	MEDIA TÉCNICA DESARROLLO DE SOFTWARE GUIA DE APRENDIZAJE # No.0 Módulo Elementos de software 1 TEMA: **Java-DBC**	

Docente: Juan Carlos Pérez P.

Alumno : _____**Fecha :** _____**Nota:**____

Justificación*:* Se pretende con éste contribuir a que el alumno reconozca los fundamentos teóricos sobre la conexión a bases de datos con jdbc/obdc

Objetivos: Reconocer los conceptos básicos de JDBC.
Identificar el mecanismo básico de JDBC.

CONTENIDO

¿Qué es JDBC?

Java Database Connectivity (JDBC) es una interfaz de acceso a bases de datos_conexión SQL que proporciona un acceso uniforme a una gran variedad de bases de datos relacionales. JDBC también proporciona una base común para la construcción de herramientas y utilidades de alto nivel.

JDBC es una API de Java para ejecutar sentencias SQL. Está formado por un conjunto de clases e interfaces programadas con el propio Java. Permite interactuar con bases de datos, de forma transparente al tipo de la misma.

Como punto de interés JDBC es una marca registrada y no un acrónimo, no obstante a menudo es conocido como "Java Database Connectivity".

JDBC suministra una API estándar para los desarrolladores y hace posible escribir aplicaciones de base de datos usando una API puro Java. Es decir, es una forma única de programar el acceso a bases de datos desde Java, independiente del tipo de la base de datos.

JDBC realiza llamadas directas a SQL.

Un único programa escrito usando la API JDBC y el programa será capaz de enviar sentencias SQL a la base de datos apropiada. Y, con una aplicación escrita en el lenguaje de programación Java, tampoco es necesario escribir diferentes aplicaciones para ejecutar en diferentes plataformas. La combinación de Java y JDBC permite al programador escribir una sola vez y ejecutarlo en cualquier entorno.

Java, siendo robusto, seguro, fácil de usar, fácil de entender, y Descargable automáticamente desde la red, es un lenguaje base excelente para aplicaciones de base de datos. JDBC expande las posibilidades de Java.

ASPECTOS BÁSICOS DE JDBC.

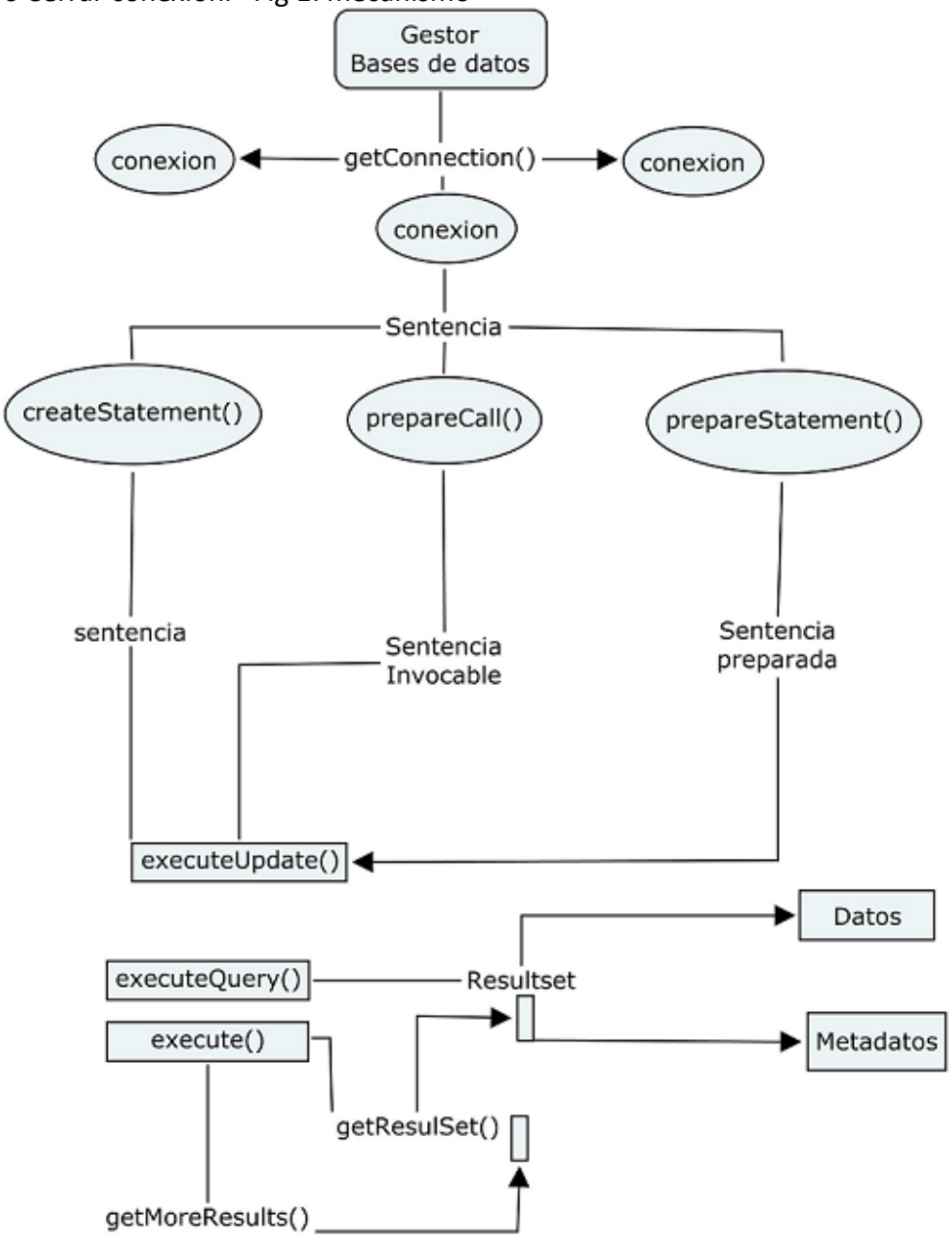
JDBC es una API de bajo nivel y una base para API's de alto nivel.

JDBC es una interfaz de bajo nivel, lo que quiere decir que se usa para 'invocar' o llamar a comandos SQL directamente. En esta función trabaja muy bien y es más fácil de usar que otras API's de conexión a bases de datos, pero está diseñado de forma que también sea la base sobre la cual construir interfaces y herramientas de alto nivel. Una interfaz de alto

nivel es ‘amigable’, usa una API mas entendible o más conveniente que luego se traduce en la interfaz de bajo nivel tal como JDBC.

Mecanismo JDBC.

1 Establecer conexión.
2 Crear sentencia.
3 Ejecutar sentencia.
4 Procesar resultados.
5 Finalizar sentencia.
6 Cerrar conexión. Fig 1: mecanismo



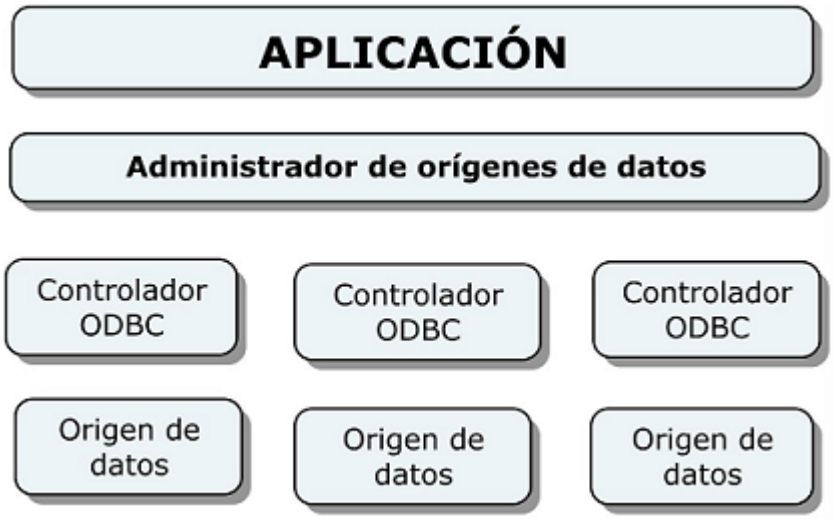
Instalación de JDBC.
Para acceder a BBDD en Java es necesario:
Tener instalado el entorno de desarrollo de java (JDK).
Tener instalada la API JDBC (normalmente incluido en el SDK).
Disponer del driver java para el gestor de BD escogido o usar el puente JDBC-ODBC.
Además de la API JDBC, el driver del gestor es una de las partes fundamentales para al acceso a los datos. Es posible usar un driver java específico del gestor a acceder o se puede usar el driver JDBC-ODBC incluido en la API.

Drivers JDBC.

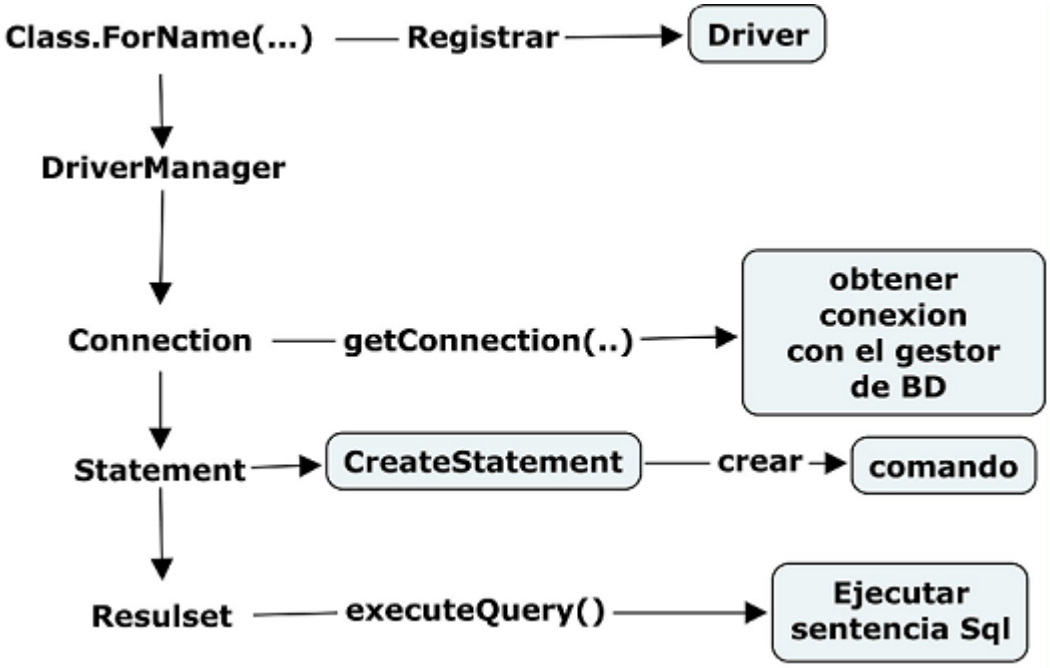
Para usar JDBC con un sistema gestor de base de datos en particular, es necesario disponer del driver JDBC apropiado que haga de intermediario entre ésta y JDBC. Dependiendo de varios factores.
Hay cuatro tipos de drivers distintos:

1**.- Puente JDBC-ODBC más driver ODBC:** Java suministra acceso vía
drivers ODBC. Nótese que el código binario ODBC, y en muchos casos el código cliente de base de datos, debe cargarse en cada máquina cliente que use este driver. Como resultado, este tipo de driver es el más apropiado en un red corporativa donde las instalaciones clientes no son un problema mayor, o para una aplicación en el servidor escrito en Java en una arquitectura en tres-niveles.

Fig2: Arquitectura ODBC



Este driver se encuentra dentro de la API JDBC. Y con él se puede acceder a cualquier gestor de BBDD mediante fuentes ODBC del propio S.O. donde se ejecute la aplicación.
Usar este tipo de driver requiere la configuración de una fuente ODBC en el SO y por consiguiente sería necesario que el cliente que vaya a usarla configure la fuente de datos ODBC antes de ejecutar la aplicación.
Una vez tenemos los requisitos previos y el driver a usar, pasamos a la carga del driver y al acceso a la BD.



2.- **driver Java parcialmente Nativo.** Este tipo de driver convierte llamadas JDBC en llamadas de la API cliente para Oracle,Mysql , Sybase, Informix, DB2 y otros DBMS. Nótese que como el driver puente, este estilo de driver requiere que cierto código binario sea cargado en cada máquina cliente.

3.- **driver Java nativo JDBC-Net.**

Este driver traduce llamadas JDBC al protocolo de red independiente del DBMS que después es traducido en el protocolo DBMS por el servidor. Este middleware en el servidor de red es capaz de conectar a los clientes puros Java a muchas bases de datos diferentes. El protocolo específico usado dependerá del vendedor. En general esta es la alternativa más flexible.

4.- **driver puro Java y nativo-protocolo**. Este tipo de driver convierte llamadas JDBC en el protocolo de la red usado por DBMS directamente. Esto permite llamadas directas desde la máquina cliente al servidor DBMS y es la solución más práctica para accesos en

intranets. Dado que muchos de estos protocolos son propietarios, los fabricantes de bases de datos serán los principales suministradores.

Es recomendable que las alternativas driver Java nativo JDBC-Net y driver puro Java y nativo-protocolo sean las formas preferidas de acceder a las bases de datos desde JDBC. Las categorías *Puente JDBC-ODBC más driver ODBC* y driver Java parcialmente Nativo son soluciones interinas cuando no están disponibles drivers directos puros Java.

CONEXIÓN.

Un objeto **Connection** representa una conexión con una base de datos. Una sesión de conexión incluye las sentencias SQL que se ejecutan y los resultados que son devueltos después de la conexión. Una única aplicación puede tener una o más conexiones con una única base de datos, o puede tener varias conexiones con varias bases de datos diferentes.

Apertura de una conexión.

La forma estándar de establecer una conexión a la base de datos es mediante la llamada al método **DriverManager.getConnection**. Este método toma una cadena que contiene una URL.

La clase DriverManager, referida como la capa de gestión JDBC, intenta localizar un driver que pueda conectar con la base de datos representada por la URL. La clase DriverManager mantiene una lista de clases Driver registradas y cuando se llama al método getConnection, se chequea con cada driver de la lista hasta que encuentra uno que pueda conectar con la base de datos especificada en la URL. El método connect de Driver usa esta URL para establecer la conexión.

Un usuario puede evitar la capa de gestión de JDBC y llamar a los métodos de *Driver* directamente. Esto puede ser útil en el caso raro que dos *drivers* puedan conectar con la base de datos y el usuario quiera seleccionar uno explícitamente. Normalmente, de cualquier modo, es mucho más fácil dejar que la clase DriverManager maneje la apertura de la conexión.

EJEMPLO:

El siguiente código muestra como ejemplo una conexión a la base de datos localizada en la URL “jdbc:odbc:wombat” con un user ID de “oboy” y password “12java”.

```
String url = "jdbc:odbc:wombat";
Connection con = DriverManager.getConnection(url, "oboy", "12Java");
```

Envío de Sentencias SQL.

Una vez que la conexión se haya establecido, se usa para pasar sentencias SQL a la base de datos subyacente. JDBC no pone ninguna restricción sobre los tipos de sentencias que pueden enviarse: esto da un alto grado de flexibilidad, permitiendo el uso de sentencias específicas de la base de datos o incluso sentencias no SQL. Se requiere de cualquier modo, que el usuario sea responsable de asegurarse que la base de datos subyacente sea capaz de procesar las sentencias SQL que le están siendo enviadas y soportar las consecuencias si no es así, en cuyo caso se generará una excepción.

Una vez terminados los Statements y la Connection, es IMPRESCINDIBLE llamar a close() en ambos casos. Esto liberará recursos, normalmente bastante antes de que lo haga el GC(*Garbage Collector*).

Clases para el envío de sentencias.

JDBC suministra tres clases para el envío de sentencias SQL y tres métodos en la interfaz Connection para crear instancias de estas tres clases. Estas clases y métodos son los siguientes:

Statement – creada por el método createStatement. Un objeto Statement se usa para enviar sentencias SQL simples

•Para ejecutar una consulta (query), se llama al método **executeQuery (sentenciaSQL)** sobre un Statement.

•Para ejecutar una modificación, se llama al método **executeUpdate (sentenciaSQL)** sobre un Statement.

EJEMPLO DE CONSULTA :

```
Connection con = DriverManager.getConnection(url, "oboy", "12Java");
Statement statement = con.createStatement();
ResultSet rs = statement.executeQuery ("SELECT * from PRODUCTOS");
```

PreparedStatement – creada por el método prepareStatement- Un objeto PreparedStatement se usa para sentencias SQL que toman uno o más parámetros como argumentos de entrada (parámetros IN). PreparedStatement tiene un grupo de métodos que fijan los valores de los parámetros IN, los cuales son enviados a la base de datos cuando se procesa la sentencia SQL.

Instancias de PreparedStatement extienden Statement y por tanto heredan los métodos de Statement. Un objeto PreparedStatement es potencialmente más eficiente que un objeto Statement porque este ha sido precompilado y almacenado para su uso futuro.

Son muy útiles cuando una sentencia SQL se ejecuta muchas veces cambiando sólo algunos valores.

EJEMPLO:

```
PreparedStatement ps = con.prepareStatement(
"select * from Propietarios where DNI=? AND NOMBRE=? AND EDAD=?”);
ps.setString(1, dni);
ps.setString(2, nombre);
ps.setInt(3, edad);
ResultSet rs= ps.executeQuery();
```

CallableStatement – creado por el método prepareCall. Los objetos CallableStatement se usan para ejecutar procedimientos almacenados SQL – un grupo de sentencias SQL que son llamados mediante un nombre, algo parecido a una función - . Un objeto CallableStatement hereda métodos para el manejo de los parámetros IN de PreparedStatement, y añade métodos para el manejo de los parámetros OUT e INOUT.

EJEMPLO:

```
String createProcedure = "create procedure SHOW_SUPPLIERS " +
"as " + "select SUPPLIERS.SUP_NAME,
COFFEES.COF_NAME " + "from SUPPLIERS, COFFEES " +
"where SUPPLIERS.SUP_ID = COFFEES.SUP_ID " + "order by SUP_NAME";
CallableStatement cs = con.prepareCall("{call SHOW_SUPPLIERS}");
ResultSet rs = cs.executeQuery();
```

Uso de los métodos.

El método createStatement se usa para:
Sentencias SQL simples (sin parámetros).
El método prepareStatement se usa para:
Sentencias SQL con uno ó más parámetros IN
Sentencias SQL simples que se ejecutan frecuentemente
El método prepareCall se usa para:
Llamar a procedimientos almacenados.

Transacciones.

Una transacción consiste en una o más sentencias que han sido ejecutadas, completas y, o bien se ha hecho commit o bien roll-back. Cuando se llama al método commit o rollback , la transacción actúal finaliza y comienza otra.
Una conexión nueva se abre por defecto en modo auto-commit, y esto significa que cuando se completa se llama automáticamente al método commit. En este caso, cada sentencia es 'commitada' individualmente, por tanto una transacción se compone de una única sentencia. Si el modo auto-commit es desactivado, la transacción no terminará hasta que se llame al método commit o al método rollback explícitamente, por lo tanto incluirá todas las sentencias que han sido ejecutadas desde la última invocación a uno de los métodos commit o rollback.
En este segundo caso, todas las sentencias de la transacción son "commitadas' o deshechas en grupo.
El método commit hace permanente cualquier cambio que una sentencia SQL realiza en la base de datos, y libera cualquier bloqueo mantenido por la transacción. El método rollback descarta estos cambios.
A veces un usuario no quiere que tenga efecto un determinado cambio a menos que se efectué otro. Esto puede hacerse desactivando el modo auto-commit y agrupando ambas actualizaciones en una transacción. Si ambas actualizaciones tienen éxito se llama al Método commit haciendo estos cambios permanentes, si uno o los dos fallan, se llama al método rollback, restaurando los valores existentes l inicio de la transacción. Muchos drivers JDBC soportan transacciones.

Clases de JDBC.
Esquema de clases: Fig 3: Clases de JDBC

Tipo	Clase JDBC
Implementación	**Java.sql.Driver** **Java.sql.DriverManager** **Java.sql.Driver.PropertyInfo**
Conexión Bases de datos	**Java.sql.Connection**
Sentencias SQL	**Java.sql.Statement** **Java.sql.PreparedStatement** **Java.sql.CallableStatement**
Datos	**Java.sql.ResultSet**
Errores	**Java.sql.SQLException** **Java.sql.SQLWarning**

DriverManager.
La clase DriverManager implementa la capa de gestión de JDBC, y trabaja como intermediaria entre el usuario y los drivers. Guarda la lista de los drivers que están disponibles y establece la conexión entre la base de datos y el driver apropiado. Además la clase DriverManager se ocupa de cosas cómo gestionar los límites de tiempo de 'login' en el driver y de la salida de los mensajes de traza y log.

Para aplicaciones simples, el único método en esta clase que necesita un programador general para su uso directamente es **DriverManager.getConnection**. Como su nombre indica, este método establece una conexión con la base de datos. JDBC permite al usuario llamar a los métodos de **DriverManager getDriver, getDrivers** y **registerDriver** así como al método de Driver connect, pero en la mayoría de los casos es preferible dejar que la clase **DriverManager** gestione los detalles al establecer la conexión.

Establecer una conexión.
Una vez que la clase Driver ha sido cargada y registrada con la clase **DriverManager**, se está en condiciones de establecer la conexión con la base de datos. La solicitud de la conexión se realiza mediante una llamada al método **DriverManager.getConnection**, y DriverManager testea los drivers registrados para ver si puede establecer la conexión.
A veces puede darse el caso de que más de un driver JDBC pueda establecer la conexión para una URL dada.
El orden en que los driver son testeados es significante porque **DriverManager** usará el primer driver que encuentre que pueda conectar con éxito a la base de datos.
Primero DriverManager intenta usar cada driver en el orden en que ha sido registrado.
Testea los drivers mediante la llamada al método Driver.connect cada uno por turno, pasándole como argumento la URL que el usuario ha pasado originalmente al método DriverManager.getConnection. El primer driver que reconozca la URL realiza la conexión.

Statement
Un objeto Statement se usa para enviar sentencias SQL a la base de datos. Actualmente hay tres tipos de objetos Statement, todos los cuales actúan como contenedores para la ejecución de sentencias en una conexión dada: Statement, PreparedStatement que hereda de Statement y Callable Statement que hereda de PreparedStatement.

Ejecución de sentencias usando objetos Statement.

La interfaz Statement nos suministra tres métodos diferentes para ejecutar sentencias SQL, executeQuery, executeUpdate y execute.
El método a usar está determinado por el producto de la sentencia SQL

El método **executeQuery** está diseñado para sentencias que producen como resultado un único resultset tal como las sentencias SELECT.
El método **executeUpdate** se usa para ejecutar sentencias INSERT, UPDATE ó DELETE así como sentencias SQL DDL (Data Definition Language) como CREATE TABLE o DROP TABLE. El efecto de una sentencia INSERT, UPDATE o DELETE es una modificación de una o más columnas en cero o más filas de una tabla. El valor devuelto de executeUpdate es un entero que indica el número de filas que han sido afectadas (referido como update count). Para sentencias tales como CREATE TABLE o DROP TABLE, que no operan sobre filas, el valor devuelto por **executeUpdate** es siempre cero.
El método **execute** debería usarse solamente para ejecutar sentencias que devuelven más de un result set, más que un update count o una combinación de ambos.

Cerrar objetos Statement.
Los objetos Statement se cerrarán automáticamente por el colector de basura de Java (garbage collector). No obstante se recomienda como una buena práctica de programación que se cierren explicitamente cuando no sean ya necesarios.

ResultSet.
Un ResultSet contiene todas las filas que satisfacen las condiciones de una sentencia SQL y proporciona el acceso a los datos de estas filas mediante un conjunto de métodos get que permiten el acceso a las diferentes columnas de la filas.
El método **ResultSet.next** se usa para moverse a la siguiente fila del resultset, convirtiendo a ésta en la fila actual.
El formato general de un resultset es una tabla con cabeceras de columna y los valores correspondientes devueltos por la 'query'.

Filas y Cursores.
Un ResultSet mantiene un cursor que apunta a la fila actúal de datos. El cursor se mueve una fila hacia abajo cada vez que se llama al método **next**. Incialmente se sitúa antes de la primera fila, por lo que hay que llamar al método next para situarlo en la primera fila conviertiendola en la fila actúal. Las filas de ResultSet se recuperan en secuencia desde la fila más alta a la más baja.
Un cursor se mantiene válido hasta que el objeto Resultset o su objeto padre Statement se cierra.

Columnas.
Los métodos **getInt("nombrecolumna"), getChar("nombrecolumna"),** etc, suministran los medios para recuperar los valores de las columnas de la fila actúal. Dentro de cada fila, los

valores de las columnas pueden recuperarse en cualquier orden, pero para asegurar la máxima portabilidad, deberían extraerse las columnas de izquierda a derecha y leer los valores de las columnas una única vez.

Si no estamos seguros del tipo de la columna, podemos usar el método **getObject().**
Puede usarse o bien el nombre de la columna o el número de columna para referirse a esta.
Si, por otro lado, la sentencia select no especifica nombres de columnas (tal como en "select * from table1" o en casos donde una columna es derivada), deben usarse los números de columna. En estas situaciones , no hay forma de que el usuario sepa con seguridad cuales son los nombres de las columnas.

EJEMPLO:
si la 'query' es SELECT a, b FROM Table1, el resultado tendrá una forma semejante a :
a b
-------- ---------
12345 Cupertino
83472 Redmond
83492 Boston
El siguiente fragmento de código es un ejemplo de la ejecución de una sentencia SQL que devolverá una colección de filas, con la columna 1 como un int y la columna 2 como una String.

```
java.sql.Statement st = con.createStatement();
ResultSet rs = st.executeQuery("SELECT a, b FROM Table1");
while (rs.next())
{
int i = rs.getInt("a");
String s = rs.getString("b");
System.out.println("ROW = " + i + " " + s);
}
```

PreparedStatement.

La interfaz PreparedStatement hereda de Statement y difiere de esta en dos maneras.
- Las instancias de **PreparedStatement** contienen una sentencia SQL que ya ha sido compilada. Esto es lo que hace que se le llame 'preparada'.
- La sentencia SQL contenida en un objeto PreparedStatement pueden tener uno o más parámetros IN. Un parámetro IN es aquel cuyo valor no se especifica en la sentencia SQL cuando se crea. En vez de ello la sentencia tiene un interrogante ('?') como un 'encaje' para cada parámetro IN. Debe suministrarse un valor para cada interrogante mediante el método apropiado setInt, setString, etc, antes de ejecutar la

sentencia.
Como los objetos PreparedStatement están precompilados, su ejecución es más rápida que los objetos Statement. Consecuentemente, una sentencia SQL que se ejecute muchas veces a menudo se crea como PreparedStatement para incrementar su eficacia.
Siendo una subclase de Statement, PreparedStatement hereda toda la funcionalidad de Statement. Además , se añade un set completo de métodos necesarios para fijar los valores que van a ser enviados a la base de datos en el lugar de los 'encajes' para los parámetros IN.
También se modifican los tres métodos execute, executeQuery y executeUpdate de tal forma que no toman argumentos. Los formatos de Statement de estos métodos (los formatos que toman una sentencia SQL como argumento) no deberían usarse nunca con objetos PreparedStatement.

CallableStatement.

Un objeto CallableStatement provee de una forma estándar de llamar a procedimientos almacenados de la base de datos. Un procedimiento almacenado se encuentra en la base de datos. La llamada al procedimiento es lo que contiene el objeto CallableStatement. Esta llamada se escribe en una sintaxis de escape que puede tomar una de dos formas: una formato con un parámetro resultado y el otro sin el. (Ver la sección 4 para más información sobre la sintaxis de escape).
Un parámetro resultado, un tipo de parámetro OUT, es el valor devuelto por el procedimiento almacenado. Ambos formatos pueden tener un número variable de parámetros de entrada (parámetros IN), de salida (parámetros OUT) ó ambos (parámetros IN OUT). Un interrogante sirve como 'anclaje' para cada parámetro.
La sintaxis para invocar un procedimiento almacenado en JDBC se muestra a continuación: Notar que los corchetes indican que lo que se encuentra contenido en ellos es opcional, no forma parte de la sintaxis.
{call procedure_name[(?, ?, ...)]}
La sintaxis para un procedimiento que devuelve un resultado es:
{? = call procedure_name[(?, ?, ...)]}
La sintaxis para un procedimiento almacenado sin parámetros se parece a algo como:
{call procedure_name}
Normalmente, alguien que crea un objeto CallableStatement debería saber ya si la DBMS que está usando soporta o no procedimientos almacenados y que son estos. Si alguien necesita chequearlo de cualquier modo, existen varios métodos de DatabaseMetaData que suministran tal información.
Por ejemplo, el método supportsStoredProcedures devolverá true si la DBMS soporta llamadas a procedimientos almacenados y el método getProcedures devolverá una descripción de los procedimientos almacenados disponibles.
CallableStatement hereda los métodos de Statement, los cuales tratan sentencias SQL en

general, y también hereda los métodos de PreparedStatement, que tratan los parámetros IN.

Todos los métodos definidos para CallableStatement tratan los parámetros OUT o los aspectos de salida de los parámetros INOUT: registro de los tipos JDBC (tipos genéricos SQL) de los parámetros OUT, recuperación de valores desde ellos o chequear si el valor devuelto es un JDBC NULL.

_____

Tomado y adaptado de JDBC: UNA VISIÓN PRÁCTICA José Antonio Prieto Montalvo, Jesús Sánchez Varas

BIBLIOGRAFÍA.

Página web oficial de SUN MICROSYSTEMS. (http://www.sun.com)
Diversos manuales de JDBC encontrados en Internet.

Actividad Evaluativa

Realice Mapa Conceptual sobre el contenido expuesto.