

AD03.- Manejo de conectores.

Caso práctico

Esta mañana **Ada** está reunida con **María** y **Juan**.

Están discutiendo unos detalles de un proyecto, en el que la empresa está embarcada desde hace tiempo, y quieren retomarlo porque lo tenían un poco aparcado.

En esta ocasión se trata de una **franquicia de inmobiliarias**. El gerente que gestiona las franquicias contactó, hace tiempo, con BK Programación para que le hicieran una página web, en la que colgar información sobre los inmuebles en venta o alquiler.

María participó entonces activamente en el proyecto, así como **Juan**, por la amplia experiencia de ambos en desarrollo web.

Desarrollaron en su momento lo más básico del proyecto, pero ahora, **el gerente de las inmobiliarias está apremiando a BK Programación** para que terminen el resto del proyecto. Sobre todo, necesita realizar un montón de consultas sobre la base de datos relacional donde guardan los datos de la inmobiliaria: clientes, inmuebles, ventas realizadas, etc.

Juan y **María** van a aprovechar a **Ana** y **Antonio** para que les echen una mano y avanzar rápidamente en el proyecto.



1.- Introducción.

Actualmente, las **bases de datos relacionales** constituyen el sistema de almacenamiento probablemente más extendido, aunque otros sistemas de almacenamiento de la información se estén abriendo paso poco a poco.

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



Diagrama de una tabla relacional. La tabla tiene varias columnas y filas. Las columnas están etiquetadas como 'Campos' y las filas como 'Registros'. Se muestran ejemplos de datos en las filas, como nombres, direcciones y números de teléfono.

[\(link: AD03 CONT R02 tabla.jpg.\)](#)

Una tabla o relación es una colección de objetos del mismo tipo (filas o tuplas).

En cada tabla de una base de datos se elige una **clave primaria** para identificar de manera unívoca a cada fila de la misma.

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)**, permite simplificar el acceso a bases de datos relacionales, 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 API's de bases de datos existentes.
- ✓ **Ser lo más sencillo posible**.

Autoevaluación

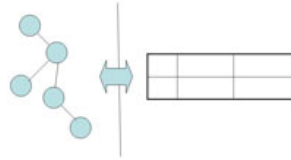
Señala si la afirmación siguiente es verdadera o falsa.

JDBC tiene como uno de sus principales objetivos facilitar el acceso a bases de datos relacionales.

☐ Verdadero ☐ Falso

1.1.- El desfase objeto-relacional.

El desfase objeto-relacional, también conocido como impedancia objeto-relacional, consiste en la **diferencia de aspectos que existen entre la programación orientada a objetos y la base de datos**. Estos aspectos se puede presentar en cuestiones como:



- ✓ **Lenguaje de programación:** el programador debe conocer el lenguaje de programación orientado a objetos (POO) y el lenguaje de acceso a datos.
- ✓ **Tipos de datos:** en las bases de datos relacionales siempre hay restricciones en el uso de tipos, mientras que la programación orientada a objetos utiliza tipos de datos mas complejos.
- ✓ **Paradigma de programación:** en el proceso de diseño y construcción del software se tiene que hacer una traducción del modelo orientado a objetos de clases al modelo Entidad-Relación (E/R) puesto que el primero maneja objetos y el segundo maneja tablas y tuplas (o filas), lo que implica que se tengan que diseñar dos diagramas diferentes para el diseño de la aplicación.

[\(link: AD03 CONT R03 Typemismatch.JPG\)](#)

El **modelo relacional trata con relaciones y conjuntos** debido a su **naturaleza matemática**. Sin embargo, **el modelo de POO trata con objetos y las asociaciones entre ellos**. Por esta razón, el problema entre estos dos modelos surge en el momento de querer persistir los objetos de negocio.

La escritura (y de manera similar la lectura) mediante JDBC implica:

- ✓ Abrir una conexión.
- ✓ Crear una sentencia en SQL.
- ✓ Copiar todos los valores de las propiedades de un objeto en la sentencia, ejecutarla y así almacenar el objeto.

Esto es sencillo para un caso simple, pero complicado si el objeto posee muchas propiedades, o bien se necesita almacenar un objeto que a su vez posee una colección de otros elementos. Se necesita crear mucho más código, además del tedioso trabajo de creación de sentencias SQL.

Este problema es lo que denominábamos **impedancia Objeto-Relacional**, o sea, el conjunto de dificultades técnicas que surgen cuando una base de datos relacional se usa en conjunto con un programa escrito en POO.

Como ejemplo de desfase objeto-relacional, podemos poner el siguiente: supón que tienes un objeto "Agenda Personal" con un atributo que sea una colección de objetos de la clase "Persona". Cada persona tiene un atributo "teléfono". Al transformar este caso a relacional, se ocuparía más de una tabla para almacenar la información, conllevando varias sentencias SQL y bastante código.

Autoevaluación

Di si la siguiente afirmación es verdadera o falsa:

El desfase objeto-relacional tiene que ver con que el paradigma de orientación a objetos tiene una naturaleza matemática y el modelo relacional no.

☐ Verdadero ☐ Falso

Caso práctico



María les pregunta a **Antonio** y a **Ana** -¿Estáis dispuestos a realizar consultas sobre bases de datos? Nos vais a tener que echar una mano a **Juan** y a mi para acabar pronto el trabajo. -¡Cuenta con ello! - responde **Antonio**. -Hacer consultas es lo que más me gusta, ¡soy una máquina en eso!

Ana comenta que a ella también le apasiona el tema, y que en el ciclo de DAM lo han estudiado muy bien. -¿Qué base de datos tienen implantada, qué protocolo de acceso hay que usar?

Hace años, cuando **Sun Microsystems** desarrolló Java, uno de los aspectos que tuvieron que pensar fue la manera de enfocar el acceso a datos, todo lo concerniente a los protocolos de acceso a bases de datos.

Debido a la confusión que había por la proliferación de API's propietarios de acceso a datos, Sun buscó los aspectos de éxito de un API de este tipo: ODBC (Open Database Connectivity).

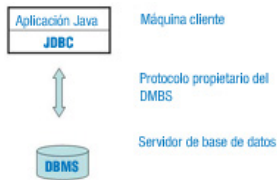
ODBC había sido desarrollado por Microsoft con la idea de tener un estándar para el acceso a bases de datos en entorno Windows.



Aunque la industria aceptó ODBC como medio principal para acceso a bases de datos en Windows, la verdad es que no se introduce bien en el mundo Java, debido a la complejidad que presenta ODBC, y que entre otras cosas ha impedido su transición fuera del entorno Windows.

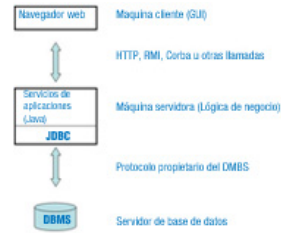
La idea en el desarrollo de JDBC era intentar ser tan sencillo como fuera posible, pero proporcionando a los desarrolladores la máxima flexibilidad.

2.1.- Arquitectura JDBC.



[\(link: AD03_CONT_R06_2capas.JPG.\)](#)

El API JDBC



[\(link: AD03_CONT_R07_3capas.JPG.\)](#)

soporta dos modelos de procesamiento para acceso a bases de datos: de dos y tres capas.

En el **modelo de dos capas**, una aplicación se comunica directamente a la fuente de datos. Esto necesita un conector JDBC que pueda comunicar con la fuente de datos específica a la que acceder.

Los comandos o instrucciones del usuario se envían a la base de datos y los resultados se devuelven al usuario. La fuente de datos puede estar ubicada en otra máquina a la que el usuario se conecte por red. A esto se denomina configuración cliente/servidor, con la máquina del usuario como cliente y la máquina que aloja los datos como servidor.

En el **modelo de tres capas**, los comandos se envían a una capa intermedia de servicios, la cual envía los comandos a la fuente de datos. La fuente de datos procesa los comandos y envía los resultados de vuelta a la capa intermedia, desde la que luego se le envían al usuario.

El API JDBC viene distribuido en dos paquetes:

- ✓ `java.sql`, dentro de `J2SE`.
- ✓ `javax.sql`, extensión dentro de `J2EE`.

Autoevaluación

Di si la siguiente afirmación es verdadera o falsa:

ODBC es lo mismo que JDBC, el primero para Windows y el segundo para Linux y Mac.

☐ Verdadero ☐ Falso

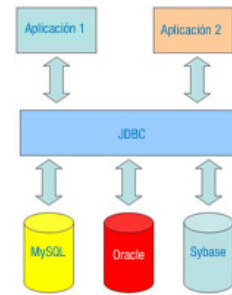
2.2.- Conectores o Drivers.

Un conector o driver es un conjunto de clases encargadas de implementar los interfaces del API y acceder a la base de datos.

Para poder conectarse a una base de datos y lanzar consultas, una aplicación necesita tener un driver adecuado. Un conector suele ser un fichero .jar que contiene una implementación de todas las interfaces del API JDBC.

Cuando se construye una aplicación de base de datos, **JDBC oculta lo específico de cada base de datos**, de modo que el programador se ocupe sólo de su aplicación.

El conector lo proporciona el fabricante de la base de datos o bien un tercero.



[\(link: AD03_CONT_R08_drivers.JPG.\)](#)

El código de nuestra aplicación no depende del driver, puesto que trabajamos mediante los paquetes java.sql y javax.sql.

JDBC ofrece las clases e interfaces para:

- ✓ Establecer una conexión a una base de datos.
- ✓ Ejecutar una consulta.
- ✓ Procesar los resultados.

Ejemplo:

```
// Establece la conexión
Connection con = DriverManager.getConnection("jdbc:oracle:thin:
    @localhost", "usuario", "password");

// Ejecuta la consulta
Statement stmt = (Statement) con.createStatement();
ResultSet rs = stmt.executeQuery("SELECT nombre, edad FROM Empleado");

// Recorre los resultados
while (rs.next()) {
    String nombre = rs.getString("nombre");
    int edad = rs.getInt("edad");
}
```

[\(link: AD03_CONT_R08_01_imgcod.JPG.\)](#)

[Código de la estructura para conectarnos, ejecutar consulta y procesar resultados. \(link: AD03_CONT_R08_02_codejecon.txt.\)](#) (0.01 MB)

En principio, todos los conectores deben ser compatibles con ANSI SQL-2 Entry Level (ANSI SQL-2 se refiere a los estándares adoptados por el American National Standards Institute en 1992. Entry Level se refiere a una lista específica de capacidades de SQL.) Los desarrolladores de drivers pueden establecer que sus conectores conocen estos estándares.

Hay **cuatro tipos de drivers JDBC**: Tipo 1, Tipo 2, Tipo 3 y Tipo 4, que veremos a continuación.

Para saber más

Base de datos con información sobre JDBC y tecnologías de base de datos de Oracle.

JDBC y otras tecnologías de Oracle ([link: https://www.oracle.com/es/database/technologies/appdev/jdbc.html](https://www.oracle.com/es/database/technologies/appdev/jdbc.html))

2.3.- Conectores tipo 1 y tipo 2.

Los conectores tipo 1 se denominan también JDBC-ODBC Bridge (puente JDBC-ODBC).

Proporcionan un puente entre el API JDBC y el API ODBC. El driver JDBC-ODBC Bridge traduce las llamadas JDBC a llamadas ODBC y las envía a la fuente de datos ODBC.

Como ventajas destacar:

- ✓ No se necesita un driver específico de cada base de datos de tipo ODBC.
- ✓ Está soportado por muchos fabricantes, por lo que tenemos acceso a muchas Bases de Datos.

Como desventajas señalar:

- ✓ Hay plataformas que no lo tienen implementado.
- ✓ El rendimiento no es óptimo ya que la llamada JDBC se realiza a través del puente hasta el conector ODBC y de ahí al interface de conectividad de la base de datos. El resultado recorre el camino inverso.
- ✓ Se tiene que registrar manualmente en el gestor de ODBC teniendo que configurar el DSN (Data Source Names, Nombres de fuentes de datos).

Este tipo de driver va incluido en el JDK.

Los conectores tipo 2 se conocen también como: API nativa

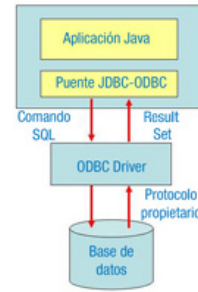
Convierten las llamadas JDBC a llamadas específicas de la base de datos para bases de datos como SQL Server, Informix, Oracle, o Sybase.

El conector tipo 2 se comunica directamente con el servidor de bases de datos, por lo que es necesario que haya código en la máquina cliente.

Como ventaja, este conector destaca por ofrecer un rendimiento notablemente mejor que el JDBC-ODBC Bridge.

Como inconveniente, señalar que la librería de la bases de datos del vendedor necesita cargarse en cada máquina cliente. Por esta razón los drivers tipo 2 no pueden usarse para Internet.

Los drivers Tipo 1 y 2 utilizan código nativo vía JNI, por lo que son más eficientes.



[.\(link: AD03_CONT_R09_drivertipo1.JPG.\)](#)



[.\(link: AD03_CONT_R10_drivertipo2.JPG.\)](#)

Para saber más

En este enlace puede ver información sobre JNI y sobre DSN.

JNI ([link: http://es.wikipedia.org/wiki/Java_Native_Interface](http://es.wikipedia.org/wiki/Java_Native_Interface))

DSN ([link: http://esedesol.sedesol.gob.mx/extranet/manuales/Data%20Source%20Name.pdf](http://esedesol.sedesol.gob.mx/extranet/manuales/Data%20Source%20Name.pdf)) (0.90 KB)

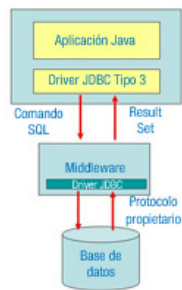
Autoevaluación

Señala si la siguiente afirmación es verdadera o falsa:

El problema de los conectores tipo 2 es que no pueden utilizarse para Internet.

☐ Verdadero ☐ Falso

2.4.- Conectores tipo 3 y tipo 4.



([link: AD03_CONT_R11_drivertipo3.JPG](#)).

Tipo 3: JDBC-Net pure Java driver.

Tiene una aproximación de tres capas. Las peticiones JDBC a la base de datos se pasan a través de la red al servidor de la capa intermedia (**middleware**). Este servidor traduce este protocolo independiente del sistema gestor a protocolo específico del sistema gestor y se envía a la base de datos. Los resultados se mandan de vuelta al middleware y se enrutan al cliente.

Es útil para aplicaciones en Internet.

([link: AD03_CONT_R12_drivertipo4.JPG](#)).

Este driver está basado en servidor, por lo que no se necesita ninguna librería de base de datos en las máquinas clientes.

Normalmente, un driver de tipo 3 proporciona soporte para **balanceo de carga**, funciones avanzadas de administrador de sistemas tales como auditoría, etc.

Tipo 4: Protocolo nativo.

En este caso se trata de conectores que convierten directamente las llamadas JDBC al protocolo de red usando por el sistema gestor de la base de datos. Esto permite una llamada directa desde la máquina cliente al servidor del sistema gestor de base de datos y es una solución excelente para acceso en intranets.

Como ventaja se tiene que no es necesaria traducción adicional o capa middleware, lo que mejora el rendimiento, siendo éste mejor que en el caso de los tipos 1 y 2.

Además, no se necesita instalar ningún software especial en el cliente o en el servidor.

Como inconveniente, de este tipo de conectores, el usuario necesita un driver diferente para cada base de datos.

Un ejemplo de este tipo de conector es **Oracle Thin**.

Los tipos 3 y 4 son la mejor forma de acceder a bases de datos JDBC.



3.- Conexión a una base de datos.

Caso práctico



Juan y María están trabajando por parejas: **Juan** con **Antonio** y **María** con **Ana**, de esta manera podrán instruirles lo antes posible en el proyecto que llevan entre manos, y cuya base de datos está en MySQL. Juan le explica a Antonio cómo conectarse a la base de datos relacional que usan desde la aplicación Java. María, hace lo propio con Ana.

Para acceder a una base de datos y así poder operar con ella, lo primero que hay que hacer es conectarse a dicha base de datos.

En Java, para establecer una conexión con una base de datos podemos utilizar el método `getConnection()` de la clase `DriverManager`. Este método recibe como parámetro la URL de JDBC que identifica a la base de datos con la que queremos realizar la conexión.

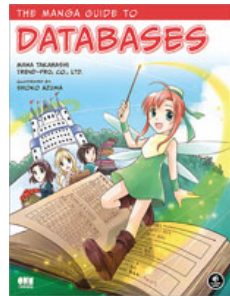
La ejecución de este método devuelve un objeto `Connection` que representa la conexión con la base de datos.

Cuando se presenta con una URL específica, `DriverManager` itera sobre la colección de drivers registrados hasta que uno de ellos reconoce la URL especificada. Si no se encuentra ningún conector adecuado, se lanza una `SQLException`.

Veamos un ejemplo con comentarios, para conectarnos a una base de datos MySQL:

```
public void main(String[] args) {  
    try {  
        // Importar el driver de MySQL  
        Class.forName("com.mysql.jdbc.Driver");  
  
        // Cadena de conexión para conectarse con MySQL en localhost.  
        // Reemplazamos la base de datos llamada "test"  
        // por nombre y contraseña del servidor de MySQL con el mismo  
        String connectionString = "jdbc:mysql://localhost/test?u=" +  
            "usuario:mysql:mysql@localhost";  
  
        // Obtenemos la conexión  
        Connection con = DriverManager.getConnection(connectionString);  
    } catch (SQLException ex) {  
        System.out.println("Error: " + ex.getMessage());  
    } catch (ClassNotFoundException ex) {  
        System.out.println("Error: " + ex.getMessage());  
    }  
}
```

([link: AD03_CONT_R14_00_codigoconec.jpg.](#))



3.1.- Instalar el conector de la base de datos.

Para que podamos ejecutar el código anterior, necesitamos instalar el conector de la base de datos.

Entre nuestra aplicación Java y el Sistema Gestor de Base de Datos (SGBD), se intercala el conector JDBC. Este conector es el que implementa la funcionalidad de las clases de acceso a datos y proporciona la comunicación entre el API JDBC y el SGBD.

La función del conector es traducir los comandos del API JDBC al protocolo nativo del SGBD.

En la siguiente presentación vamos a ver cómo descargarnos el conector que necesitamos para trabajar con MySQL. Como verás, tan sólo consiste en descargar un archivo, descomprimirlo y desde NetBeans añadir el fichero .jar que constituye el driver que necesitamos.



3.2.- Pool de conexiones.

Acabamos de ver cómo se realiza una conexión a una base de datos. En ocasiones, sobre todo cuando se trabaja en el ámbito de las aplicaciones distribuidas, en entornos web, es recomendable gestionar las conexiones de otro modo.

La explicación es que: abrir una conexión, realizar las operaciones necesarias con la base de datos, y cerrar la conexión; puede ser lento en entornos web si lo hacemos como hemos visto.

En las aplicaciones en las que es necesario el acceso concurrente y masivo a una base de datos, como es el caso de las aplicaciones web, es necesario disponer de varias conexiones. El proceso de creación y destrucción de una conexión a una base de datos es costoso e influye sensiblemente en el rendimiento de una aplicación. Es mejor en estos casos, por tanto, abrir una o más conexiones y mantenerlas abiertas.

Las versiones recientes de JDBC proporciona un pool de conexiones que funciona de forma transparente. Al iniciar un servidor **Java EE**, automáticamente el pool de conexiones crea un número de conexiones físicas iniciales.

Cuando un objeto Java del servidor J2EE necesita una conexión a través del método `dataSource.getConnection()`, la fuente de datos `javax.sql.DataSource` habla con el pool de conexiones y éste le entrega una conexión lógica `java.sql.Connection`. Esta conexión lógica la recibe por último, el objeto Java.

Cuando un objeto Java del servidor Java EE desea cerrar una conexión a través del método `connection.close()`, la fuente de datos `javax.sql.DataSource` habla con el pool de conexiones y le devuelve la conexión lógica en cuestión.

Si hay un pico en la demanda de conexiones a la base de datos, el pool de conexiones de forma transparente crea más conexiones físicas de objetos tipo `Connection`. Si por el contrario las conexiones a la base de datos disminuyen, el pool de conexiones, también de forma transparente elimina conexiones físicas de objetos de tipo `Connection`.



Debes conocer

El siguiente enlace es bastante interesante, mostrando ejemplos de código.

Ejemplo de pool de conexiones. (*link: http://chuwiki.chuidiang.org/index.php?title=Pool_de_conexiones*)

4.- Creación de la base de datos.

Caso práctico



Antonio le ha pedido a **Juan** participar en uno de los proyectos que desarrolla en la empresa. Juan es el responsable de un proyecto de aplicaciones para oficinas de farmacia. **Juan** no está muy convencido al principio, porque Antonio está bastante centrado en el proyecto de las inmobiliarias y piensa que quizás sea demasiada carga para él, pero **María** interviene en la conversación y anima a **Juan** a que le eche una mano a **Antonio**. **Juan** va a crear una base de datos con Microsoft Access para guardar la información de los productos de la farmacia.

—**Antonio**, presta atención a lo que te digo —le dice **Juan**.

Juan le explica a Antonio, que una base de datos puede crearse utilizando las herramientas proporcionadas por el fabricante de la base de datos, o por medio de sentencias SQL desde un programa Java. Pero normalmente es el administrador de la base de datos, a través de las herramientas que proporcionan el sistema gestor, el que creará la base de datos. No todos los conectores JDBC soportan la creación de la base de datos mediante el lenguaje de definición de datos (**DDL**). Es decir, la sentencia CREATE DATABASE no es parte del estándar SQL, sino que es dependiente del sistema gestor de la base de datos.



Así pues, mediante JDBC podemos conectarnos y manipular bases de datos: crear tablas, modificarlas, borrarlas, añadir datos en las tablas, etc. Pero la creación en sí de la base de datos la hacemos, generalmente, con la herramienta específica para ello.

Normalmente, cualquier sistema gestor de bases de datos incluye asistentes gráficos para crear la base de datos, sus tablas, claves, y todo lo necesario.

También, como en el caso de MySQL, o de Oracle, y la mayoría de sistemas gestores de bases de datos, se puede crear la base de datos, desde la línea de comandos de MySQL o de Oracle, con las sentencias SQL apropiadas.

A continuación puedes descargar la una pequeña base de datos en Access.

[Base de datos Farmacia. \(link: AD03_R19_01_bdfarmacia.zip.\)](#) (10 KB)

Para saber más

Hay herramientas que te permiten visualizar (y ciertas operaciones más) sobre bases de datos Access, sin necesidad de tener instalado Access en el equipo.

Visor de bases de datos Access. (link: http://www.alexnolan.net/software/mdb_viewer_plus.htm)

5.- Operaciones: ejecución de consultas.

Caso práctico



Juan y Antonio están inmersos en la creación de consultas para los informes que la aplicación de farmacias debe aportar a los usuarios de la misma. Realmente es **Juan** el que está realizándolas, pero Antonio empieza a comprender el asunto y quiere participar más.

Para operar con una base de datos, ejecutando las consultas necesarias, nuestra aplicación deberá hacer:

- ✓ **Cargar** el driver necesario para comprender el protocolo que usa la base de datos en cuestión.
- ✓ **Establecer** una conexión con la base de datos.
- ✓ **Enviar** consultas SQL y procesar el resultado.
- ✓ **Liberar** los recursos al terminar.
- ✓ **Gestionar** los errores que se puedan producir.



Podemos utilizar los siguientes tipos de sentencias:

- ✓ Statement: para sentencias sencillas en SQL.
- ✓ PreparedStatement: para consultas preparadas, como por ejemplo las que tienen parámetros.
- ✓ CallableStatement: para ejecutar procedimientos almacenados en la base de datos.

El API JDBC distingue dos tipos de consultas:

- ✓ Consultas: SELECT
- ✓ Actualizaciones: INSERT, UPDATE, DELETE, sentencias DDL.

Veamos a continuación cómo realizar una consulta en la base de datos que creamos en el apartado anterior, la base de datos **farmacia.mdb**.

En este caso utilizaremos un driver JDBC de Java open-source llamado UCanAccess.

Este driver es válido para sistemas operativos Windows y Linux y **no es necesario utilizar ODBC**.

Autoevaluación

Respecto a las consultas con JDBC, señala la respuesta correcta:

- ☐ (link:)
Podemos crear una tabla con una consulta de actualización.
- ☐ (link:)
Hay que liberar los recursos antes de cargar el driver de la base de datos.
- ☐ (link:)
No es posible realizar consultas si no usamos un puente ODBC.
- ☐ (link:)
Todas son correctas.

5.1.- Ejemplo: consultas con MS-Access (I).

Como se ha comentado anteriormente **UCanAccess** es un driver JDBC puro de Java. Utiliza los paquetes Jackcess y HSQLDB para realizar sus operaciones y necesita incluir los siguientes componentes en nuestro *Build Path*.

Cuando descargues el fichero de distribución y lo descomprimas encontrarás algo parecido a esto:

```
ucanaccess-4.0.4.jar
/lib/
  commons-lang-2.6.jar
  commons-logging-1.1.3.jar
  hsqldb.jar
  jackcess-2.1.11.jar
```

Es necesario añadir los 5 ficheros .JAR a nuestro proyecto.

Posteriormente, solamente necesitamos **realizar la conexión** a la base de datos para comenzar a trabajar con ella. La clase DriverManager define el método getConnection para crear una conexión a una base de datos. Este método toma como parámetro una URL JDBC donde se indica el sistema gestor y la base de datos. Opcionalmente, y dependiendo del sistema gestor, habrá que especificar el login y password para la conexión. En nuestro caso, el código para establecer una conexión quedaría del siguiente modo:

```
Connection conn=DriverManager.getConnection("jdbc:ucanaccess://farmacia.mdb");
```

Ahora ya podemos realizar las consultas que deseemos.

Para saber más

En el siguiente enlace puedes consultar más información acerca de UCanAccess. (*link: <http://ucanaccess.sourceforge.net/site.html>*)

5.2.- Ejemplo: consultas con MS-Access (II).

Las consultas que se realizan a una base de datos se realizan utilizando objetos de las clases Statement y PreparedStatement. Estos objetos se crean a partir de una conexión.

```
Statement stmt = con.createStatement();
```



La clase Statement contiene los métodos executeQuery y executeUpdate para realizar consultas y actualizaciones, respectivamente. Ambos métodos soportan consultas en SQL-92. Así por ejemplo, para obtener los nombres de los medicamentos que tenemos en la tabla medicamentos, de la base de datos farmacia.mdb que creamos anteriormente, tendríamos que emplear la sentencia:

```
ResultSet rs = stmt.executeQuery("SELECT nombre from medicamentos");
```

El método executeQuery devuelve un objeto ResultSet para poder recorrer el resultado de la consulta utilizando un cursor.

```
while (rs.next())  
    String usuario = rs.getString("nombre");
```

El método next se emplea para hacer avanzar el cursor. Para obtener una columna del registro utilizamos los métodos get. Hay un método get para cada tipo básico Java y para las cadenas.

Comentar que un método interesante del cursor es wasNull que nos informa si el último valor leído con un método get es nulo.

Respecto a las consultas de actualización, executeUpdate, retornan el número de registros insertados, registros actualizados o eliminados, dependiendo del tipo de consulta que se trate.

Aquí tienes el proyecto que realiza la consulta de todos los medicamentos de la tabla que los contiene, llamada medicamentos, en la base de datos farmacia.mdb.

[EjemploConexionAccess](#) (*link: [EjemploConexionAccess.java](#)*).

Recomendación

Puedes consultar todos los métodos que soporta ResultSet, además de más información, en la documentación de Oracle:

ResultSet. (*link: <https://docs.oracle.com/javase/8/docs/api/java/sql/ResultSet.html>*)

5.3.- Consultas preparadas.

Las consultas preparadas están representadas por la clase `PreparedStatement`.

Son **consultas precompiladas**, por lo que son más eficientes, y pueden tener parámetros.

Una consulta se instancia del modo que vemos con un ejemplo:

```
PreparedStatement pstmt = con.prepareStatement("SELECT * from medicamentos");
```



Para las consultas que se realizan muy a menudo es aconsejable usar este tipo de consultas, de modo que el rendimiento del sistema será mejor de esta manera.

Si hay que emplear parámetros en una consulta, se puede hacer usando el carácter '?'. Por ejemplo, para realizar una consulta de un medicamento que tenga un código determinado, haríamos la consulta siguiente:

```
PreparedStatement pstmt = con.prepareStatement("SELECT * from medicamentos WHERE codigo = ? ");
```

Establecemos los parámetros de una consulta utilizando métodos set que dependen del tipo SQL de la columna.

Así, le decimos que el primer parámetro, que es el único que tiene esta consulta, es "712786":

```
pstmt.setString(1, "712786");
```

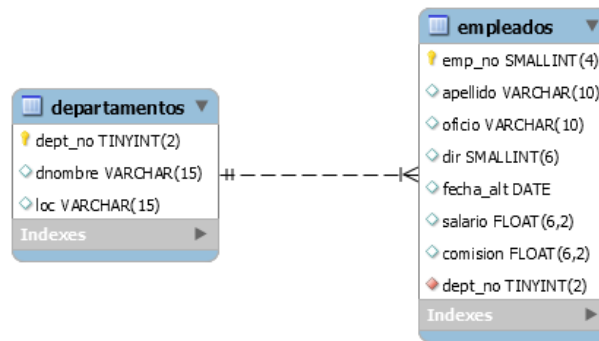
El primer argumento de este método es la posición del parámetro dentro de la consulta.

Finalmente, ejecutamos la consulta utilizando el método `executeQuery()` o `executeUpdate()`, ambos sin parámetros, dependiendo del tipo de consulta.

5.4.- Ejemplo MySQL

Para el siguiente ejemplo JAVA creamos desde MySQL una base de datos y un usuario con nombre EJEMPLO, la clave del usuario es la misma. A continuación creamos las siguientes tablas e insertamos datos en ellas.

Base de datos EJEMPLO ([link: ejemplo.sql](#))



El siguiente programa ilustra los pasos de funcionamiento de JDBC accediendo a la base de datos EJEMPLO:

[Main](#) ([link: Main.java](#))

Para poder probar el programa hemos de obtener el JAR que contiene el driver MySQL (en el ejemplo se ha utilizado mysql-connector-java-8.0.18-bin.jar) e incluirlo en el CLASSPATH o añadirlo a nuestro IDE, por ejemplo en Eclipse pulsamos en el proyecto con el botón derecho del ratón y seleccionamos Build Paths-> Add External Archives... para localizar el fichero JAR. Desde la URL <http://www.mysql.com/products/connector/> ([link: http://www.mysql.com/products/connector](#)) se puede descargar el conector.

En este ejemplo hemos visto cómo se realiza la conexión con una base de datos MySQL utilizando JDBC. Para conectarnos a través de JDBC a otras bases de datos relacionales (Oracle, Postgres) o embebidas (SQLite) podremos utilizar el mismo programa y solo cambiaremos la carga del driver (JAR adecuado) y la conexión a la base de datos.

6.- Ejecución de procedimientos almacenados en la base de datos.

Caso práctico



Ada está terminando de diseñar unos procedimientos almacenados para un proyecto que está realizando la empresa, para unos grandes almacenes. En esos grandes almacenes utilizan MySQL como base de datos, puesto que ese sistema gestor soporta la ejecución de dichos procedimientos. **Ada está pensando en pedir ayuda a alguien más de la empresa**, porque se está dando cuenta de que habrá que realizar bastante código para toda la funcionalidad que se necesita.

Un procedimiento almacenado es un procedimiento o subprograma que está almacenado en la base de datos.

Muchos sistemas gestores de bases de datos los soportan, por ejemplo: MySQL, Oracle, etc.

Además, estos procedimientos suelen ser de dos clases:

- ✓ **Procedimientos** almacenados.
- ✓ **Funciones**, las cuales devuelven un valor que se puede emplear en otras sentencias SQL.



Un procedimiento almacenado típico tiene:

- ✓ Un nombre.
- ✓ Una lista de parámetros.
- ✓ Unas sentencias SQL.

Veamos un ejemplo de sentencia para crear un procedimiento almacenado sencillo para MySQL, aunque sería similar en otros sistemas gestores:

```
CREATE PROCEDURE procedimiento1  
(IN par1 VARCHAR(13))  
BEGIN  
    DECLARE var1 VARCHAR(13);  
    IF par1 = '24' THEN  
        SET var1 = 'perro rabioso';  
    ELSE  
        SET var1 = 'gato persa';  
    END IF;  
    INSERT INTO animales (nombre) VALUES (var1);  
END
```

([link: AD03_CONT_R27_01_codPL.jpg.](#))

Como se ve en los comentarios, este procedimiento admite un parámetro, llamado par1. También se declara una variable a la que llamamos var1 y es de tipo carácter y longitud 13. Si el valor que le llega de parámetro es igual a 24, entonces se asigna a la variable var1, la cadena 'perro rabioso' y en caso contrario se le asignará la cadena: 'gato persa'. Finalmente, se inserta en la tabla "Animales" el valor que se asignó a la variable var1.

Debes conocer

En el capítulo 24 del manual de referencia de MySQL que puedes encontrar en el enlace siguiente, puedes familiarizarte con los comandos que puedes necesitar para realizar procedimientos almacenados y funciones:

Manual de referencia MySQL. ([link: https://dev.mysql.com/doc/refman/8.0/en/](https://dev.mysql.com/doc/refman/8.0/en/))

6.1.- Ejecutando procedimientos almacenados en MySQL.

A continuación, vamos a realizar un procedimiento almacenado en MySQL, que simplemente insertará datos en la tabla clientes. Desde el programa Java que realizamos, llamaremos para ejecutar a ese procedimiento almacenado. Por tanto, ¿cuál sería la secuencia que seguiríamos para realizar esto?



Si no tenemos creada la tabla de clientes, la creamos. Por simplicidad, en este ejemplo, trabajamos sobre una base de datos nueva en MySQL denominada: test. Una vez que tenemos el esquema creado. Para crear la tabla de clientes, el script correspondiente es:

[Crear tabla en MySQL. \(link: AD03_CONT_R28_CrearTabla.txt.\)](#) (1 KB)

- ✓ Creamos el procedimiento almacenado en la base de datos. Sería tan fácil como lo que ves en el siguiente enlace:

[DDL rutina insertar cliente. \(link: AD03_CONT_R28_01_RutinainsertarCliente.txt.\)](#) (1 KB)

- ✓ Crear la clase Java para desde aquí, llamar al procedimiento almacenado:

[Llamar al procedimiento almacenado. \(link: AD03_CONT_R28_02_inserConProcAlma.zip.\)](#) (1 KB)

Si hemos definido la tabla correctamente, con su clave primaria, y ejecutamos el programa, intentando insertar una fila igual que otra insertada, o sea, con la misma clave primaria, obtendremos un mensaje al capturar la excepción de este tipo:

```
SQL Exception:
com.mysql.jdbc.exceptions.jdbc4.MySQLIntegrityConstraintViolationException:
Duplicate entry '765' for key 'PRIMARY'
```

Caso práctico



Hay un aspecto sobre bases de datos, que **Ana** estudió en el ciclo formativo, y que no había tenido ocasión de ver en un caso real, y es el de las **transacciones en una base de datos**. Es un tema que le apasiona y le pide a María que le muestre alguna que haya realizado ella, para estudiarla a fondo y aprender con sus consejos.

Cuando tenemos una serie de consultas SQL que deben ejecutarse en conjunto, con el uso de transacciones podemos asegurarnos de que nunca nos quedaremos a medio camino de su ejecución.

Las transacciones tienen la característica de poder “deshacer” los cambios efectuados en las tablas, de una transacción dada, si no se han podido realizar todas las operaciones que forman parte de dicha transacción.

Por eso, las bases de datos que soportan transacciones son mucho más seguras y fáciles de recuperar si se produce algún fallo en el servidor que almacena la base de datos, ya que las consultas se ejecutan o no en su totalidad.



Al ejecutar una transacción, el motor de base de datos garantiza: **atomicidad, consistencia, aislamiento y durabilidad (ACID)** de la transacción (o conjunto de comandos) que se utilice.

El ejemplo típico que se pone para hacer más clara la necesidad de transacciones en algunos casos es el de una transacción bancaria. Por ejemplo, si una cantidad de dinero es transferida de la cuenta de Antonio a la cuenta de Pedro, se necesitarían dos consultas:

- ✓ En la cuenta de Antonio para quitar de su cuenta ese dinero:

```
UPDATE cuentas SET saldo = saldo - cantidad WHERE cliente = "Antonio";
```

- ✓ En la cuenta de Pedro para añadir ese dinero a su cuenta:

```
UPDATE cuentas SET saldo = saldo + cantidad WHERE cliente = "Pedro";
```

Pero, ¿qué ocurre si por algún imprevisto (un apagón de luz, etc.), el sistema “cae” después de que se ejecute la primera consulta, y antes de que se ejecute la segunda? Antonio tendrá una cantidad de dinero menos en su cuenta y creerá que ha realizado la transferencia. Pedro, sin embargo, creerá que todavía no le han realizado la transferencia.

Autoevaluación

Respecto a las transacciones, señala la respuesta correcta:

- ☐ (link:)
En caso de una transacción con tres operaciones, no se podrá deshacer en ningún caso.
- ☐ (link:)
Una transacción permite recuperar los datos si se produce un fallo, aportando por tanto más seguridad en la realización de operaciones en la base de datos.
- ☐ (link:)
Una transacción se hará sólo si las operaciones involucradas operan con dinero.
- ☐ (link:)
Ninguna es correcta.

7.1.- Commit y Rollback.

COMMIT



[\(link: AD03_CONT_R30_commit.JPG.\)](#)

ROLLBACK



Una transacción tiene dos finales posibles, **COMMIT o ROLLBACK**. Si se finaliza correctamente y sin problemas se hará con COMMIT, con lo que los cambios se realizan en la base de datos, y si por alguna razón hay un fallo, se deshacen los cambios efectuados hasta ese momento, con la ejecución de ROLLBACK.

Por defecto, al menos en MySQL, en una conexión trabajamos en modo autocommit con valor true. Eso significa que cada consulta es una transacción en la base de datos.

Por tanto, si queremos definir una transacción de varias operaciones, estableceremos el modo autocommit a false con el método setAutoCommit de la clase Connection.

En modo no autocommit las transacciones quedan definidas por las ejecuciones de los métodos commit y rollback. Una transacción abarca desde el último commit o rollback hasta el siguiente commit. Los métodos commit o rollback forman parte de la clase Connection.

En la siguiente porción de código de un procedimiento almacenado, puedes ver un ejemplo sencillo de cómo se puede utilizar commit y rollback: tras las operaciones se realiza el commit, y si ocurre una excepción, al capturarla realizaríamos el rollback.

```
BEGIN
...
SET AUTOCOMMIT OFF
update cuenta set saldo=saldo + 250 where dni="12345678-L";
update cuenta set saldo=saldo - 250 where dni="89009999-L";
COMMIT;
...

EXCEPTION
    WHEN OTHERS THEN
        ROLLBACK ;
END;
```

Es conveniente planificar bien la aplicación para minimizar el tiempo en el que se tengan transacciones abiertas ejecutándose, ya que consumen recursos y suponen bloqueos en la base de datos que puede parar otras transacciones. En muchos casos, un diseño cuidadoso puede evitar usos innecesarios que se salgan fuera del modo estándar AutoCommit.

Caso práctico



Cuando Ada se reunió con Juan y María, les hizo hincapié en que debían ser especialmente **rigurosos en la programación de los proyectos**, tratando los posibles errores de las aplicaciones que programaran, para evitar la finalización abrupta del programa, y también el cierre adecuado de las conexiones a bases de datos, para evitar perder la información. Así mismo, les comunicó que **transmitieran esto mismo a Ana y a Antonio**.

Debemos tener en cuenta siempre que las conexiones con una base de datos consumen muchos recursos en el sistema gestor, y por lo tanto en el sistema informático en general. Por ello, conviene cerrarlas con el método `close` siempre que vayan a dejar de ser utilizadas, en lugar de esperar a que el garbage collector de Java las elimine.

También conviene cerrar las consultas (`Statement` y `PreparedStatement`) y los resultados (`ResultSet`) para liberar los recursos.

Una excepción es una situación que no se puede resolver y que provoca la detención del programa de manera abrupta. Se produce por una condición de error en tiempo de ejecución.

En Java hay muchos tipos de excepciones, el paquete `java.lang.Exception` es el que contiene los tipos de excepciones.



8.1.- Excepciones.

Cuando se produce un **error durante la ejecución de un programa**, se genera un objeto asociado a esa excepción. Ese objeto es de la clase `Exception` o de alguna de sus subclases. Este objeto se pasa entonces al código que se ha definido para gestionar la excepción.

En una porción de programa donde se trabajara con ficheros, y con bases de datos podríamos tener esta estructura para capturar las posibles excepciones.

```
try {
    // Bloque de instrucciones del try
} catch (FileNotFoundException fde) {
    // Bloque para excepción por: Fichero no encontrado
} catch (IOException ioe) {
    // Bloque para excepción por: error de entrada salida
} catch (SQLException sqle) {
    // Bloque para excepción por: error con SQL
} catch (Exception e) {
} finally {
    // Instrucciones finales para, por ejemplo, limpiar
}
```

([link: AD03_CONT_R34_estrucexp.jpg.](#))



[Código con la estructura para capturar excepciones. \(link: AD03_CONT_R35_estrucexp.txt.\)](#) (1 KB)

El bloque de instrucciones del `try` es el que se ejecuta, y si en él ocurre un error que dispara una excepción, entonces se mira si es de tipo `FileNotFoundException`; si es así, se ejecutarían las instrucciones del bloque del `FileNotFoundException`. Si no era de ese tipo la excepción, se mira si es del siguiente tipo, o sea, de entrada salida, y así sucesivamente.

Las instrucciones que hay en el bloque del `finally`, se ejecutarán siempre, se haya producido una excepción o no, ahí suelen ponerse instrucciones de limpieza, de cierre de conexiones, etc.

Las acciones que se realizan sobre una base de datos pueden lanzar la excepción `SQLException`. Este tipo de excepción proporciona entre otra información:

- ✓ Una cadena de caracteres describiendo el error. Se obtiene con el método `getMessage`.
- ✓ Un código entero de error que especifica al fabricante de la base de datos.

Para saber más

Para saber más sobre excepciones, puedes consultar estos dos enlaces:

Excepciones en Java. ([link: http://elvex.ugr.es/decsai/java/pdf/B2-excepciones.pdf](http://elvex.ugr.es/decsai/java/pdf/B2-excepciones.pdf)) (88 KB)

8.2.- Cierre de conexiones.

Como ya hemos dicho, al trabajar con bases de datos, se consumen muchos recursos por parte del sistema gestor, así como del resto de la aplicación.

Por esta razón, resulta totalmente conveniente cerrarlas con el método close cuando ya no se utilizan.

Podríamos por tanto tener un ejemplo de cómo hacer esto:













```
Connection con = null ;
try {
    con = DriverManager.getConnection("jdbc:odbc:rdm01");
    System.out.println("Conexion establecida con éxito.");
    // Aquí hacemos lo que necesitamos
}
catch( SQLException e ) {
    // Aquí hacemos lo necesario para gestionar la excepción SQL
}
finally {
    // Se han cerrado la conexión
    if( con != null ) {
        try { con.close(); }
        catch( SQLException e ) {
            System.out.println(e.getMessage());
        }
    }
}
```

[*\(link: AD03_CONT_R37_cierre.jpg.\)*](#)



Anexo.- Licencias de recursos.

Licencias de recursos utilizados en la Unidad de Trabajo.

Recurso (1)	Datos del recurso (1)	Recurso (2)	
	<p>Autoría: krollian.</p> <p>Licencia: CC-by-nc.</p> <p>Procedencia: http://www.flickr.com/photos/krollian/3705826490/</p>		<p>Autoría: pcamb</p> <p>Licencia: CC-by.</p> <p>Procedencia: http://www.flickr.com/photos/pcamb/3705826490/</p>
	<p>Autoría: jimw.</p> <p>Licencia: CC-by.</p> <p>Procedencia: http://www.flickr.com/photos/jimwinstead/24124753/</p>		<p>Autoría: hongji</p> <p>Licencia: CC-by.</p> <p>Procedencia: http://www.flickr.com/photos/hongji/24124753/</p>
	<p>Autoría: 姒儿喵喵.</p> <p>Licencia: CC-by-nc.</p> <p>Procedencia: http://www.flickr.com/photos/crystaljingsr/3914729343/</p>		<p>Autoría: Amorti</p> <p>Licencia: CC-by.</p> <p>Procedencia: http://www.flickr.com/photos/Amorti/3914729343/</p>
	<p>Autoría: Chavezonico.</p> <p>Licencia: CC-by-nc-sa.</p> <p>Procedencia: http://www.flickr.com/photos/chavezonico/3806410897/</p>		<p>Autoría: MrDoS</p> <p>Licencia: CC-by.</p> <p>Procedencia: http://www.flickr.com/photos/MrDoS/3806410897/</p>
	<p>Autoría: Andrew Murdoch</p> <p>Licencia: CC-by-nc-sa.</p> <p>Procedencia: http://www.flickr.com/photos/andrewmurdoch/3233287999/sizes/s/in/photostream/</p>		<p>Autoría: Andrés</p> <p>Licencia: CC-by.</p> <p>Procedencia: http://www.flickr.com/photos/Andrés/3233287999/sizes/s/in/photostream/</p>
	<p>Autoría: Monroe's Dragonfly</p> <p>Licencia: CC-by.</p> <p>Procedencia: http://www.flickr.com/photos/monroesdragonfly/2739734655/</p>		<p>Autoría: Ivan W</p> <p>Licencia: CC-by.</p> <p>Procedencia: http://www.flickr.com/photos/IvanW/2739734655/</p>

