TEMA 13: BASES DE DATOS RELACIONALES

API JDBC como interfaz de acceso a bases de datos SQL

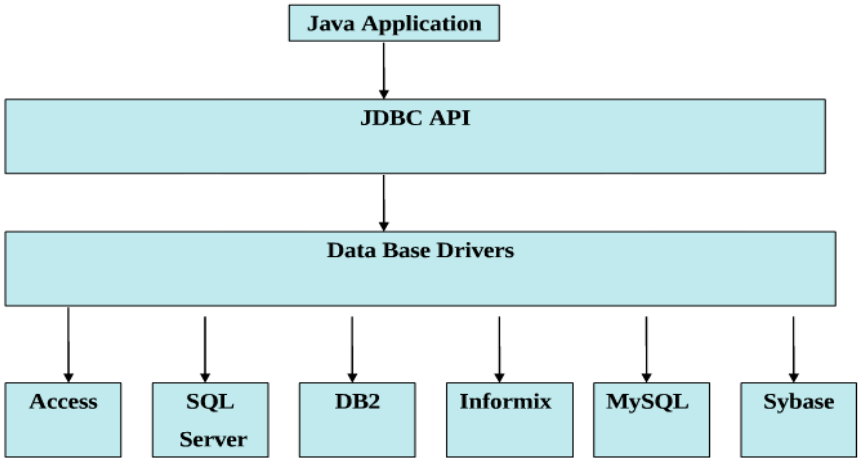
1.- INTRODUCCIÓN

En el mundo de la informática hay numerosos estándares y lenguajes, la mayoría de los cuales son incapaces de comunicarse entre sí. Afortunadamente, algunos de ellos son verdaderos referentes cuyo conocimiento es vital para los programadores. El ***Structured Query Language*** o SQL, se ha convertido en el método estándar de acceso a bases de datos.

Tener conocimientos de SQL es una necesidad para cualquier profesional de las tecnologías de la información (TI). A medida que el desarrollo de sitios web se hace más común entre personas sin conocimientos de programación, el tener una cierta noción de SQL se convierte en requisito indispensable para integrar datos en páginas HTML. No obstante, éste no es el único ámbito de SQL, puesto que la mayor parte de las aplicaciones de gestión que se desarrollan hoy en día acceden, en algún momento, a alguna base de datos utilizando sentencias SQL. La mayoría de los fabricantes ofrecen APIs (*Applications Programming Interface*) nativas para acceder a sus bases de datos, sin embargo, el hecho de utilizar estas APIs, obliga a los programadores a adaptar cada programa a las características de un servidor determinado.

Para solventar este problema, Java introduce la **API JDBC**, compuesta por un conjunto de clases que unifican el acceso a las bases de datos. Gracias a la utilización de JDBC, un programa Java puede acceder a cualquier base de datos sin necesidad de modificar la aplicación. Sin embargo, para que esto sea posible es necesario que el fabricante ofrezca un driver que cumpla la especificación JDBC.

Un **driver JDBC** es una capa de software intermedia que traduce las llamadas JDBC a los APIs específicos del vendedor. Así, por ejemplo, el desarrollador de la base de datos **MySQL** ofrece el driver **Connector/J** o en el caso de **Firebird** disponemos de **JayBird**.



El producto **JDBC**, según Oracle, tiene cuatro componentes:

- **El API JDBC**. Con ella se pueden realizar consultas SQL, recuperar datos y realizar cambios a la base de datos. Está dividida en dos paquetes, **java.sql** y **javax.sql**.
- **El administrador de controladores *(drivers)* JDBC**. La clase **DriverManager** define los objetos desde lo que se pueden conectar las aplicaciones java a un controlador JDBC.
- **La suite de test JDBC**. Utilizado para testear las aplicaciones Java que utilizan JDBC.
- **El puente JDBC-ODBC**. Proporciona acceso a gestores de bases de datos a través de ODBC.

Por último, y al margen de lo expuesto anteriormente, hay que mencionar que Java incluye **Java DB** que es una pequeña base de datos relacional que es parte del JDK (*Java Development Kit*), lo que facilita la incorporación de bases de datos en nuestras aplicaciones. Java DB está construida desde la base de datos de código abierto **Apache Derby**, y viene con su propio driver.

2.- PROGRAMACIÓN DE BASES DE DATOS CON JDBC

La información contenida en un servidor de bases de datos es normalmente el bien más preciado dentro de una empresa. La API JDBC ofrece a los desarrolladores Java un modo de conectar con dichas bases de datos. Utilizando la API JDBC, los desarrolladores pueden crear un cliente que pueda conectar con una base de datos, ejecutar instrucciones SQL y procesar el resultado de esas instrucciones.

La API proporciona conectividad y acceso a datos en toda la extensión de las bases de datos relacionales. JDBC generaliza las funciones de acceso a bases de datos más comunes abstrayendo los detalles específicos de una determinada base de datos. El resultado es un conjunto de clases e interfaces, localizadas en los paquetes **java.sql** y **javax.sql**, que pueden ser utilizadas con cualquier base de datos que disponga del driver JDBC apropiado. La utilización de este driver significa que, siempre y cuando una aplicación utilice las características más comunes de acceso a bases de datos, dicha aplicación podrá utilizarse con una base de datos diferente cambiando simplemente a un driver JDBC diferente.

Los vendedores de bases de datos más populares como Oracle, Sybase e Informix ofrecen APIs de su propiedad para el acceso del cliente. Las aplicaciones cliente escritas en lenguajes nativos pueden utilizar estos APIs para obtener acceso directo a los datos, pero no ofrecen una interfaz común de acceso a diferentes bases de datos. La API JDBC ofrece una alternativa al uso de estas APIs, permitiendo acceder a diferentes servidores de bases de datos, únicamente cambiando el driver JDBC por el que ofrezca el fabricante del servidor al que se desea acceder.

3.- TIPOS DE DRIVER JDBC

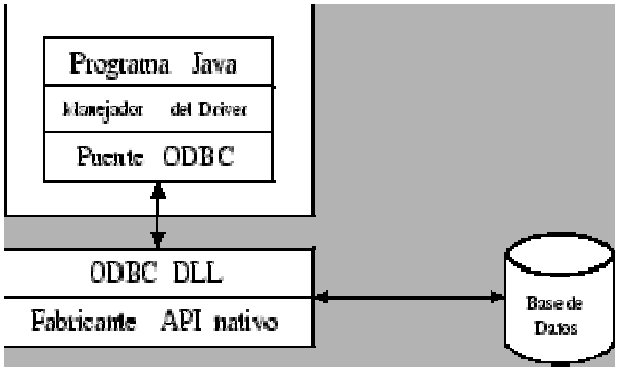
Una de las decisiones importantes en el diseño, cuando estamos proyectando una aplicación de bases de datos Java, es decidir el driver JDBC que permitirá que las clases JDBC se comuniquen con la base de datos. Los driver JDBC se clasifican en cuatro tipos o niveles:

- **Tipo 1:** Puente JDBC-ODBC
- **Tipo 2:** Driver API nativo/parte Java
- **Tipo 3:** Driver protocolo de red/todo Java
- **Tipo 4:** Driver protocolo nativo/todo Java

Entender cómo se construyen los drivers y cuales son sus limitaciones, nos ayudará a decidir qué driver es el más apropiado para cada aplicación.

3.1 Tipo 1: Driver puente JDBC-ODBC

El puente JDBC-ODBC es un driver JDBC del tipo 1 que traduce operaciones JDBC en llamadas a la API ODBC. Estas llamadas son entonces cursadas a la base de datos mediante el driver ODBC apropiado. Esta arquitectura se muestra en la siguiente figura:



El puente se implementa como el paquete `oracle.jdbc.odbc` y contiene una biblioteca nativa utilizada para acceder a ODBC.

Ventajas

A menudo, el primer contacto con un driver JDBC es un puente JDBC-ODBC, simplemente porque es el driver que se distribuye como parte de Java, como el paquete `oracle.jdbc.odbc.JdbcOdbcDriver`.

Además tiene la ventaja de poder trabajar con una gran cantidad de drivers ODBC. Los desarrolladores suelen utilizar ODBC para conectar con bases de datos en un entorno distinto de Java. Por tanto, los drivers de tipo 1 pueden ser útiles para aquellas compañías que ya tienen un driver ODBC instalado en las máquinas clientes. Se utilizará normalmente en máquinas basadas en Windows que ejecutan aplicaciones de gestión. Por supuesto, puede ser el único modo de acceder a algunas bases de datos de escritorio, como MS Access, dBase y Paradox.

En este sentido, la ausencia de complejidad en la instalación y el hecho de que nos permita acceder virtualmente a cualquier base de datos, le convierte en una buena elección. Sin embargo, hay muchas razones por las que se desecha su utilización.

Desventajas

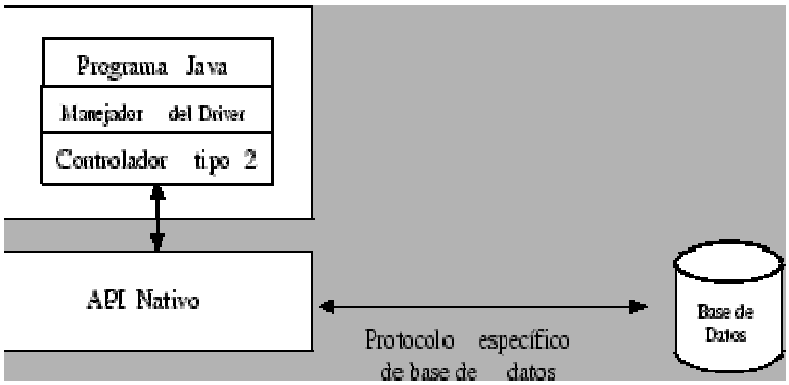
Básicamente sólo se recomienda su uso cuando se están realizando esfuerzos dirigidos a prototipos y en casos en los que no exista otro driver basado en JDBC para esa tecnología. Si es posible, debemos utilizar un driver JDBC en vez de un puente y un driver ODBC. Esto elimina totalmente la configuración cliente necesaria en ODBC.

Los siguientes puntos resumen algunos de los inconvenientes de utilizar el driver puente:

- Rendimiento: Como puede imaginar por el número de capas y traducciones que tienen lugar, utilizar el puente está lejos de ser la opción más eficaz en términos de rendimiento.
- Utilizando el puente JDBC-ODBC, el usuario está limitado por la funcionalidad del driver elegido. Es más, dicha funcionalidad se limita a proporcionar acceso a características comunes a todas las bases de datos. No pudiendo hacer uso de las mejoras que cada fabricante introduce en sus productos, especialmente en lo que afecta a rendimiento y escalabilidad.
- El driver puente no funciona adecuadamente con applets. El driver ODBC y la interfaz de conexión nativa deben estar ya instalados en la máquina cliente. Por eso, cualquier ventaja de la utilización de applets en un entorno de Intranet se pierde, debido a los problemas de despliegue que conllevan las aplicaciones tradicionales.
- La mayoría de los browser no tienen soporte nativo del puente. Como el puente es un componente opcional del Java 2 SDK Standard Edition, no se ofrece con el navegador. Incluso si fuese ofrecido, sólo los applets de confianza (aquellos que permiten escribir en archivos) serán capaces de utilizar el puente. Esto es necesario para preservar la seguridad de los applet. Para terminar, incluso si el applet es de confianza, ODBC debe ser configurado en cada máquina cliente.

3.2. Tipo 2: Driver API Nativo / parte Java

Los drivers de tipo 2, del que es un ejemplo el driver JDBC/OCI de Oracle, utilizan la interfaz de métodos nativos de Java para convertir las solicitudes de API JDBC en llamadas específicas a bases de datos para RDBMS como SQL Server, Informix, Oracle o Sybase, como se puede ver en la siguiente figura:



Aunque los drivers de tipo 2 habitualmente ofrecen mejor rendimiento que el puente JDBC-ODBC, siguen teniendo los mismos problemas de despliegue en los que la interfaz de conectividad nativa debe estar ya instalada en la máquina cliente. El driver JDBC necesita una biblioteca suministrada por el fabricante para traducir las funciones JDBC en lenguaje de consulta específico para ese servidor. Estos drivers están normalmente escritos en alguna combinación de Java y C/C++, ya que el driver debe utilizar una capa de C para realizar llamadas a la biblioteca que está escrita en C.

Ventajas

El driver de tipo 2 ofrece un rendimiento significativamente mayor que el puente JDBC-ODBC, ya que las llamadas JDBC no se convierten en llamadas ODBC, sino que son directamente nativas.

Desventajas

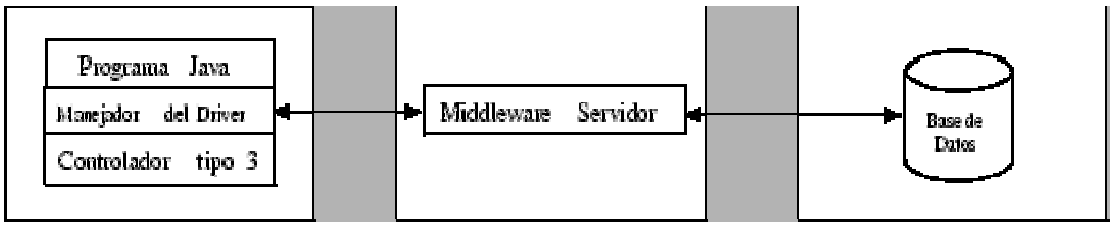
La biblioteca de la base de datos del fabricante necesita iniciarse en cada máquina cliente. En consecuencia, los drivers de tipo 2 no se pueden utilizar en Internet. Los drivers de tipo 2 muestran menor rendimiento que los de tipo 3 y 4.

Un driver de tipo 2 también utiliza la interfaz nativa de Java, que no está implementada de forma consistente entre los distintos fabricantes de JVM por lo que habitualmente no es muy portable entre plataformas.

3.3. Tipo 3: Driver protocolo de red / todo Java

Los drivers JDBC de tipo 3 están implementados en una aproximación de tres capas por lo que las solicitudes de la base de datos JDBC están traducidas en un protocolo de red independiente de la base de datos y dirigidas al servidor de capa intermedia. El servidor de la capa intermedia recibe las solicitudes y las envía a la base de datos utilizando para ello un driver JDBC del tipo 1 o del tipo 2 (lo que significa que se trata de una arquitectura muy flexible).

La arquitectura en conjunto consiste en tres capas: la capa cliente JDBC y driver, la capa intermedia y la base o las bases de datos a las que se accede.



El driver JDBC se ejecuta en el cliente e implementa la lógica necesaria para enviar a través de la red comandos SQL al servidor JDBC, recibir las respuestas y manejar la conexión.

El componente servidor intermedio puede implementarse como un componente nativo, o alternativamente escrito en Java. Las implementaciones nativas conectan con la base de datos utilizando bien una biblioteca cliente del fabricante o bien ODBC. El servidor tiene que configurarse para la base o bases de datos a las que se va a acceder. Esto puede implicar asignación de números de puerto, configuración de variables de entorno, o de cualquier otro parámetro que pueda necesitar el servidor. Si el servidor intermedio está escrito en Java, puede utilizar cualquier driver en conformidad con JDBC para comunicarse con el servidor de bases de datos mediante el protocolo propietario del fabricante. El servidor JDBC maneja varias conexiones con la base de datos, así como excepciones y eventos de estado que resultan de la ejecución de SQL. Además, organiza los datos para su transmisión por la red a los clientes JDBC.

Ventajas

El driver protocolo de red/todo Java tiene un componente en el servidor intermedio, por lo que no necesita ninguna biblioteca cliente del fabricante para presentarse en las máquinas clientes.

Los drivers de tipo 3 son los que mejor funcionan en redes basadas en Internet o Intranet, aplicaciones intensivas de datos, en las que un gran número de operaciones concurrentes como consultas, búsquedas, etc., son previsibles y escalables y su rendimiento es su principal factor. Hay muchas oportunidades de optimizar la portabilidad, el rendimiento y la escalabilidad.

El protocolo de red puede estar diseñado para hacer el driver JDBC cliente muy pequeño y rápido de iniciar, lo que es perfecto para el despliegue de aplicaciones de Internet.

Además, un driver tipo 3 normalmente ofrece soporte para características como almacenamiento en memoria caché (conexiones, resultados de consultas, etc.), equilibrio de carga, y administración avanzada de sistemas como el registro.

La mayor parte de aplicaciones web de bases de datos basadas en 3 capas implican seguridad, firewalls y proxis y los drivers del tipo 3 ofrecen normalmente estas características.

Inconvenientes

Los drivers de tipo 3 requieren código específico de bases de datos para realizarse en la capa intermedia. Además, atravesar el conjunto de registros puede llevar mucho tiempo, ya que los datos vienen a través del servidor de datos.

3.4. Tipo 4: Driver protocolo nativo / todo Java

Este tipo de driver comunica directamente con el servidor de bases de datos utilizando el protocolo nativo del servidor. Estos drivers pueden escribirse totalmente en Java, son independientes de la plataforma y eliminan todo los aspectos relacionados con la configuración en el cliente. Sin embargo, este driver es específico de un fabricante determinado de base de datos. Cuando la base de datos necesita ser cambiada a un producto de otro fabricante, no se puede utilizar el mismo driver. Por el contrario, hay que reemplazarlo y también el programa cliente, o su asignación, para ser capaces de utilizar una cadena de conexión distinta para iniciar el driver.

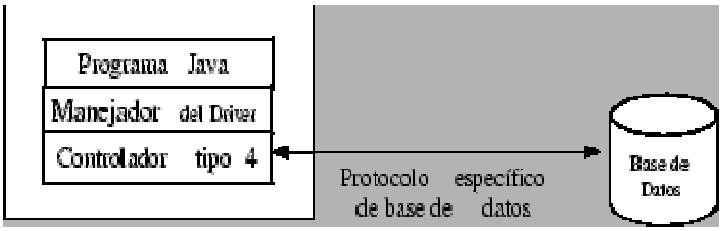
Estos drivers traducen JDBC directamente a protocolo nativo sin utilizar ODBC o la API nativa, por lo que pueden proporcionar un alto rendimiento de acceso a bases de datos.

Ventajas

- Como los drivers JDBC de tipo 4 no tienen que traducir las solicitudes de ODBC o de una interfaz de conectividad nativa, o pasar la solicitud a otro servidor, el rendimiento es bastante bueno.
- El driver protocolo nativo/todo Java da lugar a un mejor rendimiento que los de tipo 1 y 2.
- No hay necesidad de instalar ningún software especial en el cliente o en el servidor.
- Estos drivers pueden bajarse de la forma habitual.

Desventajas

Con los drivers de tipo 4, el usuario necesita un driver distinto para cada base de datos.



4.- ARQUITECTURAS PARA APLICACIONES CON BASES DE DATOS

Hay una gran variedad de arquitecturas posibles para las aplicaciones de bases de datos (dependiendo de los requisitos de la aplicación). Elegir el driver JDBC correcto es importante porque tiene un impacto directo en el rendimiento de la aplicación.

El puente JDBC-ODBC se podría considerar únicamente como una solución transitoria ya que no soporta todas las características de Java, y el usuario está limitado por la funcionalidad del driver ODBC elegido. Las aplicaciones a gran escala utilizarán drivers de los tipos 2, 3 o 4.

En las aplicaciones de Intranet es útil considerar los driver de tipo 2, pero estos drivers, como el puente ODBC, necesitan que ese código se instale en cada cliente. Por lo tanto, tienen los mismos problemas de mantenimiento que el driver puente JDBC-ODBC. Sin embargo, los drivers de tipo 2 son más rápidos que los de tipo 1 porque se elimina el nivel extra de traducción. Como los drivers de tipo 3 y de tipo 4 muestran mejor rendimiento que los drivers de tipo 2, la tendencia se dirige hacia un driver Java puro más robusto.

Para las aplicaciones relacionadas con Internet, no hay otra opción que utilizar drivers del tipo 3 o del tipo 4. Los drivers de tipo 3 son los que mejor funcionan con los entornos que necesitan proporcionar conexión a gran cantidad de servidores de bases de datos y a bases de datos heterogéneas. Los drivers de tipo 3 funcionan con aplicaciones intensivas de datos multiusuario, en las que se espera un alto número de operaciones concurrentes de datos, siendo el rendimiento y la escalabilidad el principal factor. El servidor puede proporcionar facilidades de registro y de administración, características del equilibrio de carga y puede soportar cachés de catálogo y de consultas.

Los drivers de tipo 4 están generalmente recomendados para aplicaciones que requieren un acceso rápido y eficiente a bases de datos. Como estos drivers traducen llamadas JDBC directamente a protocolo nativo sin utilizar ODBC o el API nativos, pueden aportar acceso a bases de datos de alto rendimiento.

4.1. Herramientas necesarias para trabajar con bases de datos y JDBC

Para trabajar con JDBC se necesitará crear un entorno mínimo que permita compilar y ejecutar los programas Java. La manera de desarrollar más cómoda y versátil es tener en la máquina de desarrollo la base de datos y demás software. De esta manera es fácil administrar la base de datos y podremos evitar todos los problemas que podrían surgir al tener el cliente y la base de datos en máquinas distintas.

Para crear el entorno JDBC se deberá tener lo siguiente:

- Una versión de **Java** (preferiblemente la última versión del Java SE SDK).
- Una **base de datos**. En los ejemplos que se muestran a lo largo del tema se va a emplear MySQL, pero las diferencias serían mínimas si queremos utilizar cualquier otro SGBD. MySQL es una base de datos relacional, multihilo y multiusuario.

  Para nuestras prácticas no instalaremos MySQL sólo, sino que instalaremos XAMPP, que es un servidor independiente de plataforma, software libre, que consiste principalmente en la base de datos MySQL, el servidor web Apache y los intérpretes para lenguajes de script: PHP y Perl. El nombre proviene del acrónimo de **X** (para cualquiera de los diferentes sistemas operativos), **A**pache, **M**ySQL, **P**HP, **P**erl. XAMPP me permite administrar y manejar la base de datos desde un navegador web.
- Los **drivers** necesarios para conectarse con la base de datos utilizada. En el caso de que se utilice MySQL como base de datos, habrá que instalar **Connector/J**. Si fuésemos a utilizar Firebird, habría que instalar **JayBird.**

5.- MANEJANDO SQLEXCEPTIONS

Cuando JDBC encuentra un error al trabajar con una base de datos en vez de una **Exception**, lanza una **SQLException**. A la hora de programar, el objeto SQLException contiene mucha información que puede servirnos de ayuda para determinar el origen del error:

- Descripción del error. Se puede recupera esta descripción utilizando el método `getMessage()` el cual devuelve un datos de tipo String.
- Código SQLState. Estos códigos y su significado están estandarizados por la ISO/ANSI y el *Open Group*. Este código es un objeto String y se puede recuperar mediante el método `getSQLState()`.
- Código de error. Código numérico que identifica el error producido. Este código se puede obtener llamando al método `getErrorCode()`.
- Causa del error. Una SQLException puede haber sido lanzada debido a una o varias causas. Para recuperar todas estas causas basta con llamar de manera recursiva al método `getCause()` hasta que se recupere el valor *null*.
- Si en vez de una sola excepción se han producido varias se pueden recuperar llamando al método `getNextException()` en la excepción lanzada.

A continuación se muestra cómo podríamos tratar una SQLException de forma detallada:

```
try {
        ...
}

catch (SQLException e) {
        printSQLException(e);  // Método nuestro que maneja la excepción
}
```

```
public static void printSQLException(SQLException ex)
{
	ex.printStackTrace(System.err);
	System.err.println("SQLState: "+ex.getSQLState());
	System.err.println("Error code: "+ex.getErrorCode());
	System.err.println("Message: "+ex.getMessage());
	Throwable t = ex.getCause();
	while (t!=null) {
		System.out.println("Cause: "+t);
		t = t.getCause();
	}
}
```

6.- PROFUNDIZANDO EN LA API JDBC

Hasta el momento se han presentado, por un lado, las características generales de la API JDBC y por otro, los distintos tipos de driver que se pueden utilizar. Sin embargo, la arquitectura JDBC está basada en un conjunto de clases Java que permiten conectar con bases de datos, crear y ejecutar sentencias SQL o recuperar y modificar la información almacenada en una base de datos. En las siguientes secciones se describirán cada una de estas operaciones:

- Cargar un driver de base de datos
- Establecer conexiones con bases de datos
- Crear y ejecutar instrucciones SQL
- Consultar bases de datos
- Realizar transacciones

6.1. Cargar un driver de base de datos y abrir conexiones

La interfaz `java.sql.Connection` representa una conexión con una base de datos. Es una interfaz porque la implementación de una conexión depende de la red, del protocolo y del vendedor. El API JDBC ofrece dos vías diferentes para obtener conexiones. La primera utiliza la clase `java.sql.DriverManager` y es adecuada para acceder a bases de datos desde programas cliente escritos en Java. El segundo enfoque se basa en el acceso a bases de datos desde aplicaciones J2EE (Java 2 Enterprise Edition).

Consideremos cómo se obtienen las conexiones utilizando la clase `java.sql.DriverManager`. En una aplicación, podemos obtener una o más conexiones para una o más bases de datos utilizando drivers JDBC. Cada driver implementa la interfaz `java.sql.Driver`. Uno de los métodos que define esta interfaz es el método `connect`(), que permite establecer una conexión con la base de datos y obtener un objeto `Connection`.

En lugar de acceder directamente a clases que implementan la interfaz `java.sql.Driver`, el enfoque estándar para obtener conexiones es registrar cada driver con `java.sql.DriverManager` y utilizar los métodos proporcionados en esta clase para obtener conexiones. `java.sql.DriverManager` puede gestionar múltiples drivers. Antes de entrar en los detalles de este enfoque, es preciso entender cómo JDBC representa la URL de una base de datos.

Los URL de JDBC

La noción de un URL en JDBC es muy similar al modo típico de utilizar los URL. Los URL de JDBC proporcionan un modo de identificar un driver de base de datos. Un URL de JDBC representa un driver y la información adicional necesaria para localizar una base de datos y conectar a ella. Su sintaxis es la siguiente:

```
jdbc:<subprotocol>:<subname>
```

Existen tres partes separadas por dos puntos:

- Protocolo: En la sintaxis anterior, `jdbc` es el protocolo. Éste es el único protocolo permitido en JDBC.
- Subprotocolo: Utilizado para identificar el driver que utiliza la API JDBC para acceder al servidor de bases de datos. Este nombre depende de cada fabricante.
- Subnombre: La sintaxis del subnombre es específica del driver.

Por ejemplo, para una base de datos MySQL llamada ``Banco", el URL al que debe conectar es: `jdbc:mysql:Banco`

Alternativamente, si estuviéramos utilizando Oracle mediante el puente JDBC-ODBC, nuestro URL sería: `jdbc:odbc:Banco`

Los URL de JDBC son lo suficientemente flexibles como para especificar información específica del driver en el subnombre. Veamos para terminar este apartado un ejemplo de URL, más completo usando MySQL:

```
jdbc:mysql://localhost:3306/Banco
```

donde,

`localhost`. Es la dirección de la máquina donde reside la base de datos.
`3306`. Es el puerto donde escucha la base de datos.
`Banco`. Es la base de datos a la que se conectará el programa.

Clase `DriverManager`

El propósito de la clase `java.sql.DriverManager` (gestor de drivers) es proporcionar una capa de acceso común encima de diferentes drivers de base de datos utilizados en una aplicación. En este enfoque, en lugar de utilizar clases de implementación Driver directamente, las aplicaciones utilizan la clase `DriverManager` para obtener conexiones. Esta clase ofrece tres métodos estáticos para obtener conexiones. Sin embargo, `DriverManager` requiere que cada driver que necesite la aplicación sea registrado antes de su uso, de modo que el `DriverManager` sepa que está ahí.

El enfoque JDBC para el registro de un driver de base de datos puede parecer oscuro al principio. Fíjese en el siguiente fragmento de código que carga el driver de base de datos de MySQL:

```
try {
    Class.forName("com.mysql.jdbc.Driver");
} catch (ClassNotFoundException e) {
    // Driver no encontrado
}
```

En tiempo de ejecución, el `ClassLoader` localiza y carga la clase `com.mysql.jdbc.Driver` desde la ruta de clases utilizando el cargador de clase de autoarranque. Mientras carga una clase, el cargador de clase ejecuta cualquier código estático de inicialización para la clase. En JDBC, se requiere que cada proveedor de driver registre una instancia del driver con la clase `java.sql.DriverManager` durante esta inicialización estática. Este registro tiene lugar automáticamente cuando el usuario carga la clase del driver (utilizando la llamada `Class.forName()`).

Una vez que el driver ha sido registrado con el `java.sql.DriverManager`, podemos utilizar sus métodos estáticos para obtener conexiones. El gestor de drivers tiene tres variantes del método estático `getConnection()` utilizado para establecer conexiones. El gestor de drivers delega estas llamadas en el método `connect()` de la interfaz `java.sql.Driver`.

Dependiendo del tipo de driver y del servidor de base de datos, una conexión puede conllevar una conexión de red física al servidor de base de datos o a un proxy de conexión. Las bases de datos integradas no requieren conexión física. Exista o no una conexión física, el objeto de conexión es el único objeto que utiliza una conexión para comunicar con la base de datos. Toda comunicación debe tener lugar dentro del contexto de una o más conexiones.

Consideremos ahora los diferentes métodos para obtener una conexión:

- **`public static Connection getConnection(String url) throws SQLException`**

  `java.sql.DriverManager` recupera el driver apropiado del conjunto de drivers registrados. El URL de la base de datos está especificado en la forma de `jdbc:subprotocol:subname`. Para poder obtener una conexión a la base de datos es necesario que se introduzcan correctamente los parámetros de autentificación requeridos por el servidor de bases de datos.

- **`public static Connection getConnection(String url, Properties info) throws SQLException`**

  Este método requiere un URL y un objeto `java.util.Properties`. El objeto Properties contiene cada parámetro requerido para la base de datos especificada. La lista de propiedades difiere entre bases de datos. Dos propiedades comúnmente utilizadas para una base de datos son `autocommit=true` y `create=false`. Podemos especificar estas propiedades junto con el URL como `jdbc:subprotocol:subname;`

  `autocommit=true;create=true` o podemos establecer estas propiedades utilizando el objeto `Properties` y pasar dicho objeto como parámetro en el anterior método `getConnection()`.

```
String url = "jdbc:mysql:Banco";
Properties p = new Properties();
p.put("autocommit", "true");
p.put("create", "true");
Connection connection = DriverManager.getConnection(url, p);
```

  En caso de que no se adjunten todas las propiedades requeridas para el acceso, se generará una excepción en tiempo de ejecución.

- **`public static Connection getConnection(String url, String user, String password) throws SQLException`**

  La tercera variante toma como argumentos además del URL, el nombre del usuario y la contraseña.

```
String url      = "jdbc:mysql:Banco";
String user     = "root";
String password = "nadiuska";
Connection connection = DriverManager.getConnection(url, user, password);
```

Observe que todos estos métodos están sincronizados, lo que supone que sólo puede haber un hilo accediendo a los mismos en cada momento. Estos métodos lanzan una excepción `SQLException` si el driver no consigue obtener una conexión.

Interfaz `Driver`

Cada driver debe implementar la interfaz `java.sql.Driver`. En MySQL, la clase `com.mysql.jdbc.Driver` implementa la interfaz `java.sql.Driver`.

La clase `DriverManager` utiliza los métodos definidos en esta interfaz. En general, las aplicaciones cliente no necesitan acceder directamente a la clase `Driver` puesto que se accederá a la misma a través de la API JDBC. Esta API enviará las peticiones al Driver, que será, quién en último término, acceda a la base de datos.

6.2. Ejemplo de establecimiento de una conexión

Como hemos visto para comunicar con una base de datos utilizando JDBC, debemos en primer lugar establecer una conexión con la base de datos a través del driver JDBC apropiado. El API JDBC especifica la conexión en la interfaz `java.sql.Connection`.

El siguiente código muestra un ejemplo de conexión JDBC a una base de datos MySQL:

```
Connection connection=null;
String url      = "jdbc:mysql://localhost:3306/Banco ";
String login    = "root";
String password = "nadiuska";

try {
     connection = DriverManager.getConnection(url, login, password);

     System.out.println("Conexión establecida");
     // Acceso a datos utilizando el objeto de conexión
     . . .
} catch (SQLException sqle) {
        e.printStackTrace();
} finally {
     try {
          connection.close();
          System.out.println("Conexión cerrada");
     } catch (SQLException e) {
          e.printStackTrace();
     }
}
```

En este ejemplo, la clase DriverManager intenta establecer una conexión con la base de datos `Banco` utilizando el driver JDBC que proporciona MySQL. Para poder acceder al RDBMS MySQL es necesario introducir un *login* y un *password* válidos.

En el API JDBC, hay varios métodos que pueden lanzar la excepción `SQLException`. En este ejemplo, la conexión se cierra al final del bloque `finally`, de modo que los recursos del sistema puedan ser liberados independientemente del éxito o fracaso de cualquier operación de base de datos.

6.3. Crear y ejecutar instrucciones SQL

Antes de poder ejecutar una sentencia SQL, es necesario obtener un objeto de tipo **`Statement`**. Una vez creado dicho objeto, podrá ser utilizado para ejecutar cualquier operación contra la base de datos.

Existen tres tipos de objetos `Statement`:

- **Statement**. Sirve para enviar ordenes SQL a la base de datos sin parámetros.
- **PreparedStatement**. Hereda de *Statement*. Se utiliza para ejecutar comandos SQL con sin parámetros de entrada ya precompilados.

- **CallableStatement**. Hereda de *PreparedStatement*. Se utiliza para llamar a procedimientos almacenados en la base de datos. Permite trabajar con parámetros de entrada y de salida.

Un objeto de la clase `Statement`, se crea mediante el método `createStatement` de la interfaz `Connection`. Con este objeto, podemos enviar instrucciones SQL a la base de datos y recibir los resultados. Para crear el objeto de manera exitosa primero hay que conectarse a la base de datos.

```
Statement createStatement() throws SQLException
```

En el siguiente ejemplo se muestra como se realizaría:

```
Connection con =
    DriverManager.getConnection("jdbc:mysql://localhost:3306/mibase","rosi","123");

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

Como ya se ha indicado, la finalidad de un objeto `Statement` es ejecutar una instrucción SQL que puede o no devolver resultados. Para ello, la interfaz `Statement` dispone de los siguientes métodos:

- **`executeQuery()`**. Se utiliza para ejecutar sentencias SELECT y la llamada a este método devuelve un objeto `ResultSet`, el cual permite tratar los datos devueltos por la base de datos.
- **`executeUpdate()`**. Se utiliza para realizar actualizaciones que no devuelvan un `resultSet`. Por ejemplo, sentencias DML SQL (*Data Manipulation Language*) como `INSERT`, `UPDATE` y `DELETE`, o sentencias DDL SQL (*Data Definition Language*) como `CREATE TABLE`, `DROP TABLE` y `ALTER TABLE`. El valor que devuelve este método es un entero (conocido como la cantidad de actualizaciones) que indica el número de filas que se vieron afectadas. Las sentencias que no operan en filas, como `CREATE TABLE` o `DROP TABLE`, devuelven el valor cero.
- **`execute()`**. Utilizado en sentencias que devuelven más de un `resultSet`. Se utiliza solamente en programación avanzada.

Como buena práctica, se recomienda cerrar los objetos `Statement` mediante este comando:

```
smtp.close()
```

Ejemplo: Creación de tablas

Al ejecutar este ejemplo, se crearán las tablas EQUIPO y JUGADORES en la base de datos de nombre “mibase”.

```
import java.sql.*;

public class CreacionTablas {

    public static void main(String[] args) {
           try {
                  Class.forName("com.mysql.jdbc.Driver");
                  Connection conexion = DriverManager.getConnection
                         ("jdbc:mysql://localhost:3306/mibase","rosi","123");
                  System.out.println("Conexión establecida");
                  creaTablaEquipo(conexion, "mibase");
                  creaTablaJugador(conexion, "mibase");
           }

           catch (SQLException e2) {
                  printSQLException(e2);
           }
           catch (Exception e1) {
                  e1.printStackTrace();
           }
    }
```

```
	/**
	 * @param con: Conexión
	 * @param BDNombre: Nombre de la base de datos
	 *    Crea la tabla EQUIPOS
	 */
	public static void creaTablaEquipo(Connection con, String BDNombre)
			 throws SQLException
	{
		String creaTabla = "create table " + BDNombre + ".EQUIPO " +
					"(codEquipo INT NOT NULL PRIMARY KEY, " +
					"nombre varchar(40) NOT NULL, " +
					"estadio varchar(40) NOT NULL, " +
					"poblacion varchar(20) NOT NULL, "+
					"provincia varchar(20) NOT NULL, "+
					"codPostal char(5)) ";

		Statement stmt = null;
		try {
			stmt = con.createStatement();
			stmt.executeUpdate(creaTabla);
			System.out.println("Tabla EQUIPO creada.");
		}
		catch(SQLException e) {
			printSQLException(e);  // Ver apartado 5
		}
		finally {
			stmt.close();
		}
	}


	/**
	 * @param con: Conexión
	 * @param BDNombre: Nombre de la base de datos
	 *    Crea la tabla JUGADORES
	 */
	public static void creaTablaJugador(Connection con, String BDNombre)
			 throws SQLException
	{
		String creaTabla = "create table " + BDNombre + ".JUGADORES " +
					"(codJugador INT NOT NULL PRIMARY KEY, " +
					"codEquipo INT NOT NULL, " +
					"nombre varchar(40) NOT NULL, " +
					"dorsal int NOT NULL, "+
					"edad int NOT NULL, "+
					"provincia varchar(20) NOT NULL, "+
				"FOREIGN KEY (codEquipo) REFERENCES EQUIPO(codEquipo))";

		Statement stmt = null;
		try {
			stmt = con.createStatement();
			stmt.executeUpdate(creaTabla);
			System.out.println("Tabla JUGADORES creada.");
		}
		catch(SQLException e) {
			printSQLException(e); // Ver apartado 5
		}
		finally {
			stmt.close();
		}
	}
}
```

Los métodos `creaTablaEquipo` y `creaTablaJugador` crean las tablas `EQUIPO` y `JUGADORES`, respectivamente, utilizando para ello un objeto de tipo `Statement`. Sin embargo, dado que el método `executeUpdate()` ejecuta una sentencia SQL de tipo `CREATE TABLE`, ésta no actualiza ningún registro de la base de datos y por ello este método devuelve cero. En caso de ejecutar una sentencia de tipo `INSERT`, `UPDATE` o `DELETE`, el método devolvería el número de filas que resultasen afectadas por el cambio.

Una vez creada las tablas, el siguiente paso podría ser la introducción en las mismas de los datos. La estructura es prácticamente igual a las anteriormente vistas de creación de tablas, lo único que cambia es la sentencia SQL que se ejecuta.

Ejemplo: Carga de datos

Al ejecutar este ejemplo, se cargarán datos en las tablas EQUIPO y JUGADORES de la base de datos de nombre "mibase".

```
public static void main(String[] args) {
	try {
		Class.forName("com.mysql.jdbc.Driver");
		Connection conexion = DriverManager.getConnection
		    ("jdbc:mysql://localhost:3306/mibase","rosi","123");
		System.out.println("Conexión establecida");
		cargaEquipos(conexion, "mibase");
		cargaJugadores(conexion, "mibase");
	}

	catch (SQLException e2) {
		printSQLException(e2);
	}
	catch (Exception e1) {
		e1.printStackTrace();
	}


}

/**
 * Añade datos a la tabla equipos
 * @param con
 * @param BDNombre
 */
public static void cargaEquipos(Connection con, String BDNombre)
	 throws SQLException
{
	Statement stmt = null;
	try {
		stmt = con.createStatement();

		stmt.executeUpdate("insert into "+BDNombre+".EQUIPO values ("+
			"1,'ESTEPONA','MONTERROSO','ESTEPONA','MALAGA','29680')");

		stmt.executeUpdate("insert into "+BDNombre+".EQUIPO values ("+
			"2,'ALCORCON','SANTO DOMINGO','ALCORCON','MADRID','28924')");

		stmt.executeUpdate("insert into "+BDNombre+".EQUIPO values ("+
			"3,'PORCUNA','SAN CRISTOBAL','PORCUNA','JAEN','23790')");

		System.out.println("Datos cargados en la tabla EQUIPOS.");
	}
	catch(SQLException e) {
		printSQLException(e);
	}
	finally {
		stmt.close();  // Puede lanzar SQLExceptions, por eso he puesto el
				// throws en la cabecera del método
	}
}


/**
 * Añade datos a la tabla JUGADORES
 * @param con
 * @param BDNombre
 */
```

```
public static void cargaJugadores(Connection con, String BDNombre)
        throws SQLException
{
        Statement stmt = null;
        try {
                stmt = con.createStatement();

                // Cargando datos de Estepona
                stmt.executeUpdate("insert into "+BDNombre+".JUGADORES values ("+
                                "1,1,'JOSE ANTONIO',1,42)");

                stmt.executeUpdate("insert into "+BDNombre+".JUGADORES values ("+
                                "2,1,'IGNACIO',2,62)");

                stmt.executeUpdate("insert into "+BDNombre+".JUGADORES values ("+
                                "3,1,'DIEGO',3,20)");

                // Cargando datos de Alcorcón
                stmt.executeUpdate("insert into "+BDNombre+".JUGADORES values ("+
                                "4,2,'TURRION',1,37)");

                stmt.executeUpdate("insert into "+BDNombre+".JUGADORES values ("+
                                "5,2,'LUIS ABEL',2,37)");

                stmt.executeUpdate("insert into "+BDNombre+".JUGADORES values ("+
                                "6,2,'ISAAC',3,40)");

                // Cargando datos de Porcuna
                stmt.executeUpdate("insert into "+BDNombre+".JUGADORES values ("+
                                "7,3,'JUAN FRANCISCO',1,33)");

                stmt.executeUpdate("insert into "+BDNombre+".JUGADORES values ("+
                                "8,3,'PARRA',2,37)");

                stmt.executeUpdate("insert into "+BDNombre+".JUGADORES values ("+
                                "9,3,'RAUL',3,19)");

                System.out.println("Datos cargados en la tabla JUGADORES.");
        }
        catch(SQLException e) {
                printSQLException(e);
        }
        finally {
                stmt.close();  // Puede lanzar SQLExceptions, por eso he puesto el
                               // throws en la cabecera del método
        }
}
```

Ejemplo: Creación de una tabla y carga de datos a partir de un fichero

También es habitual, cargar los datos desde un fichero. Un ejemplo del código para leerlos e introducirlos en la base de datos sería el que se muestra a continuación, suponiendo que el formato en el que los datos están almacenados en el fichero es: nombre del cliente, clave de acceso y saldo, introducidos en líneas separadas, y seguidos de una línea separatoria como se muestra a continuación:

```
Iván Samuel Tejera Santana
nadiuska
1000000
-------------------------
Carmen López García
carmen
23000
-------------------------
Sebastian Moreno Fuentes
tornado
345000
```

```
import java.io.*;
import java.sql.*;

public class CreayCargaBanco {

   public static void main(String[] args) {

       try {
           String driver   = "com.mysql.jdbc.Driver";
           String url      = "jdbc:mysql://localhost:3306/mibase";
           String login    = "rosario";
           String password = "olmedo";

           Class.forName(driver);
           Connection conexion = DriverManager.getConnection(url, login, password);

           System.out.println("Conexión establecida");
           creaTablaBanco(conexion, "mibase");
           cargaTablaBanco(conexion, "mibase");
         }

        catch (SQLException e2) {printSQLException(e2); }
        catch (Exception e1) {e1.printStackTrace();}
       }


       /**
        *
        * @param con: Conexión
        * @param BDNombre: Nombre de la base de datos
        *
        * Crea la tabla BANCO
        */
       public static void creaTablaBanco(Connection con, String BDNombre)
              throws SQLException
       {
             String creaTabla = "create table " + BDNombre + ".BANCO " +
                 "(client VARCHAR(100) NOT NULL, "  +
                 "password VARCHAR(20) NOT NULL, " +
                 "balance INT NOT NULL, " +
                 " PRIMARY KEY(client))";

              Statement stmt = null;
              try {
                     stmt = con.createStatement();
                     stmt.executeUpdate(creaTabla);
                     System.out.println("Tabla BANCO creada.");
              }
              catch(SQLException e) {
                     printSQLException(e);
              }
              finally {
                     stmt.close();
              }
       }


       /**
        * Añade datos desde un fichero a la tabla BANCO
        * @param con
        * @param BDNombre
        */
       public static void cargaTablaBanco(Connection con, String BDNombre)
              throws SQLException
       {
              String client, password;
              int balance;
              BufferedReader br = null;
              Statement stmt = null;
```

```
            // Crea el objeto Statement
            stmt = con.createStatement();

            try {
               // Crea un flujo para poder recorrer el fichero datos.txt
               br = new BufferedReader(new FileReader("datos.txt"));

               do {
                   // Lee del fichero los datos de un cliente (nombre, clave, balance)
                    client = br.readLine();
                    password = br.readLine();
                    balance = Integer.parseInt(br.readLine());

                    // Forma la sentencia para insertar el cliente obtenido del fichero
                    String sqlString = "insert into "+BDNombre+".BANCO values ("
                              + "'"+ client + "', '" + password + "', "+ balance + ")";

                    System.out.println("Se ha ejecutado: "+sqlString);

                    // Añade el cliente al fichero
                    stmt.executeUpdate(sqlString);
                   }
               while (br.readLine() != null);  // Lee la línea de "-------"
            } // end try
            catch(IOException e) {
                  e.printStackTrace();
            }
            finally {
                  stmt.close();
                  try {
                     br.close();
                  }
                  catch(IOException e1) {
                        e1.printStackTrace();
                  }
            }
      }



      /**
       *
       * Muestra información más detallada sobre una excepción tipo SQLException
       */
      public static void printSQLException(SQLException ex)
      {
            ex.printStackTrace(System.err);
            System.err.println("SQLState: "+ex.getSQLState());
            System.err.println("Error code: "+ex.getErrorCode());
            System.err.println("Message: "+ex.getMessage());
            Throwable t = ex.getCause();
            while (t!=null) {
                  System.out.println("Cause: "+t);
                  t = t.getCause();
            }
      }
}
```

6.4. Consultar la base de datos

El objeto `Statement` devuelve un objeto `java.sql.ResultSet` que encapsula los resultados de la ejecución de una sentencia `SELECT`. Ésta interfaz es implementada por los vendedores de drivers. Dispone de métodos que permiten al usuario navegar por los diferentes registros que se obtienen como resultado de la consulta.

El método, `executeQuery`, definido en la interfaz `java.sql.Statement` nos permite ejecutar las instrucciones `SELECT`:

```
public ResultSet executeQuery (String sql) throws SQLException
```

La interfaz `java.sql.ResultSet` ofrece varios métodos para recuperar los datos que se obtienen de realizar una consulta: `getBoolean()`, `getInt()`, `getShort()`, `getByte()`, `getDate()`, `getDouble()` y `getfloat()`.

Todos estos métodos requieren el nombre de la columna (tipo String) o el índice de la columna (tipo int) como argumento. Así por ejemplo, la sintaxis para las dos variantes del método `getString()` es la siguiente:

- `public String getString(int columnIndex) throws SQLException`
- `public String getString(String columnName) throws SQLException`

El método `toString` puede servirnos para recuperar datos CHAR y VARCHAR de las bases de datos.

Regresando a la clase del apartado anterior, creemos un nuevo método `consultaBanco` que recupere todos los datos de la tabla `BANCO`.

```
public static void consultaBanco(Connection con, String BDNombre)
      throws SQLException
{
 //  String sqlString = "SELECT client, password, balance FROM BANCO";
     String sqlString = "SELECT * FROM BANCO";
     Statement statement = con.createStatement();
     ResultSet rs = statement.executeQuery(sqlString);

     while (rs.next()) {
         System.out.println(rs.getString("client") + " " +
                            rs.getString("password") + " " +
                            rs.getInt("balance"));
      }
     statement.close();
  }
```

Este método crea un objeto `Statement`, utilizado para invocar el método `executeQuery()` con una instrucción SQL `(SELECT)` como argumento. El objeto `java.sql.ResultSet` contiene todas las filas de la tabla `BANCO` que coinciden con la instrucción `SELECT`. Utilizando el método `next()` del objeto `ResultSet`, podemos recorrer todas las filas contenidas en el bloque de resultados. En cualquier fila, podemos utilizar uno de los métodos `getXXX()` descritos anteriormente para recuperar los campos de una fila.

Como ya se ha indicado, también podemos utilizar los métodos `getXXX()` utilizando en lugar del nombre de los campos, el orden que ocupan en la sentencia SQL, comenzando por el número 1. Así el ejemplo anterior podría haber quedado de la siguiente manera:

```
System.out.println(rs.getString(1) + " " +
                   rs.getString(2) + " " +
                   rs.getInt(3));
```

Este tipo de recuperación de información se utiliza cuando recuperamos **varias columnas que tienen el mismo nombre**, ya que de la primera forma no sería posible recuperar la información.

La interfaz `ResultSet` también permite conocer la estructura del bloque de resultados. El método `getMetaData()` ayuda a recuperar un objeto `java.sql.ResultSetMetaData` que tiene varios métodos para describir el bloque de resultados, algunos de los cuales se enumeran a continuación:

- `getTableName()`
- `getColumnCount()`
- `getColumnName()`
- `getColumnType()`

Tomando un bloque de resultados, podemos utilizar el método `getColumnCount()` para obtener el número de columnas de dicho bloque. Conocido el número de columnas, podemos obtener la información de tipo asociada a cada una de ellas.

Por ejemplo, el siguiente método imprime la estructura del bloque de resultados:

```
public static void getMetaData(Connection con) throws SQLException {

    String sqlString = "SELECT * FROM BANCO";
    Statement statement = con.createStatement();
    ResultSet rs = statement.executeQuery(sqlString);
    ResultSetMetaData metaData = rs.getMetaData();
    int noColumns = metaData.getColumnCount();
    for (int i=1; i<noColumns+1; i++) {
        System.out.println(metaData.getColumnName(i)
                           + " " +
                           metaData.getColumnType(i));
    }
    statement.close();
  }
```

El método anterior obtiene el número de columnas del bloque de resultados e imprime el nombre y el tipo de cada columna. En este caso, los nombres de columna son `client`, `password` y `balance`. Observe que los tipos de columna son devueltos como números enteros. Por ejemplo, todas las columnas de tipo `VARCHAR` retornarán el entero 12, las del tipo `DATE`, 91. Estos tipos son constantes definidas en la interfaz `java.sql.Types`. Fíjese también en que los números de columnas empiezan desde 1 y no desde 0.

6.4.1. El interfaz ResultSet

La interfaz `ResultSet` como hemos visto, tiene métodos para recuperar y manipular los datos relativos a comandos SQL realizados a una base de datos. Existen distintos tipos de objetos `ResultSet` dependiendo de sus características, siendo estos:

- **TYPE_FORWARD_ONLY**. Este cursor es el cursor por defecto. Los ejemplos anteriores están realizados con este cursor. Como su nombre indica, es un cursor unidireccional y sólo se mueve en un sentido hacia delante (desde la primera fila hasta la última).
- **TYPE_SCROLL_INSENSITIVE**. Este cursor puede moverse hacia delante y hacia detrás (*forward y backward*) siempre teniendo en cuenta la posición en la que se encuentra el cursor. Aunque los datos con los que está trabajando cambien en la base de datos no le afectará. Contiene los datos que se recuperaron cuando se ejecutó el comando SQL. *(Ver apartado 6.4.2).*
- **TYPE_SCROLL_SENSITIVE**. Este cursor puede moverse hacia delante y hacia detrás, la diferencia radica que cuando los datos con los que está trabajando cambian, en la base de datos el cursor al moverse trabaja con los datos más actuales reflejando los últimos cambios realizados. *(Ver apartado 6.4.2).*

Concurrencia

Determina si los datos del `ResultSet` son actualizables en la base de datos o no. Existen dos niveles de concurrencia:

- **CONCUR_READ_ONLY**. Es el tipo de concurrencia por defecto. El objeto `ResultSet` NO puede ser actualizado utilizando la interfaz `ResultSet`.
- **CONCUR_UPDATABLE.** El objeto `ResultSet` puede ser actualizado utilizando el interface `ResulSet`.

No todos los drivers JDBC soportan la concurrencia con base de datos. El método `DatabaseMetaData.supportsResultSetConcurrency`, devolverá *true*, si el nivel de concurrencia es soportado por el driver y *false* en caso contrario.

Persistencia

Cuando se llama al método `Connection.commit` esto puede implicar que los objetos `ResultSet` que estaban abiertos en la transacción se cierren. Esto puede provocar errores en el programa. Mediante la propiedad *holdability* del cursor se puede especificar el funcionamiento del cursor cuando se ejecuta un *commit*.

Cuando se llama a los métodos `createStatement`, `prepareStatement` o `prepareCall` del objeto `Connection` se le pueden pasar las siguientes constantes:

- **HOLD_CURSORS_OVER_COMMIT**. Los cursores `ResultSet cursors` NO se cerrarán cuando se ejecute el método `commit`.
- **CLOSE_CURSORS_AT_COMMIT**. Los cursores `ResultSet cursors` SI se cerrarán cuando se ejecute el método `commit`.

El tipo de persistencia varía dependiendo del gestor de base de datos. Algunas bases de datos no soportan alguno de estos tipos de persistencia.

6.4.2. Los cursores.

Como ya hemos visto, el acceso a los datos mediante `ResultSet` se denomina cursor. No se debe confundir este cursor con los cursores de bases de datos. Son dos cosas diferentes. El cursor del que estamos hablando es un puntero a una zona de memoria donde residen los datos recuperados por el comando SQL. Inicialmente se coloca en una posición anterior a la primera posición de los datos recuperados y mediante la llamada al método `ResultSet.next()` vamos posicionándonos en la siguiente fila de los datos recuperados. Esto se suele hacer utilizando un bucle. Al final del bucle cuando ya no existen más datos el método `next()` devuelve false.

También hemos visto que los cursores por defecto son unidireccionales y sólo se mueven hacia delante. No obstante, se pueden crear cursores bidireccionales que pueden utilizar otros métodos para desplazarse por los datos como son:

- **`next()`**. Mueve el cursor una posición hacia delante. Devuelve *true* si el cursor está posicionado en una fila y *false* en caso de que esté después de la última fila. Es el único método que se puede llamar cuando se crea un cursor por defecto (TYPE_FORWARD_ONLY).
- **`previous()`**. Mueve el cursor una posición hacia atrás. Devuelve *true* si el cursor está posicionado en una fila y *false* en caso de que esté antes de la primera fila.
- **`first()`**. Coloca el cursor en la primera fila. Devuelve *true* si el cursor contiene al menos una fila y *false* en caso contrario.
- **`last()`**. Coloca el cursor en la última fila. Devuelve *true* si el cursor contiene al menos una fila y *false* en caso contrario.

- **`beforeFirst()`**. Coloca el cursor antes de la primera fila.
- **`afterLast()`**. Coloca el cursor después de la primera fila.
- **`relative(int rows)`**. Mueve el cursor *rows* de forma relativa a la posición actual.
- **`absolute(int row)`**. Coloca el cursor en la posición especificada en el parámetro *row*.

6.5. Modificar y actualizar la base de datos

La modificación de una tabla en una base de datos es similar a la ejecución de otras sentencias como las inserciones y borrados de tablas. Únicamente cambia la sintaxis SQL.

Ejemplo: Actualización de una columna

```
public static void modificaEquipo(Connection con, String BDNombre)
              throws SQLException
{
       Statement stmt = null;
       try {
              stmt = con.createStatement();
              stmt.executeUpdate("UPDATE " + BDNombre +
                            ".EQUIPO SET ESTADIO='ALBORAN'" +
                            " WHERE codEquipo = 1");
              System.out.println("Equipo actualizado");
       }
       catch(SQLException e) {
              printSQLException(e);
       }
       finally {
              stmt.close();
       }

 }
```

Este ejemplo modifica el nombre del *"estadio"* del equipo con un *"codEquipo"* igual a 1.

Ejemplo: Modificar datos en una tabla utilizando ResultSet

```
public static void modificaEdadJugadores(Connection con, String BDNombre,
               int incremento) throws SQLException
{
       Statement stmt = null;
       try {
              stmt = con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
                                         ResultSet.CONCUR_UPDATABLE);

              ResultSet rs = stmt.executeQuery(
                             "SELECT * FROM " + BDNombre + ".JUGADORES");

              while (rs.next()) {
                     int i = rs.getInt("EDAD");
                     rs.updateInt("EDAD", i+incremento);
                     rs.updateRow();
              }
              System.out.println("Edades actualizadas");
       }
       catch(SQLException e) {
                     printSQLException(e);
       }
       finally {
              stmt.close();
       }
}
```

En este ejemplo se actualiza la edad de los jugadores y se le suma el valor introducido en el parámetro *"incremento"*, para ello se está creando un `ResultSet` bidireccional y actualizable. Como ya se ha estudiado la propiedad TYPE_SCROLL_SENSITIVE hace que el objeto `ResultSet` creado pueda moverse bidireccionalmente de forma relativa a su posición actual y la propiedad CONCUR_UPDATABLE hace que se puedan modifiar los datos del cursor y estos se repliquen a la base de datos. Hasta que no se invoca al método `ResultSet.updateRow` no se actualizará la base de datos.

Ejemplo: Insertar datos en una tabla utilizando ResultSet

```
public static void insertaJugador(Connection con, String BDNombre, int codJu,
              int codEq, String nombre, int dorsal, int edad) throws SQLException
{
       Statement stmt = null;
       try {
              stmt = con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
                                         ResultSet.CONCUR_UPDATABLE);

              ResultSet rs = stmt.executeQuery(
                 "SELECT * FROM " + BDNombre + ".JUGADORES");

              rs.moveToInsertRow();
              rs.updateInt("codJugador", codJu);
              rs.updateInt("codEquipo", codEq);
              rs.updateString("nombre", nombre);
              rs.updateInt("dorsal", dorsal);
              rs.updateInt("edad", edad);
              rs.insertRow();
              rs.beforeFirst();
              System.out.println("Jugador añadido");
       }
       catch(SQLException e) {
              printSQLException(e);
       }
       finally {
              stmt.close();
       }
}
```

Es posible que el driver JDBC utilizado no tenga la posibilidad de insertar datos en la base datos. En ese caso, se lanza la excepción `SQLFeatureNotSupportedException`. En el método anterior se puede observar cómo se pueden realizar inserciones de datos en una base de datos. Al ejecutar al método `inserRow()` del objeto `ResultSet` se insertarán los datos tanto en el `ResultSet` como en la base de datos.

7. TRANSACCIONES

Si hay una propiedad que distingue una base de datos de un sistema de archivos, esa propiedad es la capacidad de soportar transacciones. Si está escribiendo en un archivo y el sistema operativo cae, es probable que el archivo se corrompa. Si está escribiendo en un archivo de base de datos, utilizando correctamente las transacciones, se asegura que, o bien el proceso se completará con éxito, o bien la base de datos volverá al estado en el que se encontraba antes de comenzar a escribir en ella.

Cuando múltiples instrucciones son ejecutadas en una única transacción, todas las operaciones pueden ser realizadas (convertidas en permanentes en la base de datos) o descartadas (es decir, se deshacen los cambios aplicados a la base de datos).

Cuando se crea un objeto `Connection`, éste está configurado para realizar automáticamente cada transacción (está habilitado el modo ***auto-commit***). Esto significa que cada vez que se ejecuta una instrucción, se realiza en la base de datos y no puede ser deshecha. Los siguientes métodos en la interfaz `Connection` son utilizados para gestionar las transacciones en la base de datos:

- `void setAutoCommit(boolean autoCommit).` Permite habilitar/deshabilitar el modo auto-commit.
- `void commit() throws SQLException`
- `void rollback() throws SQLException`

Para iniciar una transacción, invocamos `setAutoCommit(false).` Esto nos otorga el control sobre lo que se realiza y cuándo se realiza. Una llamada al método `commit()` realizará todas las instrucciones emitidas desde la última vez que se invocó el método `commit().` Por el contrario, una llamada `rollback()` deshará todos los cambios realizados desde el último `commit().` Sin embargo, una vez se ha emitido una instrucción `commit()`, esas transacciones no pueden deshacerse con `rollback().`

Consideremos un caso práctico de empresa consistente en crear un pedido, actualizar el inventario y crear registro de envíos. La lógica de empresa puede requerir que todo sea efectivo o que falle. El fracaso de la creación de un registro de envíos puede dictar que no cree un pedido. En tales casos, los efectos de las instrucciones SQL correspondientes a las dos primeras tareas (crear un pedido y actualizar el inventario) deben deshacerse.

El siguiente fragmento de código ilustra esta situación:

```
Connection connection = null;

// Obtener una conexión
...
try{
     // Iniciar una transacción
     connection.setAutoCommit(false);

     Statement statement = connection.createStatement();

     // Crear un pedido
     statement.executeUpdate(
         "INSERT INTO ORDERS(ORDER ID,
         "PRODUCT ID,...) VALUES(...)");

     // Actualizar el inventario
     statement.executeUpdate("UPDATE TABLE INVENTORY " +
                             "SET QUANTITY = QUANTITY-1 "
                                  + "WHERE PRODUCT ID = ...");

     // Crear un registro de envíos
     if (...) {
          // Operación éxitosa
          statement.execute(
               "INSERT INTO SHIP RECORD(...) " +
               "VALUES (...)");
          connection.commit();
     } else {
          // Deshacer operación
          connection.rollback();
     }
} catch (SQLException) {
     // Manejar excepciones aquí
} finally {
     // Cerrar instrucción y conexión
}
```

En este fragmento de código, una vez invocado el método `rollback(),` la base de datos restaura las tablas `ORDERS` e `INVENTORY` a su estado anterior. `commit()` convierte en permanentes los cambios efectuados por las instrucciones `INSERT` y `UPDATE`.

8. FUNCIONES Y PROCEDIMIENTOS

En SQL es muy común utilizar funciones en los comandos como por ejemplo count(*), sum (columna), avg(columna), etc. En ocasiones, el programador necesita realizar operaciones muy concretas que devuelven un valor y en ese caso se utilizan funciones almacenadas. Estas funciones almacenadas son muy parecidas a los procedimientos almacenados aunque su forma de utilización es diferente. Las diferencias entre funciones y procedimientos almacenados son las siguientes:

- Las funciones siempre devuelven un valor mientras que los procedimientos no. El tipo de este valor se define cuando se declara la función. Obviamente este tipo de dato tiene que ser un tipo de datos de la base de datos (en nuestros ejemplos MySQL).
- Desde una sentencia SQL se puede llamar a una función pero no a un procedimiento almacenado.
- Las funciones devuelven un valor pero nunca un ResultSet.
- Las funciones tienen sólo parámetros de entrada. El único parámetro de salida por así decirlo es el dato que devuelve la función (aunque el valor de salida no se puede considerar como parámetro).

La sintaxis de las funciones y procedimientos almacenados difieren de un SGBDR a otro, incluso de una versión a otra podrían cambiar. Debéis consultar los manuales de referencia de cada gestor cuando vayáis a realizar alguna función o procedimiento.

8.1. Funciones

La sintaxis de las funciones almacenadas en MySQL es la siguiente:

```
CREATE FUNCTION nombre_de_funcion(parámetro1, parámetro2, …, parámetro N)
       RETURNS tipo(CHAR|INT|FLOAT|DECIMAL|…)

       BEGIN
              ...
       RETURN valor
END
```

Un ejemplo de función almacenada en MySQL, es el que se define a continuación. En esta función almacenada se cuenta y se devuelve el número de filas que existe en la tabla jugadores.

```
DELIMITER //
CREATE FUNCION dimeCuantos()
RETURNS INT
BEGIN
   DECLARE j INT;
   SELECT COUNT(*) INTO j FROM jugadores;
   RETURN j;
END //
```

Nótese que se utiliza como delimitador “//” dado las sentencias SQL utilizan el delimitador por defecto ‘;’.

Para comprobar si funciona nuestra función se puede ejecutar la siguiente consulta SQL:

```
SELECT dimeCuantos() FROM jugadores;
```

El siguiente método muestra la forma de utilización de funciones almacenadas en una base de datos.

```
public static void getJugadores(Connection con) throws SQLException
{
    Statement stmt = null;
    // String query = "select dimeCuantos() FROM dual";
    String query = "select dimeCuantos() FROM jugadores";

    try {
        stmt = con.createStatement();
        ResultSet rs = stmt.executeQuery(query);
        while (rs.next()) {
            String cuantos = rs.getString(1);
            System.out.println("Existen "+cuantos+
                                    " jugadores en nuestra bases de datos");
            System.out.println("*********************************************");
       }
    }
    catch(SQLException e) {
       printSQLException(e);
     }
    finally {
       stmt.close();
     }
}
```

8.2. Procedimientos

Un procedimiento almacenado es una serie de comandos que realizan una tarea concreta en una base de datos. Los procedimientos almacenados generalmente se crean cuando existe una tarea que se va a realizar muchas veces.

En ocasiones es bueno utilizar procedimientos almacenados para estandarizar tareas complejas en una base de datos. Imaginemos que en un banco tenemos una serie de clientes a los cuales se les va a dar una gratificación, descuento o prima dependiendo de los productos que tengan contratados, saldo medio, … Si esa operación se va a realizar desde varias aplicaciones es posible que alguna de ella lo calcule de forme diferente. Si se estandariza y se incluye en un procedimiento almacenado el proceso se centraliza y estaremos seguros que no existen diferentes formas de hacer lo mismo.

La sintaxis a grandes rasgos es la siguiente:

```
CREATE PROCEDURE nombre_de_procedimiento([parámetro1], ...)
      BEGIN
      CODIGO DEL PRODEDIMIENTO
END
```

Los parámetros se definen de la siguiente manera:

```
[IN | OUT | INOUT | NOMBRE TIPO_DE_DATO]
```

En el siguiente código se muestra un procedimiento almacenado. Se ha creado un procedimiento similar a la función almacenada del apartado anterior.

```
DELIMITER //
CREATE PROCEDURE cuantosJugadores(OUT c INT)
BEGIN
   SELECT COUNT(*) INTO c FROM jugadores;
END //
```

En este caso el procedimiento almacenado tiene un parámetro de salida (OUT) y devuelve el número de filas de la tabla jugadores. El programa Java que ejecute dicho procedimiento deberá de recoger dicho valor de salida.

```
public static void getJugadoresProc(Connection con) throws SQLException
{
    try {
       CallableStatement cs = con.prepareCall("{call cuantosJuagadores (?)}");
       cs.registerOutParameter(1, Types.INTEGER);
       cs.execute();
       int cuantos = cs.getInt(1);
       System.out.println("Existen "+cuantos+
             " jugadores en nuestra bases de datos");
       System.out.println("*************************************************");
    }
    catch(SQLException e) {
       printSQLException(e);
    }
}
```

En el ejemplo anterior se puede observar cómo se invoca el procedimiento almacenado *cuantosJugadores* desde el método *getJugadoresProc*.

Anteriormente hablamos de parámetros de entrada, salida y entrada/salida. La llamada a procedimientos almacenados con diferentes tipos de parámetros no es la misma. En el siguiente ejemplo se puede ver cómo se realizarán dichas llamadas.

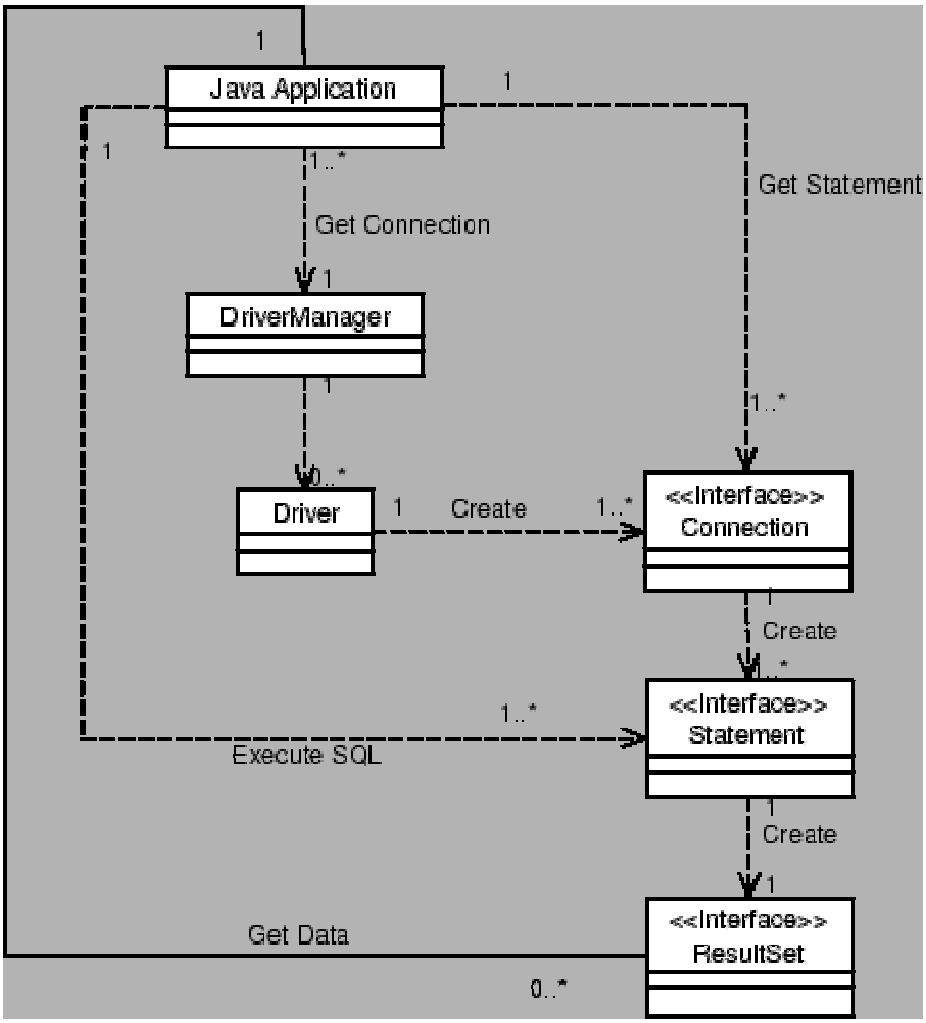
```
       try {
             CallableStatement cs;

             // Llamada a un procedimiento sin parámetros
                cs = con.prepareCall("{call miproc}");
                cs.execute();

             // Llamada a un procedimiento con un parámetro OUT
                 cs = con.prepareCall("{call miproc_out(?)}");
                 // Hay que registar el parámetro OUT con su tipo
                 cs.registerOutParameter(1, Types.VARCHAR);
                 // Ejecutar el procedimiento y recuperar el parámetro
                 cs.execute();
                 String dato = cs.getNString(1);

             // Llamada a un procedimiento con un parámetro IN
                 cs = con.prepareCall("{call miproc_in(?)}");
                 // Hay que actualizar el valor del parámetro IN
                 cs.setString(1, "Hola");
                 // Ejecutar el procedimiento
                 cs.execute();

              // Llamada a un procedimiento con un parámetro IN / OUT
                 cs = con.prepareCall("{call miproc_inout(?)}");
                 // Hay que registar el parámetro IN/OUT con su tipo
                 cs.registerOutParameter(1, Types.VARCHAR);
                 // Hay que actualizar el valor del parámetro IN/OUT
                 cs.setString(1, "Hola");
                 // Ejecutar el procedimiento
                 cs.execute();
                 String dato1 = cs.getNString(1);
       }
       catch (SQLException e) {
             printSQLException(e);
       }
}
```

Los objetos de la clase ***CallableStatement*** son utilizados en Java para llamar a procedimientos almacenados de la base de datos.

9. ARQUITECTURA GENERAL DE UNA APLICACIÓN JDBC

En las secciones anteriores se han visto los conceptos básicos necesarios para poder interactuar con una base de datos. En esta sección, lo que se pretende es mostrar la arquitectura general de una aplicación Java que utilice la API JDBC.

En la siguiente figura se ilustra en un *diagrama de clases* las relaciones de dependencia y asociación que se establecen entre las clases que integran la API JDBC y nuestra aplicación Java.



En primer lugar, se puede observar cómo existe una relación de dependencia entre nuestra aplicación y las clases e interfaces `DriverManager`, `Connection` y `Statement`. Ello es así puesto que si no dispusiéramos de la clase `DriverManager` no podríamos crear conexiones con la base de datos. Estas conexiones se obtienen haciendo uso de la clase `Driver` que nos proporciona el fabricante del RDBMS, tal y cómo se vio anteriormente, pudiendo obtener en nuestra aplicación distintas conexiones a diferentes bases de datos.

Las conexiones son representadas por medio de objetos que implementan la interfaz `Connection`. Estos objetos son necesarios en la aplicación JDBC ya que a través de los mismos se lleva a cabo el acceso a la base de datos. Sin ellos, no sería posible crear los objetos `Statement` que contienen las sentencias SQL objeto de ejecución en el servidor.

Del resultado de la ejecución de la sentencia SQL que permite un objeto `Statement` se obtienen 0 o más resultados representados por un objeto que implementa la interfaz `ResultSet`, a los cuales podemos acceder desde la aplicación Java.

Las dependencias que se establecen entre la aplicación Java y las clases JDBC son *dependencias de uso* (`use`) e instanciación (`instantiate`) porque la aplicación requiere de dichas clases para poder desempeñar correctamente el fin para el que fue diseñada.