /* Para SQL Server:
11     jdbc --> driver
12     odbc --> protocolo driver
13     DSN_gestionPedidos --> Nombre de la BD
14 */
15 jdbc:odbc:DSN_gestionPedidos
16
17 /* Para Oracle:
18     jdbc --> driver
19     oracle --> protocolo driver
20     juan --> nombre de usuario
21     @servidor --> dirección del servidor
22     3306 --> puerto
23     gestionPedidos --> nombre de la BD
24 */
25 jdbc:oracle:juan@servidor:3306:gestionPedidos

```

Vamos a necesitar información adicional como son los datos de *usuario* y *contraseña*.

La clase `java.sql.DriverManager` es la capa gestora del driver JDBC. Se encarga de manejar el Driver apropiado y **permite crear conexiones con una base de datos** mediante el método estático `getConnection()` que tiene dos variantes:

```

1  //opcion 1
2  DriverManager.getConnection(String URL)
3
4  //opcion2
5  DriverManager.getConnection(String URL, String user, String password)

```

Este método intentará establecer una conexión con la base de datos según el URL indicado. Opcionalmente se le puede pasar el usuario y contraseña como argumento (también se puede indicar en la propia URL). Si la conexión es satisfactoria devolverá un objeto `Connection`.

Ejemplo de conexión a la base de datos *prueba* en localhost:

```

1  //mysql://<username>:<password>@<host>:<port>/<db_name>
2  String url = "jdbc:mysql://localhost:3306/prueba";
3  Connection conn = DriverManager.getConnection(url, "root", "");

```

Este método puede lanzar dos tipos de excepciones (que habrá que manejar con un *try-catch*):

- `SQLException`: la conexión no ha podido producirse. Puede ser por multitud de motivos como una URL mal formada, un error en la red, host o puerto incorrecto, base de datos no existente, usuario y contraseña no válidos, etc.
- `SQLTimeoutException`: se ha superado el `LoginTimeout` sin recibir respuesta del servidor.

Ampliación

Aquí podemos ver un vídeo en el que Makigas explica porqué en el mundo real no se usa

`DriverManager.getConnection()` :



4.3. Clase `Connection`

Paso 2. Crear un objeto `Statement`

Un objeto `java.sql.Connection` **representa una sesión de conexión con una base de datos**. Una aplicación puede tener tantas conexiones como necesite, ya sea con una o varias bases de datos.

El método más relevante es `createStatement()` que devuelve un objeto `Statement` asociado a dicha conexión que permite ejecutar sentencias SQL. El método `createStatement()` puede lanzar excepciones de tipo `SQLException`.

```
1 | Statement st = conn.createStatement();
```

Cuando ya no la necesitemos es aconsejable **cerrar la conexión con** `close()` para liberar recursos:

```
1 | conn.close();
```

4.4. Clase `Statement`

Paso 3. Ejecutar sentencia SQL

Un objeto `java.sql.Statement` permite **ejecutar sentencias SQL en la base de datos** a través de la conexión con la que se creó el `Statement` (ver apartado anterior). Los tres métodos más comunes de ejecución de sentencias SQL son `executeQuery(...)`, `executeUpdate(...)` y `execute(...)`. Pueden lanzar excepciones de tipo `SQLException` y `SQLTimeoutException`.

- `ResultSet executeQuery(String sql)` : ejecuta la sentencia sql indicada (de tipo *SELECT*). Devuelve un objeto `ResultSet` con los datos proporcionados por el servidor.

```
1 | ResultSet rs = st.executeQuery("SELECT * FROM vendedores");
```

- `int executeUpdate(String sql)` : ejecuta la sentencia SQL indicada (de tipo *DML* como por ejemplo *INSERT*, *UPDATE* o *DELETE*). Devuelve un el número de registros que han sido afectados (insertados, modificados o eliminados).

```
1 | int nr = st.executeUpdate ("INSERT INTO vendedores VALUES (1,'Pedro Gil', '2017-04-11', 15000);")
```

Cuando ya no lo necesitemos es aconsejable **cerrar el `Statement` con `close()`** para liberar recursos:

```
1 | st.close();
```

Importante Podríamos decir que este `ResultSet` es una especie de *tabla virtual* que se almacena en memoria con la información en su interior.

4.5. Clase `ResultSet`

Paso 4. Leer el `ResultSet`

Un objeto `java.sql.ResultSet` contiene un conjunto de resultados (datos) obtenidos tras ejecutar una sentencia SQL, normalmente de tipo SELECT. Es una **estructura de datos en forma de tabla** con **registros (filas)** que podemos recorrer para acceder a la información de sus **campos (columnas)**.

`ResultSet` utiliza internamente un cursor que apunta al *registro actual* sobre el que podemos operar. Inicialmente dicho cursor está situado antes de la primera fila y disponemos de varios métodos para desplazar el cursor. El más común es `next()`:

- `boolean next()`: mueve el cursor al siguiente registro. Devuelve `true` si fue posible y `false` en caso contrario (si ya llegamos al final de la tabla).

Algunos de los métodos para obtener los datos del registro actual son:

- `String getString(String columnLabel)`: devuelve un dato `String` de la columna indicada por su nombre.

Por ejemplo:

```
1 | rs.getString("nombre");
```

- `String getString(int columnIndex)`: devuelve un dato `String` de la columna indicada por su índice

Importante La primera columna es la 1, no la cero 🧑

Por ejemplo:

```
1 | rs.getString(2);
```

Existen métodos análogos a los anteriores para obtener valores de tipo `int`, `long`, `float`, `double`, `boolean`, `Date`, `Time`, `Array`, etc. Pueden consultarse todos en la [documentación oficial de Java](#).

- `int getInt(String columnLabel)`
- `int getInt(int columnIndex)`
- `double getDouble(String columnLabel)`
- `double getDouble(int columnIndex)`
- `boolean getBoolean(String columnLabel)`
- `boolean getBoolean(int columnIndex)`
- `Date getDate(String columnLabel)`
- `Date getDate(int columnIndex)`
- etc.

Más adelante veremos cómo se realiza la modificación e inserción de datos.

Todos estos métodos pueden lanzar una `SQLException`.

Veamos un ejemplo de cómo recorrer un `ResultSet` llamado `rs` y mostrarlo por pantalla:

```
1 while(rs.next()) {  
2     int id = rs.getInt("id");  
3     String nombre = rs.getString("nombre");  
4     Date fecha = rs.getDate("fecha_ingreso");  
5     float salario = rs.getFloat("salario");  
6     System.out.println(id + " " + nombre + " " + fecha + " " + salario);  
7 }
```

5. Navegabilidad y concurrencia

Cuando invocamos a `createStatement()` **sin argumentos**, como hemos visto anteriormente, al ejecutar sentencias SQL obtendremos un `ResultSet` **por defecto en el que el cursor solo puede moverse hacia adelante y los datos son de solo lectura**. A veces esto no es suficiente y necesitamos mayor funcionalidad.

Por ello el método `createStatement()` está sobrecargado (existen varias versiones de dicho método) lo cual nos permite invocarlo con argumentos en los que podemos especificar el funcionamiento.

- `Statement createStatement (int resultSetType, int resultSetConcurrency)`: devuelve un objeto `Statement` cuyos objetos `ResultSet` serán del tipo y concurrencia especificados. Los valores válidos son constantes definidas en `ResultSet`.

Valores válidos para el argumento `resultSetType` indica el tipo de `ResultSet`:

- `ResultSet.TYPE_FORWARD_ONLY`: `ResultSet` por defecto, *forward-only* y *no-actualizable*.
 - Solo permite movimiento hacia delante con `next()`.
 - Sus datos NO se actualizan. Es decir, no reflejará cambios producidos en la base de datos. Contiene una instantánea del momento en el que se realizó la consulta.
- `ResultSet.TYPE_SCROLL_INSENSITIVE`: `ResultSet` desplazable y no actualizable.
 - Permite libertad de movimiento del cursor con otros métodos como `first()`, `previous()`, `last()`, etc. además de `next()`.
 - Sus datos NO se actualizan, como en el caso anterior.
- `ResultSet.TYPE_SCROLL_SENSITIVE`: `ResultSet` desplazable y actualizable.
 - Permite libertad de movimientos del cursor, como en el caso anterior.
 - Sus datos SÍ se actualizan. Es decir, mientras el `ResultSet` esté abierto se actualizará automáticamente con los cambios producidos en la base de datos. Esto puede suceder incluso mientras se está recorriendo el `ResultSet`, lo cual puede ser conveniente o contraproducente según el caso.

Diferencia entre `TYPE_SCROLL_INSENSITIVE` y `TYPE_SCROLL_SENSITIVE`

```

1 package es.martinezpenya.ejemplos.UD10._05_NavegabilidadConcurrencia;
2
3 import
4 es.martinezpenya.ejemplos.UD10._03_Patrones._02_PoolObject.HikariCPSingleton;
5
6 import java.sql.*;
7
8 public class EjemploScrollInsensitive {
9
10     public static void main(String[] args) {
11         try (Connection con = HikariCPSingleton.getConnection();
12             Statement stmt =
13 con.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
14 ResultSet.CONCUR_READ_ONLY);
15         ResultSet rs = stmt.executeQuery("SELECT id, name FROM
16 characters")) {
17
18             // Mover a la primera fila
19             if (rs.first()) {
20                 System.out.println("primera fila: " + rs.getInt("id") + ", " +
21 + rs.getString("name"));
22             }
23
24             // Mover a la última fila
25             if (rs.last()) {
26                 System.out.println("última fila: " + rs.getInt("id") + ", " +
27 rs.getString("name"));
28             }
29
30             // Simulamos un retraso y actualizamos la base de datos (en otra
31 sesión)
32             System.out.println("Esperando las actualizaciones...");
33             Thread.sleep(10000); // Esperar 10 segundos
34         }
35     }
36 }
```

```

27
28         // Mover a la primera fila otra vez
29         if (rs.first()) {
30             System.out.println("primera fila después de esperar: " +
rs.getInt("id") + ", " + rs.getString("name"));
31         }
32
33     } catch (SQLException | InterruptedException ex) {
34         ex.printStackTrace();
35     }
36 }
37 }

```

En este ejemplo, incluso si la base de datos cambia mientras el programa está esperando (durante el `Thread.sleep(10000)`), el `ResultSet` no reflejará esos cambios cuando se vuelva a consultar la primera fila.

En caso de cambiar el valor `TYPE_SCROLL_INSENSITIVE` por `TYPE_SCROLL_SENSITIVE`, si hay cambios en la base de datos durante el tiempo de espera, el `ResultSet` reflejará esos cambios cuando se vuelva a consultar la primera fila. Por ejemplo, si se actualiza el nombre del primer registro en la base de datos mientras el programa espera, el nuevo nombre aparecerá en la salida.

Estos ejemplos demuestran cómo `TYPE_SCROLL_INSENSITIVE` no refleja cambios en la base de datos después de su creación, mientras que `TYPE_SCROLL_SENSITIVE` sí lo hace.

El argumento `ResultSet.Concurrency` indica la concurrencia del `ResultSet`:

- `ResultSet.CONCUR_READ_ONLY`: solo lectura. Es el valor por defecto.
- `ResultSet.CONCUR_UPDATABLE`: permite modificar los datos almacenados en el `ResultSet` para luego aplicar los cambios sobre la base de datos (más adelante se verá cómo).

Importante El `ResultSet` por defecto que se obtiene con `createStatement()` sin argumentos es el mismo que con `createStatement(ResultSet.TYPE_FORWARD_ONLY, ResultSet.CONCUR_READ_ONLY)`.

6. Consultas (Query)

6.1. Navegación de un `ResultSet`

Como ya se ha visto, en un objeto `ResultSet` se encuentran los resultados de la ejecución de una sentencia SQL. Por lo tanto, un objeto `ResultSet` contiene las filas que satisfacen las condiciones de una sentencia SQL, y ofrece métodos de navegación por los registros como `next()` que desplaza el curso al siguiente registro del `ResultSet`.

Además de este método de desplazamiento básico, existen otros de desplazamiento libre que podremos utilizar siempre y cuando el `ResultSet` sea de tipo `ResultSet.TYPE_SCROLL_INSENSITIVE` o `ResultSet.TYPE_SCROLL_SENSITIVE` como se ha dicho antes.

Algunos de estos métodos son:

- `void beforeFirst()` : mueve el cursor antes de la primera fila.
- `boolean first()` : mueve el cursor a la primera fila.
- `boolean next()` : mueve el cursor a la siguiente fila. Permitido en todos los tipos de `ResultSet`.
- `boolean previous()` : mueve el cursor a la fila anterior.
- `boolean last()` : mueve el cursor a la última fila.
- `void afterLast()` : mover el cursor después de la última fila.
- `boolean absolute(int row)` : posiciona el cursor en el número de registro indicado. Hay que tener en cuenta que el primer registro es el 1, no el cero.

Ejemplo

`absolute(n)`

`absolute(7)` desplazará el cursor al séptimo registro. Si valor es negativo se posiciona en el número de registro indicado pero empezando a contar desde el final (el último es el -1). Por ejemplo si tiene 10 registros y llamamos `absolute(-2)` se desplazará al registro número 9.

- `boolean relative(int registros)` : desplaza el cursor un número relativo de registros, que puede ser positivo o negativo.

Ejemplo

`relative(n)`

Si el cursor está en el registro 5 y llamamos a `relative(10)` se desplazará al registro número 15. Si luego llamamos a `relative(-4)` se desplazará al registro 11.

Los métodos que devuelven un tipo `boolean` devolverán `true` si ha sido posible mover el cursor a un registro válido, y `false` en caso contrario, por ejemplo si no tiene ningún registro o hemos saltado a un número de registro que no existe.

Todos estos métodos pueden producir una excepción de tipo `SQLException`.

También existen otros métodos relacionados con la posición del cursor.

- `int getRow()` : devuelve el número de registro actual. Cero si no hay registro actual.
- `boolean isBeforeFirst()` : devuelve `true` si el cursor está antes del primer registro.
- `boolean isFirst()` : devuelve `true` si el cursor está en el primer registro.
- `boolean isLast()` : devuelve `true` si el cursor está en el último registro.
- `boolean isAfterLast()` : devuelve `true` si el cursor está después del último registro.

6.2. Obteniendo datos del `ResultSet`

Los métodos `getXXX()` ofrecen los medios para recuperar los valores de las columnas (campos) de la fila (registro) actual del `ResultSet`. No es necesario que las columnas sean obtenidas utilizando un orden determinado.

Para designar una columna podemos utilizar su nombre o bien su número (empezando por 1).

Por ejemplo si la segunda columna de un objeto `ResultSet` se llama `titulo` y almacena datos de tipo `String`, se podrá recuperar su valor de las dos formas siguientes:

```
1 // rs es un objeto ResultSet
2 String valor = rs.getString(2);
3 String valor = rs.getString("titulo");
```

Importante

Es importante tener en cuenta que las columnas se numeran de izquierda a derecha y que la primera es la número 1, no la cero. También que las columnas no son case sensitive, es decir, no distinguen entre mayúsculas y minúsculas.

Ejemplo

La información referente a las columnas de un `ResultSet` se puede obtener llamando al **método** `getMetaData()` que devolverá un objeto `ResultSetMetaData` que contendrá el número, tipo y propiedades de las columnas del `ResultSet`.

```
1 package es.martinezpenya.ejemplos.UD10._06_Consultas;
2
3 import es.martinezpenya.ejemplos.UD10._03_Patrones._02_PoolObject.HikariCPSingleton;
4
5 import java.sql.*;
6
7 public class EjemploResultSetMetaData {
8     public static void main(String[] args) {
9         try (Connection con = HikariCPSingleton.getConnection();
10             Statement stmt = con.createStatement();
11             ResultSet rs = stmt.executeQuery("SELECT id, episode, title FROM
12 films")) {
13
14             // Obtener metadata del ResultSet
15             ResultSetMetaData rsmd = rs.getMetaData();
16
17             // Obtener el número de columnas
18             int columnCount = rsmd.getColumnCount();
19             System.out.println("Número de columnas: " + columnCount);
20
21             // Listar las columnas de detalles
22             for (int i = 1; i <= columnCount; i++) {
23                 String columnName = rsmd.getColumnName(i);
24                 String columnType = rsmd.getColumnTypeName(i);
25                 int columnDisplaySize = rsmd.getColumnDisplaySize(i);
26                 boolean isNullable = rsmd.isNullable(i) ==
27                 ResultSetMetaData.columnNullable;
28
29                 System.out.println("Columna " + i + ":");
30                 System.out.println("  Nombre: " + columnName);
31                 System.out.println("  Tipo: " + columnType);
32                 System.out.println("  Tamaño display: " + columnDisplaySize);
33                 System.out.println("  Nullable: " + isNullable);
34             }
35
36             // Iterar sobre el conjunto de resultados
37             while (rs.next()) {
38                 for (int i = 1; i <= columnCount; i++) {
39                     System.out.print(rs.getString(i) + " ");
40                 }
41                 System.out.println();
42             }
43
44             } catch (SQLException ex) {
45                 System.out.println("Error de SQL: " + ex.getMessage());
46             }
47     }
48 }
```

Esto provocará la siguiente salida por pantalla:

```
1 Número de columnas: 3
```



```

2  Columna 1:
3  Nombre: id
4  Tipo: INT
5  Tamaño display: 10
6  Nullable: false
7  Columna 2:
8  Nombre: episode
9  Tipo: VARCHAR
10 Tamaño display: 12
11 Nullable: true
12 Columna 3:
13 Nombre: title
14 Tipo: VARCHAR
15 Tamaño display: 30
16 Nullable: true
17 1 Episode I The Phantom Menace
18 2 Episode II Attack of the Clones
19 3 Episode III Revenge of the Sith
20 4 Episode IV A New Hope
21 5 Episode V The Empire Strikes Back
22 6 Episode VI Return of the Jedi

```

Si conocemos el nombre de una columna, pero no su índice, el método `findColumn()` puede ser utilizado para obtener el número de columna, pasándole como argumento un objeto `String` que sea el nombre de la columna correspondiente, este método nos devolverá un entero que será el índice correspondiente a la columna.

6.3. Tipos de datos y conversiones

Cuando se lanza un método `getXXX()` determinado sobre un objeto `ResultSet` para obtener el valor de un campo del registro actual, el driver JDBC convierte el dato que se quiere recuperar al tipo Java especificado y entonces devuelve un valor Java adecuado. Por ejemplo si utilizamos el método `getString()` y el tipo del dato en la base de datos es `VARCHAR`, el driver JDBC convertirá el dato `VARCHAR` de tipo SQL a un objeto `String` de Java.

Algo parecido sucede con otros tipos de datos SQL como por ejemplo `DATE`. Podremos acceder a él tanto con `getDate()` como con `getString()`. La diferencia es que el primero devolverá un objeto Java de tipo `Date` y el segundo devolverá un `String`.

Siempre que sea posible el driver JDBC convertirá el tipo de dato almacenado en la base de datos al tipo solicitado por el método `getXXX()`, pero hay conversiones que no se pueden realizar y lanzarán una excepción, como por ejemplo si intentamos hacer un `getInt()` sobre un campo que no contiene un valor numérico.

6.4. Sentencias que no devuelven datos

Las ejecutamos con el método `executeUpdate()`. Serán todas las sentencias SQL **excepto el SELECT**, que es la de consulta. Es decir, nos servirá para las siguientes sentencias:

- Sentencias que cambian las estructuras internas de la BD donde se guardan los datos (instrucciones conocidas con las siglas **DDL**, del inglés **Data Definition Language**), como por ejemplo `CREATE TABLE`, `CREATE VIEW`, `ALTER TABLE`, `DROP TABLE`, ...
- Sentencias para otorgar permisos a los usuarios existentes o crear otros nuevos (subgrupo de instrucciones conocidas como **DCL** o **Data Control Language**), como por ejemplo `GRANT`.
- Y también las sentencias para modificar los datos guardados utilizando las instrucciones `INSERT`, `UPDATE` y `DELETE`.

Aunque se trata de sentencias muy dispares, desde el punto de vista de la comunicación con el SGBD se comportan de manera muy similar, siguiendo el siguiente patrón:

1. **Instanciación del `Statement`** a partir de una conexión activa.
2. **Ejecución de una sentencia SQL** pasada por parámetro al método `executeUpdate()`.
3. **Cierre del objeto `Statement`** instanciado.

Miremos este ejemplo, en el que

Ejemplo

Vamos a crear una tabla muy sencilla en la Base de Datos MySQL

```

1  [...]
2  public void createTable() throws SQLException{
3      Statement st = con.createStatement();
4      st.executeUpdate("CREATE TABLE T1 (c1 varchar(50))");
5      st.close();
6  }
7  [...]

```

6.5. Asegurar la liberación de recursos

Las instancias de `Connection` y las de `Statement` guardan, en memoria, mucha información relacionada con las ejecuciones realizadas. Además, mientras continúan activas mantienen en el SGBD una sesión abierta, que supondrá un conjunto importante de recursos abiertos, destinados a servir de forma eficiente las peticiones de los clientes. Es importante cerrar estos objetos para liberar recursos tanto del cliente como del servidor.

Si en un mismo método debemos cerrar un objeto `Statement` y el `Connection` a partir del cual la hemos creado, se deberá cerrar primero el `Statement` y después el `Connection`. Si lo hacemos al revés, cuando intentamos cerrar el `Statement` nos saltará una excepción de tipo `SQLException`, ya que el cierre de la conexión le habría dejado inaccesible.

Además de respetar el orden, asegurar la liberación de los recursos situando las operaciones de cierre dentro de un bloque `finally`. De este modo, aunque se produzcan errores, no se dejarán de ejecutar las instrucciones de cierre.

Hay que tener en cuenta todavía un detalle más cuando sea necesario realizar el cierre de varios objetos a la vez. En este caso, aunque las situamos una tras otra, todas las instrucciones de cierre dentro del bloque `finally`, no sería suficiente garantía para asegurar la ejecución de todos los cierres, ya que, si mientras se produce el cierre de uno de los objetos se lanza una excepción, los objetos invocados en una posición posterior a la del que se ha producido el error no se cerrarán.

La solución de este problema pasa por evitar el lanzamiento de cualquier excepción durante el proceso de cierre. Una posible forma es encapsular cada cierre entre sentencias `try-catch` dentro del `finally`.

Ejemplo

Aquí tenéis un ejemplo completo:

```

1  package es.martinezpenya.ejemplos.UD10._06_Consultas;
2
3  import es.martinezpenya.ejemplos.UD10._03_Patrones._02_PoolObject.HikariCPSingleton;
4
5  import java.sql.Connection;
6  import java.sql.ResultSet;
7  import java.sql.SQLException;
8  import java.sql.Statement;
9
10 public class LiberacionRecursos {
11     static Statement stmt = null;
12     static ResultSet rs = null;
13     static Connection con;
14
15     public static void main(String[] args) {
16         try {
17             con = HikariCPSingleton.getConnection();
18             stmt = con.createStatement();
19             rs = stmt.executeQuery("SELECT id, name, hair_color FROM characters");
20
21             while (rs.next()) {
22                 System.out.print(rs.getInt(1) + "\t");
23                 System.out.print(rs.getString(2) + "\t");
24                 System.out.println(rs.getString(3));
25             }
26         } catch (SQLException ex) {
27             System.out.println("Error de SQL: " + ex.getMessage());
28         } finally {
29             try {
30                 //Siempre se debe cerrar todo lo abierto
31                 if (rs != null) {
32                     rs.close();

```

```

33     }
34     } catch (java.sql.SQLException ex) {
35         System.out.println("ERROR: " + ex.getMessage());
36     }
37     try {
38         //Siempre se debe cerrar todo lo abierto
39         if (stmt != null) {
40             stmt.close();
41         }
42     } catch (java.sql.SQLException ex) {
43         System.out.println("ERROR: " + ex.getMessage());
44     }
45 }
46 }
47 }

```

De todos modos recuerda que lo más fácil es usar `try-with-resources` y que se encargue Java de cerrar los recursos que hemos usado dentro del `try`.

```

1  package es.martinezpenya.ejemplos.UD10._06_Consultas;
2
3  import es.martinezpenya.ejemplos.UD10._03_Patrones._02_PoolObject.HikariCPSingleton;
4
5  import java.sql.Connection;
6  import java.sql.ResultSet;
7  import java.sql.SQLException;
8  import java.sql.Statement;
9
10 public class LiberacionRecursosTryWithResources {
11     public static void main(String[] args) {
12         try (Connection con = HikariCPSingleton.getConnection();
13             Statement stmt = con.createStatement();
14             ResultSet rs = stmt.executeQuery("SELECT id, name, hair_color FROM
characters");) {
15
16             while (rs.next()) {
17                 System.out.print(rs.getInt(1) + "\t");
18                 System.out.print(rs.getString(2) + "\t");
19                 System.out.println(rs.getString(3));
20             }
21         } catch (SQLException ex) {
22             System.out.println("Error de SQL: " + ex.getMessage());
23         }
24     }
25 }

```

Comparación y Recomendación

- **Claridad y Simplicidad:** `try-with-resources` es más claro y conciso, reduciendo la cantidad de código y evitando errores comunes al cerrar recursos manualmente.
- **Seguridad:** `try-with-resources` garantiza que los recursos se cierren en el orden correcto y automáticamente, incluso si ocurre una excepción.
- **Compatibilidad:** `try-with-resources` requiere Java 7 o superior, pero dado que las versiones anteriores de Java ya no son compatibles en la mayoría de los entornos, esto no suele ser un problema.

7. Modificación (update)

Para poder modificar los datos que contiene un `ResultSet` necesitamos un `ResultSet` de tipo modificable. Para ello debemos utilizar la constante `ResultSet.CONCUR_UPDATABLE` al llamar al método `createStatement()` como se ha visto antes.

Para modificar los valores de un registro existente se utilizan una serie de métodos `updateXXX()` de `ResultSet`. Las XXX indican el tipo del dato y hay tantos distintos como sucede con los métodos `getXXX()` de este mismo interfaz: `updateString()`, `updateInt()`, `updateDouble()`, `updateDate()`, etc.

La diferencia es que **los métodos `updateXXX()` necesitan dos argumentos**:

- La columna que deseamos actualizar (por su nombre o por su número de columna).
- El valor que queremos almacenar en dicha columna (del tipo que sea).

Por ejemplo para modificar el campo `edad` almacenando el entero `28` habría que llamar al siguiente método, suponiendo que `rs` es un objeto `ResultSet`:

```
1 | rs.updateInt("edad", 28);
```

También podría hacerse de la siguiente manera, suponiendo que la columna `edad` es la segunda:

```
1 | rs.updateInt(2, 28);
```

Los métodos `updateXXX()` no devuelven ningún valor (son de tipo `void`). Si se produce algún error se lanzará una `SQLException`.

Posteriormente hay que **llamar a `updateRow()` para que los cambios realizados se apliquen sobre la base de datos**. El *Driver JDBC* se encargará de ejecutar las sentencias SQL necesarias. Esta es una característica muy potente ya que nos facilita enormemente la tarea de modificar los datos de una base de datos.

En resumen, el proceso para realizar la modificación de una fila de un `ResultSet` es el siguiente:

1. **Desplazamos el cursor** al registro que queremos modificar.
2. Llamamos a todos los métodos `updateXXX(...)` que necesitemos.
3. Llamamos a `updateRow()` para que los cambios se apliquen a la base de datos.

Importante hay que llamar a `updateRow()` antes de desplazar el cursor. Si desplazamos el cursor antes de llamar a `updateRow()`, se perderán los cambios.

Ejemplo Si queremos **cancelar las modificaciones de un registro del `ResultSet`** podemos llamar a `cancelRowUpdates()`, que cancela todas las modificaciones realizadas sobre el registro actual.

Si ya hemos llamado a `updateRow()` el método `cancelRowUpdates()` no tendrá ningún efecto.

Ejemplo El siguiente código de ejemplo muestra cómo modificar el campo `title` del último registro de un `ResultSet` que contiene el resultado de una `SELECT` sobre la tabla de `films`:

```
1 | package es.martinezpenya.ejemplos.UD10._07_Modificacion;
2 |
3 | import es.martinezpenya.ejemplos.UD10._03_Patrones._02_PoolObject.HikariCPSingleton;
4 |
5 | import java.sql.*;
6 |
7 | public class EjemploUpdateRow {
8 |
9 |     public static void main(String[] args) {
10 |         String sql = "SELECT * FROM films";
11 |         try (Connection con = HikariCPSingleton.getConnection());
12 |             // Creamos un Statement scrollable y modificable
```

```
13         Statement st = con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,  
14         ResultSet.CONCUR_UPDATABLE);  
15         // Ejecutamos un SELECT y obtenemos la tabla films en un ResultSet  
16         ResultSet rs = st.executeQuery(sql); {  
17         // Nos movemos al último registro y lo actualizamos  
18         rs.last();  
19         rs.updateString(3, "Return Of The Jedi"); // Ponemos todas las palabras  
20         en mayúsculas  
21         rs.updateRow();  
22     } catch (SQLException ex) {  
23         System.out.println("Error de SQL: " + ex.getMessage());  
24     }  
25 }
```

8. Inserción (`insert`)

Para insertar nuevos registros necesitaremos utilizar, al menos, estos dos métodos:

- `void moveToInsertRow()` : desplaza el cursor al *registro de inserción*. Es un registro especial utilizado para insertar nuevos registros en el `ResultSet`. Posteriormente tendremos que llamar a los métodos `updateXXX()` ya conocidos para establecer los valores del registro de inserción. Para finalizar hay que llamar a `insertRow()`.
- `void insertRow()` : inserta el *registro de inserción* en el `ResultSet`, pasando a ser un registro normal más, y también lo inserta en la base de datos.

Ejemplo El siguiente código inserta un nuevo registro en la tabla `films` :

```

1 package es.martinezpenya.ejemplos.UD10._08_Insercion;
2
3 import es.martinezpenya.ejemplos.UD10._03_Patrones._02_PoolObject.HikariCPSingleton;
4
5 import java.sql.Connection;
6 import java.sql.ResultSet;
7 import java.sql.SQLException;
8 import java.sql.Statement;
9
10 public class EjemploMoveToInsertRow {
11     public static void main(String[] args) {
12         String sql = "SELECT * FROM films";
13         try (Connection con = HikariCPSingleton.getConnection();
14             Statement st = con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
15             ResultSet.CONCUR_UPDATABLE);
16             ResultSet rs = st.executeQuery(sql);) {
17             // Creamos un nuevo registro y lo insertamos
18             rs.moveToInsertRow();
19             rs.updateInt(1, 7);
20             rs.updateString(2, "Episode VII");
21             rs.updateString(3, "The force awakens");
22             rs.insertRow();
23         } catch (SQLException ex) {
24             System.out.println("Error de SQL: " + ex.getMessage());
25         }
26     }
27 }
```

Los campos cuyo valor no se haya establecido con `updateXXX()` tendrán un valor `NULL`. Si en la base de datos dicho campo no está configurado para admitir nulos se producirá una `SQLException`.

Tras insertar nuestro nuevo registro en el objeto `ResultSet` podremos volver a la anterior posición en la que se encontraba el cursor (antes de invocar `moveToInsertRow()`) llamando al método `moveToCurrentRow()`. Este método sólo se puede utilizar en combinación con `moveToInsertRow()`.

El siguiente ejemplo muestra como usar sentencias SQL para realizar la inserción de nuevos registros, ya veremos que esta no es la forma más aconsejable por seguridad.

```

1 package es.martinezpenya.ejemplos.UD10._08_Insercion;
2
3 import es.martinezpenya.ejemplos.UD10._03_Patrones._02_PoolObject.HikariCPSingleton;
4
5 import java.sql.*;
6
7 public class EjemploInsercionConsQL {
8
9     public static void insertFilmFijo(){
10         String sql = "INSERT INTO films (id, episode, title) VALUES (7, 'Episode VII', 'The force awakens')";
11         try (Connection con = HikariCPSingleton.getConnection();
12             // Creamos un Statement scrollable y modificable
13             Statement st = con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
14             ResultSet.CONCUR_UPDATABLE);
```

```

14         // Ejecutamos un SELECT y obtenemos la tabla films en un ResultSet
15         ResultSet rs = st.executeQuery(sql);) {
16     } catch (SQLException ex) {
17         System.out.println("ERROR al insertar el film: " + ex.getMessage());
18     }
19 }
20
21 public static void insertFilmParametros(int id, String episode, String title){
22     // OJO!! Forma no recomendada por problemas de seguridad (SQL Injection)
23     String sql = "INSERT INTO films (id, episode, title) VALUES (" + id + ", '" +
episode + "', '" + title + "')";
24     try (Connection con = HikariCPSingleton.getConnection();
25         // Creamos un Statement scrollable y modificable
26         Statement st = con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
ResultSet.CONCUR_UPDATABLE);
27         // Ejecutamos un SELECT y obtenemos la tabla films en un ResultSet
28         ResultSet rs = st.executeQuery(sql);) {
29     } catch (SQLException e) {
30         System.out.println("Se ha producido un error al insertar el film.
Mensaje: " + e.getMessage());
31     }
32 }
33
34 public static void main(String[] args) {
35     insertFilmFijo();
36     insertFilmParametros(8, "Episode VIII", "The last Jedi");
37 }
38 }

```

9. Borrado (delete)

Para eliminar un registro solo hay que desplazar el cursor al registro deseado y llamar al método:

- `void deleteRow()` : elimina el registro actual del `ResultSet` y también de la base de datos.

Ejemplo

El siguiente código borra el tercer registro de la tabla `films` :

```

1 package es.martinezpenya.ejemplos.UD10._09_Borrado;
2
3 import es.martinezpenya.ejemplos.UD10._03_Patrones._02_PoolObject.HikariCPSingleton;
4
5 import java.sql.*;
6
7 public class EjemploDeleteRow {
8
9     public static void main(String[] args) {
10         String sql = "SELECT * FROM films";
11         try (Connection con = HikariCPSingleton.getConnection();
12             Statement st = con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
13             ResultSet.CONCUR_UPDATABLE);
14             ResultSet rs = st.executeQuery(sql);) {
15             // Desplazamos el cursor al tercer registro
16             rs.absolute(3);
17             rs.deleteRow();
18         } catch (SQLException ex) {
19             System.out.println("Error de SQL: " + ex.getMessage());
20         }
21     }
22 }
```


10. Sentencias predefinidas

Para solucionar el problema de crear sentencias sql complejas, se utiliza `PreparedStatement`.

JDBC dispone de un objeto derivado del `Statement` que se llama `PreparedStatement`, al que se le pasa la sentencia SQL en el momento de crearlo, no en el momento de ejecutar la sentencia (como pasaba con `Statement`). Y además esta sentencia puede admitir parámetros, lo que nos puede ir muy bien en determinadas ocasiones.

De cualquier modo, `PreparedStatement` presenta ventajas sobre su antecesor `Statement` cuando nos toque trabajar con sentencias que se hayan de ejecutar varias veces. La razón es que cualquier sentencia SQL, cuando se envía al SGBD será compilada antes de ser ejecutada.

- Utilizando un objeto `Statement`, cada vez que hacemos una ejecución de una sentencia, ya sea vía `executeUpdate` o bien vía `executeQuery`, el SGBD la compilará, ya que le llegará en forma de cadena de caracteres.
- En cambio, al `PreparedStatement` la sentencia nunca varía y por lo tanto se puede compilar y guardar dentro del mismo objeto, por lo que las siguientes veces que se ejecute no habrá que compilarse. Esto reducirá sensiblemente el tiempo de ejecución.

En algunos sistemas gestores, además, usar `PreparedStatement` puede llegar a suponer más ventajas, ya que utilizan la secuencia de bytes de la sentencia para detectar si se trata de una sentencia nueva o ya se ha servido con anterioridad. De esta manera se propicia que el sistema guarde las respuestas en la memoria caché, de manera que se puedan entregar de forma más rápida.

La principal diferencia de los objetos `PreparedStatement` en relación a los `Statement`, es que en los primeros se les pasa la sentencia SQL predefinida en el momento de crearlo. Como la sentencia queda predefinida, ni los métodos `executeUpdate` ni `executeQuery` requerirán ningún parámetro. Es decir, justo al revés que en el `Statement`.

Los parámetros de la sentencia se marcarán con el símbolo de interrogación (?) Y se identificarán por la posición que ocupan en la sentencia, empezando a contar desde la izquierda a partir del número 1. El valor de los parámetros se asignará utilizando el método específico, de acuerdo con el tipo de datos a asignar. El nombre empezará por `set` y continuará con el nombre del tipo de datos (ejemplos: `setString`, `setInt`, `setLong`, `setBoolean` ...). Todos estos métodos siguen la misma sintaxis:

```
1 | setXXXX(<posiciónEnLaSentenciaSQL>, <valor>);
```

Ejemplo Este es el mismo método para insertar y eliminar un **film** pero usando `PreparedStatement`:

```
1 | package es.martinezpenya.ejemplos.UD10._10_Predefinidas;
2 |
3 | import java.sql.Connection;
4 | import java.sql.PreparedStatement;
5 | import java.sql.SQLException;
6 | import java.util.Scanner;
7 |
8 | import es.martinezpenya.ejemplos.UD10._03_Patrones._02_PoolObject.HikariCPSingleton;
9 |
10 | public class EjemploInsercionBorradoConPreparedStatement {
11 |
12 |     public static void insertFilmPrepared(int id, String episode, String title) {
13 |         String sql = "INSERT INTO films (id, episode, title) VALUES (?, ?, ?)";
14 |         try (Connection con = HikariCPSingleton.getConnection(); // Obtener conexión
15 |             del pool
16 |             PreparedStatement pst = con.prepareStatement(sql)) {
17 |             // Asignamos los valores a los parámetros
18 |             pst.setInt(1, id);
19 |             pst.setString(2, episode);
20 |             pst.setString(3, title);
21 |             // Ejecutamos la sentencia preparada sin peligro de SQL Injection
22 |             pst.executeUpdate();
23 |         } catch (SQLException e) {
24 |             System.out.println("Se ha producido un error al insertar el film.
25 |             Mensaje: " + e.getMessage());
26 |         }
27 |     }
28 | }
```

```

24     }
25 }
26
27 public static void deleteFilmPrepared(int id) {
28     String sql = "DELETE FROM films WHERE id = ?";
29     try (Connection con = HikariCPSingleton.getConnection(); // Obtener conexión
del pool
30         PreparedStatement pst = con.prepareStatement(sql)) {
31         // Asignamos el valor al parámetro
32         pst.setInt(1, id);
33         // Ejecutamos la sentencia preparada sin peligro de SQL Injection
34         pst.executeUpdate();
35     } catch (SQLException e) {
36         System.out.println("Se ha producido un error al borrar el film. Mensaje:
" + e.getMessage());
37     }
38 }
39
40 public static void main(String[] args) {
41     insertFilmPrepared(8, "Episode VIII", "The last Jedi");
42     Scanner sc = new Scanner(System.in);
43     System.out.println("Pausa, pulsa ENTER para borrar el film introducido....");
44     sc.nextLine();
45     deleteFilmPrepared(8);
46 }
47 }

```

10.1. Ventajas de `PreparedStatement` desde el Punto de Vista de Seguridad

`PreparedStatement` ofrece varias ventajas en términos de seguridad, principalmente al prevenir ataques de inyección SQL.

Prevención de Inyección SQL:

Definición La **inyección SQL** es un tipo de ataque en el que un atacante inserta o "inyecta" código SQL malicioso en una consulta a través de entradas de usuario. Esto puede permitir a un atacante ejecutar comandos SQL no autorizados, acceder a datos sensibles o manipular la base de datos de formas inesperadas.

Ejemplo de Inyección SQL:

Supongamos que tienes un código que construye una consulta SQL concatenando cadenas de texto:

```

1 | String query = "SELECT * FROM usuarios WHERE username = '" + username + "' AND password =
  | '" + password + "'";

```

Si un atacante proporciona un `username` como `' OR '1'='1'` y una `password` como `' OR '1'='1'`, la consulta resultante sería:

```

1 | SELECT * FROM usuarios WHERE username = '' OR '1'='1' AND password = '' OR '1'='1'

```

Esta consulta siempre devolvería todos los registros de la tabla `usuarios`, lo que podría permitir al atacante acceder a datos que no deberían estar disponibles.

11. Píldoras informáticas relacionadas

- [Curso Java. Acceso a BBDD. JDBC I. Vídeo 201](#)
- [Curso Java. Acceso a BBDD. JDBC II. Vídeo 202](#)
- [Curso Java. Acceso a BBDD. JDBC III. Vídeo 203](#)
- [Curso Java. Acceso a BBDD. JDBC IV. Vídeo 204](#)
- [Curso Java Acceso a BBDD. JDBC V. Consultas Preparadas. Vídeo 205](#)
- [Curso Java Acceso a BBDD. JDBC VI. Práctica guiada. Vídeo 206](#)
- [Curso Java Acceso a BBDD. JDBC VII. Práctica guiada II. Vídeo 207](#)
- [Curso Java Acceso a BBDD. JDBC VIII. Práctica guiada III. Vídeo 208](#)
- [Curso Actualizado de JDBC 2024 \(Makigas\)](#)

12. Fuentes de información

- [Wikipedia](#)
- [Programación \(Grado Superior\) - Juan Carlos Moreno Pérez \(Ed. Ra-ma\)](#)
- Apuntes IES Henri Matisse (Javi García Jimenez?)
- Apuntes AulaCampus
- [Apuntes José Luis Comesaña](#)
- [Apuntes IOC Programació bàsica \(Joan Arnedo Moreno\)](#)
- [Apuntes IOC Programació Orientada a Objectes \(Joan Arnedo Moreno\)](#)
- <https://arturoblasco.github.io/pr>