Manual de Java

(JDBC)

1. INTRODUCCION.....4

1.1¿Qué es JDBC?.....4

1.1.1 ¿Qué hace JDBC?.....5

1.1.2 JDBC es un API de bajo nivel y una base para API's de alto nivel.....5

1.1.3 JDBC frente a ODBC y otros API's.....6

1.1.5 SQL Conforme.....8

1.2 Productos JDBC.....9

1.2.1 JavaSoft Framework.....10

1.2.2 Tipos de drivers JDBC.....11

2. CONEXIÓN.....12

2.1 Vista Preliminar.....12

2.1.1 Apertura de una conexión.....12

2.1.2 Uso general de URL's.....12

2.1.3 JDBC y URL's.....13

2.1.4 El subprotocolo "odbc".....15

2.1.5 Registro de subprotocolos.....15

2.1.6 Envío de Sentencias SQL.....16

2.1.7 Transacciones.....17

2.1.8 Niveles de aislamiento de transacciones.....18

3. LA CLASE DriverManager.....19

3.1 Vista preliminar.....19

3.1.1 Mantenimiento de la lista de drivers disponibles.....19

3.1.2 Establecer una conexión.....20

4. LA CLASE Statement.....22

4.1 Vista Preliminar.....22

4.1.1 Creación de objetos Statement.....22

4.1.2 Ejecución de sentencias usando objetos Statement.....23

4.1.3 Realización de Statement.....24

4.1.4 Cerrar objetos Statement.....24

4.1.5 Sintaxis de Escape SQL en objetos Statement.....24

4.1.6 Uso del método execute.....27

5. LA CLASE ResultSet.....30

5.1 Vista Preliminar.....30

5.1.1 Filas y Cursores.....30

5.1.2 Columnas.....31

5.1.3 Tipos de datos y conversiones.....32

Unicode.....35

Unicode es un esquema de codificación de caracteres que utiliza 2 bytes por cada carácter. ISO (International Standards Organization) define un número dentro del intervalo 0 a 65.535 (216 – 1) por cada carácter y símbolo de cada idioma (más algunos

espacios vacíos para futuras ampliaciones). En todas las versiones de 32 bits de Windows, el Modelo de objetos componentes (COM), que es la base de las tecnologías OLE y ActiveX, utiliza Unicode. Unicode es totalmente compatible con Windows NT. Aunque Unicode y DBCS tienen caracteres de doble byte, los esquemas de codificación son completamente diferentes. 35
5.1.5 Valores resultado NULL..... 36
5.1.6 Result sets opcionales o múltiples..... 36

6. LA CLASE PreparedStatement..... 37

6.1 Vista Preliminar..... 37
6.1.1 Creación de objetos PreparedStatement..... 37
6.1.2 Pasar parámetros IN..... 38
6.1.4 Usar setObject..... 39
6.1.5 Envío de JDBC NULL como un parámetro IN..... 40
6.1.6 Envio de parámetros IN muy grandes..... 40

7. LA CLASE CallableStatement..... 41

7.1 Vista Preliminar..... 41
7.1.1 Crear objetos CallableStatement..... 42
7.1.2 Parámetros IN y OUT..... 42
7.1.3 Parámetros INOUT..... 43
7.1.4 Recuperar parámetros OUT después de resultados..... 44
7.1.5 Recuperar valores NULL en parámetros OUT..... 44

9. EJEMPLO DE CODIGO..... 45

JDBC.

1. INTRODUCCION

Java Database Connectivity (JDBC) es una interfase de acceso a bases de datos estándar 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.

El paquete actual de JDK incluye JDBC y el puente JDBC-ODBC. Estos paquetes son para su uso con JDK 1.0

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, este driver puede estar escrito en Java puro, o ser una mezcla de Java y métodos nativos JNI (Java Native Interface).

1.1 ¿Qué es JDBC?

JDBC es el API para la ejecución de sentencias SQL. (Como punto de interés JDBC es una marca registrada y no un acrónimo, no obstante a menudo es conocido como “Java Database Connectivity”). Consiste en un conjunto de clases e interfases escritas en el lenguaje de programación Java. JDBC suministra un API estándar para los desarrolladores y hace posible escribir aplicaciones de base de datos usando un API puro Java.

Usando JDBC es fácil enviar sentencias SQL virtualmente a cualquier sistema de base de datos. En otras palabras, con el API JDBC, no es necesario escribir un programa que acceda a una base de datos Sybase, otro para acceder a Oracle y otro para acceder a Informix. Un único programa escrito usando el 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. Por ejemplo, con Java y JDBC API, es posible publicar una página web que contenga un applet que usa información obtenida de una base de datos remota. O una empresa puede usar JDBC para conectar a todos sus empleados (incluso si usan un conglomerado de máquinas Windows, Macintosh y UNIX) a una base de datos interna vía intranet. Con cada vez más y más programadores desarrollando en lenguaje Java, la necesidad de acceso fácil a base de datos desde Java continúa creciendo.

1.1.1 ¿Qué hace JDBC?

Simplemente JDBC hace posible estas tres cosas:

- Establece una conexión con la base de datos.
- Envía sentencias SQL
- Procesa los resultados.

El siguiente fragmento de código nos muestra un ejemplo básico de estas tres cosas:

```
Connection con = DriverManager.getConnection (
           "jdbc:odbc:wombat", "login", "password");
Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery("SELECT a, b, c FROM Table1");
while (rs.next()) {
  int x = rs.getInt("a");
  String s = rs.getString("b");
  float f = rs.getFloat("c");
  }
```

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

JDBC es una interfase 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 otros 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 interfase de alto nivel es 'amigable', usa un API mas entendible o más conveniente que luego se traduce en la interfase de bajo nivel tal como JDBC.

1.1.3 JDBC frente a ODBC y otros API's

En este punto, el ODBC de Microsoft (Open Database Connectvity), es probablemente el API más extendido para el acceso a bases de datos relacionales. Ofrece la posibilidad de conectar a la mayoría de las bases de datos en casi todas las plataformas. ¿Por qué no usar, entonces, ODBC, desde Java?.

La respuesta es que se puede usar ODBC desde Java, pero es preferible hacerlo con la ayuda de JDBC mediante el puente JDBC-ODBC. La pregunta es ahora ¿por qué necesito JDBC?. Hay varias respuestas a estas preguntas:

1.- ODBC no es apropiado para su uso directo con Java porque usa una interface C. Las llamadas desde Java a código nativo C tienen un número de inconvenientes en la seguridad, la implementación, la robustez y en la portabilidad automática de las aplicaciones.

2.- Una traducción literal del API C de ODBC en el API Java podría no ser deseable. Por ejemplo, Java no tiene punteros, y ODBC hace un uso copioso de ellos, incluyendo el notoriamente propenso a errores "void * ". Se puede pensar en JDBC como un ODBC traducido a una interfase orientada a objeto que es el natural para programadores Java.

3. ODBC es difícil de aprender. Mezcla características simples y avanzadas juntas, y sus opciones son complejas para 'querys' simples. JDBC por otro lado, ha sido diseñado para mantener las cosas sencillas mientras que permite las características avanzadas cuando éstas son necesarias.

4. Un API Java como JDBC es necesario en orden a permitir una solución Java "pura". Cuando se usa ODBC, el gestor de drivers de ODBC y los drivers deben instalarse manualmente en cada máquina cliente. Como el driver JDBC esta completamente escrito en Java, el código JDBC es automáticamente instalable, portable y seguro en todas las plataformas Java.

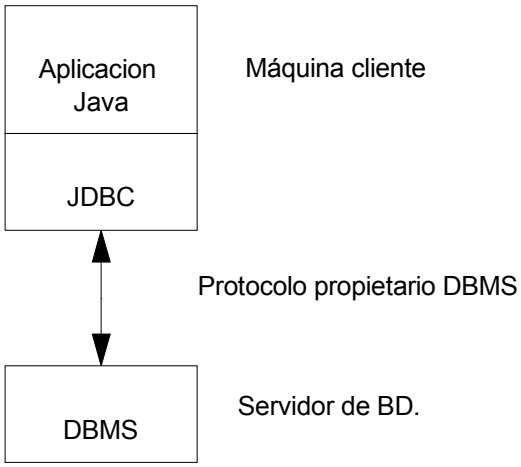
En resumen, el API JDBC es el interfase natural de Java para las abstracciones y conceptos básicos de SQL. JDBC retiene las características básicas de diseño de ODBC; de hecho, ambos interfaces están basados en el X/Open SQL CLI (Call Level Interface).

Más recientemente Microsoft ha introducido nuevas API detrás de ODBC. RDO, ADO y OLE DB. Estos diseños se mueven en la misma dirección que JDBC en muchas maneras, puesto que se les da una orientación a objeto basándose en clases que se implementan sobre ODBC.

1.1.4 Modelos en dos y tres pisos.

El API JDBC soporta los modelos en dos y tres pisos de acceso a base de datos.

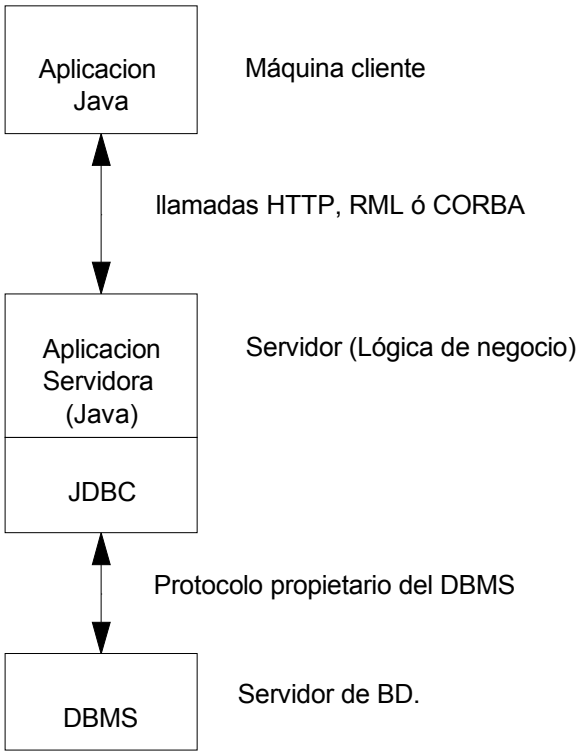
En el modelo de dos-pisos, un applet Java o una aplicación habla directamente con la base de datos. Esto requiere un driver JDBC que pueda comunicar con el gestor de base de datos particular al que se pretende acceder. Las sentencias SQL de usuario se envían a la base de datos, y el resultado de estas sentencias se envían al usuario. La base de datos puede estar localizada en otra máquina a la que el usuario se conecta mediante la red. Esta es una configuración *Cliente/Servidor* en la que la máquina del usuario es el cliente y la máquina que hospeda a la base de datos es el servidor. La red puede ser una intranet, por ejemplo, que conecta a los empleados dentro de la corporación, o puede ser Internet.



En el modelo de tres-pisos, los comandos se envían a un 'piso intermedio' de servicios, que envía las sentencias SQL a la base de datos. La base de datos procesa las sentencias SQL y devuelve los resultados a el 'piso intermedio', que a su vez lo envía al usuario. Los directores de IS encuentran este modelo muy atractivo por que el 'piso intermedio' hace posible mantener el control sobre los datos y los tipos de actualizaciones que pueden hacerse en los datos corporativos. Otra ventaja es que el usuario puede emplear un API de alto nivel más sencillo que es traducido por el 'piso intermedio' en las llamadas de bajo nivel apropiadas. Finalmente en muchos casos la arquitectura de tres niveles puede proporcionar ventajas de rendimiento.

Hasta ahora, este nivel intermedio ha sido escrito en lenguajes como C ó C++, que ofrecen un rendimiento más rápido. De cualquier modo, con la introducción de compiladores optimizadores que traducen el bytecode en código máquina eficiente, se está convirtiendo en práctico desarrollar este nivel intermedio en Java.

Esta es una gran ventaja al hacer posible aprovechar las características de robustez, multiproceso y seguridad de Java.



1.1.5 SQL Conforme

SQL es el lenguaje estándar para el acceso a las bases de datos relacionales. Una de las áreas de dificultad es que aunque muchas DBMS's (Data Base Management Systems) usan un formato estándar de SQL para la funcionalidad básica, estos no conforman la sintaxis más recientemente definidas o semánticas para funcionalidades más avanzadas. Por ejemplo, no todas las bases de datos soportan procedimientos almacenados o joins de salida, y aquellas que lo hacen no son consistentes con otras. Es de esperar que la porción de SQL que es verdaderamente estándar se expandirá para incluir más y más funcionalidad. Entretanto, de cualquier modo, el API de JDBC debe soportar SQL tal como es.

Una forma en que el API JDBC trata este problema es permitir que cualquier cadena de búsqueda se pase al driver subyacente del DBMS. Esto quiere decir que una aplicación es libre de usar la sentencia SQL tal como quiera, pero se corre el riesgo de recibir un error en el DBMS. De hecho una consulta de una aplicación incluso no tiene por que ser SQL, o puede ser una derivación especializada de SQL

diseñada para especificas DBMS (para consultas a imágenes o documentos por ejemplo).

Una segunda forma en que JDBC trata este problema es proveer cláusulas de escape al estilo de ODBC , que se discutirán en el 4.1.5. ”Sintaxis de escape en Sentencias Objetos”.

La sintaxis de escape provee una sintaxis JDBC estándar para varias de las áreas más comunes de divergencia SQL. Por ejemplo, ahí escapes para literales de fecha o procedimientos almacenados.

Para aplicaciones complejas, JDBC trata la conformidad SQL de una tercera manera. Y es proveer información descriptiva sobre el DBMS por medio de la interfase `DatabaseMetaData` por la que las aplicaciones pueden adaptarse a los requerimientos y posibilidades de cada DBMS:

Como el API JDBC se usará como base para el desarrollo de herramientas de acceso y API’s de alto nivel , direccionará el problema de la conformidad a cualquiera de estas. La designación “JDBC COMPLIANT” se creó para situar un nivel estándar de funcionalidad JDBC en la que los usuarios puedan confiar. En orden a usar esta designación, un driver debe soportar al menos el nivel de entrada ANSI SQL-2 Entry Level. Los desarrolladores de drivers pueden cerciorarse que sus drivers cumplan estas especificaciones mediante la suite de test disponible en el API JDBC.

La designación “JDBC COMPLIANT” indica que la implementación JDBC de un vendedor ha pasado los tests de conformidad suministrados por JavaSoft. Estas pruebas de conformidad chequean la existencia de todas las clases y métodos definidos en el API JDBC, y chequean tanto como es posible que la funcionalidad SQL Entry Level esté disponible. Tales tests no son exhaustivos, por supuesto, y JavaSoft no esta distribuyendo implementaciones de vendedores, pero esta definición de compliance tiene algún grado de seguridad en una implementación JDBC. Con la mayor aceptación de JDBC por parte de vendedores de bases de datos, de servicios de Internet y desarrolladores, JDBC se está convirtiendo en el estándar de acceso a bases de datos.

1.2 Productos JDBC.

Existen una serie de productos basados en JDBC que ya han sido desarrollados. Por supuesto la información de este apartado será rápidamente obsoleta.

```
http://java.sun.com/products/jdbc
```

1.2.1 JavaSoft Framework

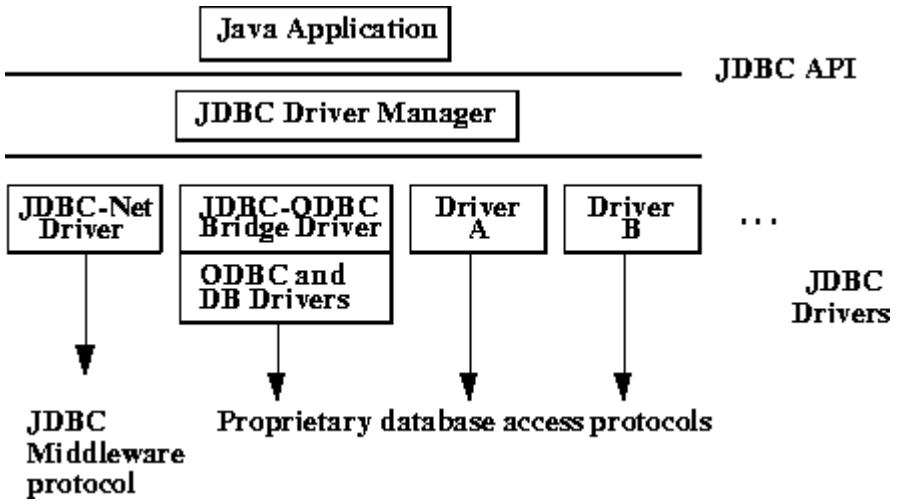
JavaSoft suministra tres componentes JDBC como parte del JDK

- El gestor de drivers JDBC
- La suite de testeo de drivers JDBC
- El puente JDBC-ODBC

El gestor de drivers es la espina dorsal de la arquitectura JDBC. Actualmente es bastante pequeña y simple; su función principal es conectar las aplicaciones Java al driver JDBC correcto y después soltarlo.

La suite de testeo JDBXC suministra seguridad y confianza en los drivers JDBC que se ejecutarán en el programa. Solo los drivers que pasan el test pueden ser designados JDBC COMPLIANT.

El puente JDBC-ODBC permite a los drivers ODBC usarse como drivers JDBC. Fue implementado como una forma de llegar rápidamente al fondo de JDBC y para proveer de acceso a los DBMS menos populares si no existen drivers JDBC para ellos.



1.2.2 Tipos de drivers JDBC

Los drivers que son susceptibles de clasificarse en una de estas cuatro categorías.

1.- *puente JDBC-ODBC más driver ODBC*: El producto de JavaSoft 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.

2.- driver Java parcialmente Nativo. Este tipo de driver convierte llamadas JDBC en llamadas del API cliente para Oracle, 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.

Esperamos que las alternativas 3 y 4 sean las formas preferidas de acceder a las bases de datos desde JDBC. Las categorías 1 y 2 son soluciones interinas cuando no están disponibles drivers directos puros Java.

2. CONEXIÓN

2.1 Vista Preliminar

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.

2.1.1 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 D`river` 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.

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");
```

2.1.2 Uso general de URL's

Dado que URL's causan a menudo cierta confusión, daremos primero una breve explicación de URL en general y luego entraremos en una discusión sobre URL's de JDBC.

Una URL (Uniform Resource Locator) da información para localizar un recurso en Internet. Puede pensarse en ella como una dirección.
La primera parte de una URL especifica el protocolo usado para acceder a la información y va siempre seguida por dos puntos. Algunos protocolos comunes son `ftp`, que especifica "file transfer protocol" y `http` que especifica "hypertext transfer protocol". Si el protocolo es "`file`" indica que el recurso está en un sistema de ficheros local mejor que en Internet : veamos unos ejemplos:

```
ftp://javasoft.com/docs/JDK-1_apidocs.zip
http://java.sun.com/products/jdk/CurrentRelease
file:/home/haroldw/docs/books/tutorial/summary.html
```

El resto de la URL, todo después de los dos puntos, da información sobre donde se encuentra la fuente de los datos. Si el protocolo es file, el resto de la URL es el path al fichero. Para los protocolos `ftp` y `http`, el resto de la URL identifica el host y puede opcionalmente dar un path más específico al sitio. Por ejemplo, el siguiente es la URL para la home page de JavaSoft. Esta URL identifica solo al host:

```
http://java.sun.com
```

2.1.3 JDBC y URL's

Una URL JDBC suministra una forma de identificar una base de datos para que el driver apropiado pueda reconocerla y establecer la conexión con ella. Los desarrolladores de drivers son los que determinan actualmente que JDBC URL identifica su driver particular. Los usuarios no tienen por que preocuparse sobre como se forma una URL JDBC; ellos simplemente usan la URL suministrada con el driver que usan. El rol de JDBC es recomendar algunas convenciones a los fabricantes de drivers para su uso.

Dado que los JDBC URL se usan con varios tipos de drivers, las convenciones son necesariamente muy flexibles. Primero, permiten a diferentes drivers usar diferentes esquemas para nombrar las bases de datos. EL subprotocolo `odbc`, por ejemplo, permite que las URL contengan valores de atributos (pero no los requieren).

Segundo, las URL's JDBC permiten a los desarrolladores de drivers codificar toda la información de la conexión dentro de ella. Esto hace posible, por ejemplo, para un applet que quiera hablar con una base de datos dada el abrir la conexión sin necesitar que el usuario realice ninguna tarea de administración de sistemas.

Tercero, las URL's JDBC permiten un nivel de indirección. Esto quiere decir que la URL JDBC puede referirse a una base de datos lógica o un host lógico que se traduce dinámicamente al nombre actual por el sistema de nombramiento de la red.

Esto permite a los administradores de sistemas evitar dar especificaciones de sus hosts como parte del nombre JDBC. Hay una variedad de servicios de nomenclatura de red diferentes (tales como DNS, NIS y DCE), y no hay restricción acerca de cual usar.
La sintaxis para las URL's JDBC que se muestra a continuación tiene tres partes separadas por dos puntos:

jdbc:<subprotocol>:<subname>

Las tres partes se descomponen como sigue:

1 **jdbc** – el protocolo. El protocolo en una URL JDBC es siempre `jdbc`

2 - `<subprotocol>` - el nombre del driver o el nombre del mecanismo de conectividad con la base de datos, que puede estar soportado por uno o más drivers. Un ejemplo sobresaliente de un subprotocolo es "odbc", que ha sido reservado para URL's que especifican nombres de fuentes de datos estilo ODBC. Por ejemplo para acceder a la base de datos a través del puente JDBC-ODBC, la URL a usar podría ser algo así como lo siguiente:

```
jdbc:odbc:fred
```

En este ejemplo, el subprotocolo es "odbc" y el subnombre "fred" es el nombre de la fuente de datos ODBC local.

Si se quiere usar un servicio de nombre de la red (ya que el nombre de la base de datos en la URL JDBC no tiene por que ser su nombre actúal), el servicio de nombre puede ser el subprotocolo. Por tanto, para el ejemplo, podría tener una URL como :

```
jdbc:dcenaming:accounts-payable
```

En este ejemplo, la URL especifica que el servicio local DCE resolverá el nombre de la base de datos "accounts-payable" para poder conectar con la base de datos real.

3 **<subname>** - una forma de identificar la base de datos. El subnombre puede variar dependiendo del subprotocolo, y puede tener un subnombre con cualquier sintaxis interna que el fabricante del driver haya escogido. El punto de un subnombre es para dar información suficiente para localizar la base de datos. En el ejemplo anterior "fred" es suficiente porque ODBC suministra la información restante. Una base de datos en un servidor remoto requiere más información. Si la base de datos va a ser accesible a través de Internet, por ejemplo, la dirección de red debería incluirse en la URL JDBC como parte del

subnombre y debería seguir a la convención estándar de nomenclatura de URL.

//hostname:port/subsubname

Suponiendo que “dbnet” es un protocolo para conectar a un host, la URL JDBC debería parecerse a algo como:

```
jdbc:dbnet://wombat:356/fred
```

2.1.4 El subprotocolo “odbc”

El subprotocolo “odbc” es un caso especial. Ha sido reservado para URL’s que especifican el estilo ODBC de nombres de fuentes de datos y que tiene la característica de permitir especificar cualquier número de valores de atributos después del subnombre (el nombre de la fuente de datos) La sintaxis completa para el protocolo “odbc” es:

```
jdbc:odbc:<data-source-name>[;<attribute-name>=<attribute-value>]*
```

Todos los siguientes son nombres válidos jdbc:odbc

```
jdbc:odbc:qeor7
jdbc:odbc:wombat
jdbc:odbc:wombat;CacheSize=20;ExtensionCase=LOWER
jdbc:odbc:qeora;UID=kgh;PWD=fooey
```

2.1.5 Registro de subprotocolos

Un desarrollador de drivers puede reservar un nombre para usar como el subprotocolo en una URL JDBC. Cuando la clase `DriverManager` presenta este nombre a su lista de drivers registrados, el driver para el que este nombre está reservado debería reconocerlo y establecer una conexión a la base de datos que lo identifica. Por ejemplo `odbc` está reservado para el puente JDBC-ODBC. Si fuera, por poner otro ejemplo, Miracle Corporation, y quisiera registrar “miracle” como el subprotocolo para el driver JDBC que conecte a su DBMS Miracle no tiene que usar sino ese nombre.

2.1.6 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. Por ejemplo, una aplicación que intenta enviar una llamada a un procedimiento almacenado a una DBMS que no soporta procedimientos almacenados no tendrá éxito y generará una excepción. JDBC requiere que un driver cumpla al menos ANSI SQL-2 Entry Level para ser designado JDBC COMPLIANT. Esto significa que los usuarios pueden contar al menos con este nivel de funcionalidad.

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:

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

2.- **`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.

3.- **`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.

La lista siguiente da una forma rápida de determinar que método Connection es el apropiado para crear los diferentes tipos de sentencias SQL.

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.

2.1.7 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. De hecho, un driver JDBC-compliant debe soportar transacciones. DatabaseMetaData suministra información que describe el nivel de transacción soportado por el DBMS.

2.1.8 Niveles de aislamiento de transacciones

Si un DBMS soporta el proceso de transacciones, tendrá que manejar de alguna forma los potenciales conflictos que puedan surgir cuando dos transacciones están operando sobre una base de datos concurrentemente. El usuario puede especificar un nivel de aislamiento para indicar que nivel de precaución debería ejercitar el DBMS para la resolución de estos conflictos. Por ejemplo, ¿que ocurrirá cuando una transacción cambia un valor y una segunda transacción lee el valor antes de que el cambio haya sido 'commitado' o descartado?. ¿Debería permitirse, dado que el valor cambiado leído por la segunda transacción será invalido si la primera transacción ha hecho rollback?. Un usuario JDBC puede instruir a la DBMS para que un valor que ha sido leído antes del 'commit' ("dirty reads") con el siguiente código donde `con` es el objeto de la actual conexión:

```
con.setTransactionIsolation(TRANSACTION_READ_UNCOMMITTED);
```

El nivel de aislamiento más alto, el que más cuidado toma para evitar conflictos. La interfase `Connection` define cinco niveles de aislamiento con el nivel más bajo que especifica que no soporte transacciones hasta el más alto que especifica que mientras una transacción esté abierta ninguna otra transacción puede realizar cambios en los datos leídos por esa transacción. Normalmente, el nivel de transacción más alto es el más lento en la ejecución de la aplicación.(debido a que se incrementan los bloqueos y se disminuye la concurrencia de los usuarios). El desarrollador debe balancear la necesidad de rendimiento con la necesidad de la consistencia de los datos al tomar la decisión del nivel de aislamiento a usar. Por supuesto, el nivel de aislamiento que pueda soportarse depende de las posibilidades de la base de datos subyacente.

Cuando se crea un nuevo objeto `Connection`, su nivel de aislamiento depende del driver, pero normalmente por defecto es el de la DBMS subyacente. Un usuario puede llamar al método `setIsolationLevel` para cambiar el nivel de aislamiento de la transacción, y este nuevo nivel estará efectivo durante la sesión de conexión. Para cambiar el nivel de aislamiento solo para una transacción, es necesario fijar este antes de la transacción comience y volverlo a situar en su valor anterior una vez que la transacción haya terminado. No se recomienda cambiar el nivel de aislamiento de transacción en medio de una puesto que lanzará una llamada inmediata al método `commit`, provocando que los cambios hasta ese punto se hagan permanentes en la base de datos.

3. LA CLASE DriverManager

3.1 Vista preliminar

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.

3.1.1 Mantenimiento de la lista de drivers disponibles.

La clase `DriverManager` mantiene una lista de clases disponibles que han sido registrados mediante el método `DriverManager.registerDriver`. Todas las clases `Driver` deben escribirse con una sección estática que cree una instancia de la clase y luego la registre en la clase `DriverManager` cuando se cargue. Además el usuario normalmente no debería llamar a `DriverManager.registerDriver` directamente; debería llamarse automáticamente por el driver cuando esté se carga,. Una clase `Driver` se carga, y por tanto se registra, de dos formas diferentes:

1 Mediante una llamada al método **Class.forName**. Este carga explícitamente la clase driver. Dado que no depende de ningún 'setup' externo, esta forma de cargar drivers es la recomendada. El siguiente código carga la clase `acme.db.Driver`:

```
Class.forName("acme.db.Driver");
```

Si `acme.db.Driver` se ha escrito para que al cargar produzca una instancia y llame al método `DriverManager.registerDriver` con esta instancia como argumento (es lo que debería hacer), entonces este estará en la lista de drivers disponibles para efectuar la conexión.

2 Mediante la adición del driver a la propiedad jdbc.drivers de java.lang.System- Esta es una lista de nombres de clases de drivers, separadas por dos puntos, que es la que carga la clase DriverManager. Cuando la clase DriverManager se inicializa, mira en la propiedad jdbc.drivers, y si el usuario ha introducido uno o más drivers, la clase DriverManager intenta cargarlos. El siguiente código ilutsra como un programador debería introducir estas tres clases en `~/.hotjava/properties`

```
jdbc.drivers=foo.bah.Driver:wombat.sql.Driver:bad.test.ourDriver;
```

La primera llamada a el método `DriverManager` hará que estos drivers se carguen automáticamente.

Notese que en esta segunda manera de cargar los drivers es necesario una preparación del entorno que es persistente. Si existe alguna duda sobre esto es preferible y más seguro llamar al método `Class.forName` para cargar explicitamente cada driver. Este es también el método que se usa para traer un driver particular puesto que una vez que la clase `DriverManager` ha sido inicializada no chequeará la lista de propiedades jdbc.drivers.

En ambos casos, es responsabilidad de la clase `Driver` recién cargada registrarse a si misma mediante una llamada `a DriverManager.registerDriver`. Como se ha mencionado anteriormente, esto debería hacerse automáticamente al cargar la clase.

Por razones de seguridad, la capa de gestión de JDBC guardará traza de que clases de cargadores provee cada driver. Entonces cuando la clase DriverManager abre una conexión solo usará los drivers que vienen desde el sistema de ficheros local o desde las mismas clases cargadoras como el código que solicita la conexión.

3.1.2 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 regsitrados 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. Por ejemplo, cuando conectamos con una base de datos remota, podría ser posible usar un driver puente JDBC-ODBC, o un driver JDBC de protocolo genérico de red, o un driver suministrado por el vendedor.

En tales casos, 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 (los drivers listados en la propiedad jdbc.drivers son siempre los registrados primero). Saltará cualquier driver con código 'untrusted', a menos que se cargue desde el mismo código fuente que el código que intenta abrir la conexión.

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.

Una primera ojeda puede parecer insuficiente, pero son necesarias solo unas pocas llamadas a procedimientos y comparaciones de cadena por conexión puesto que no es probable que haya docenas de drivers se carguen concurrentemente.

El siguiente código es un ejemplo de todo lo necesario normalmente para establecer una conexión con un driver puente JDBC-ODBC:

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");  //loads the driver
String url = "jdbc:odbc:fred";
DriverManager.getConnection(url, "userID", "passwd");
```

4. LA CLASE Statement

4.1 Vista Preliminar

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 `CallableStatement` que hereda de `PreparedStatement`. Estas estàn especializadas para enviar tipos particulares de sentencias SQL, Un objeto `Statement` se usa para ejecutar una sentencia SQL simple sin parámetros. Un objeto `PreparedStatement` se usa para ejecutar sentencias SQL precompiladas con o sin parámetros IN; y un objeto `CallableStatement` se usa para ejecutar un procedimieno de base de datos almacenado.

La interfase `Statement` suminstra métodos básicos para ejecutar sentencias y devolver resultados. La interfase `PreparedStatement` añade métodos para trabajat con los parámetros IN; y la interfase `CallableStatement` añade métodos para trabajar con parameters OUT.

4.1.1 Creación de objetos Statement

Una vez establecida la conexión con una base de datos particular, esta conexión puede usarse para enviar sentencias SQL. Un objeto `Statement` se crea mediante el método de `Connection createStatement`, como podemos ver en el siguiente fragmento de código.

```
Connection con = DriverManager.getConnection(url, "sunny", "");
Statement stmt = con.createStatement();
```

La sentencia SQL que será enviada a la base de datos es alimentada como un argumento a uno de los métodos de ejecución del objeto `Statement`. Por ejemplo:

```
ResultSet rs = stmt.executeQuery("SELECT a, b, c FROM Table2");
```

4.1.2 Ejecución de sentencias usando objetos Statement.

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

El método `executeQuery` esta diseñado para sentencias que producen como resultado un único result set 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, le valor devuelto por `executeUpdate` es siempre cero.

El método `execute` se usa para ejecutar sentencias que devuelven más de un result set, más que un update count o una combinación de ambos. Como es esta una característica avanzada que muchos programadores no necesitarñan nunca se verá en su propia sección.

Todos los métodos que ejecutan sentencias cierran los objetos `Resultset` abiertos como resultado de las llamadas a `Statement`. Esto quiere decir que es necesario completar el proceso con el actual objeto `Resulset` antes de reejecutar una sentencia `Statement`.

Debe notarse que la interfase `PreparedStatement`, que hereda los métodos de la interfase `Statement`, tiene sus propias versiones de los métodos `executeQuery`, `executeUpdate` y `execute`. Los objetos Statement en si mismos no contienen una sentencia SQL, por tanto debe suministrarse como un argumento a los métodos Statement.`execute`. Los objetos `PreparedStatement` no suministran una sentencia SQL como argumento a estos métodos puesto que ya tienen la sentencia precompilada. Los objetos `CallableStatement` heredan las formas de estos métodos de `PreparedStatement`. Usar un parametro de query con las versiones de los métodos de `PreparedStatement` o `CallableStatement` producirái una `SQLException`,.

4.1.3 Realización de Statement

Cuando una conexión está en modo auto-commit, las sentencias ejecutadas son 'comitadas' o rechazadas cuando se completan. Un sentencia se considera completa cuando ha sido ejecutada y se han devuelto todos los resultados. Pare el método `executeQuery`, que devuelve un único result set, la sentencia se completa cuando todas las filas del objeto `ResultSet` se han devuelto. Para el método `executeUpdate`, un sentencia se completa cuando se ejecuta. En los raros casos en que se llama al método `execute`, de cualquier modo, no se completa hasta que los result sets o update counts que se generan han sido devueltos.

Algunos DBMS tratan cada sentencia en un procedimiento almacenado como sentencias separadas. Otros tratan el procedimiento entero como una sentencia compuesta. Esta diferencia se convierte en importante cuando está activo el modo auto-commit porque afecta cuando se llama al método commit. En el primer caso, cada sentencia individual es commitada. En el segundo, se commiten todas juntas.

4.1.4 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. Esto libera recursos DBMS inmediatamente y ayuda a evitar potenciales problemas de memoria.

4.1.5 Sintaxis de Escape SQL en objetos Statement

Los objetos Statement pueden contener sentencias SQL que usen sintaxis de escape SQL. La sintaxis de escape señala al driver que el código que lleva debe ser tratado diferentemente. El driver buscará por cualquier sintaxis de escape y lo traducirá en código que entiende la base de datos en particular. Esto hace que la sintaxis de escape sea independiente de la DBMS y permite al programador usar características que de otro modo no estarían disponibles.

Una clausula de escape se enmarca entre llaves y tiene una palabra clave:

```
{keyword . . . parameters . . . }
```

La palabra clave (keyword) indica el tipo de clausula de escape, según se muestra:

- `escape` para caracteres LIKE

  Los caracteres “%” y “_” trabajan como wildcards en la clausula SQL LIKE (“%” significa cero o más caracteres y “_” significa exactamente un carácter”. En orden a interpretarlos literalmente, pueden estar precedidos por un backslash (‘\’), que es un carácter de escape especial en cadenas. Se puede especificar un carácter que se use como carácter de escape por la inclusión de la sintaxis siguiente al final de la consulta.

  ```
  {escape 'escape-character'}
  ```

  Por ejemplo, la siguiente query, usando backslash como caracter de escape, encuentra nombres de identificador que comiencen con ‘_’.

  ```
  stmt.executeQuery("SELECT name FROM Identifiers
  WHERE Id LIKE `\_%' {escape `\'};
  ```

- `fn` para funciones escalares

  Casi todas las DBMS tienen funciones numèricas, de cadena, de fecha y conversión sobre valores escalares. Una de estas funciones puede usarse colocándola en la sintaxis de escape con la clave `fn` seguida del nombre de la función deseada y sus argumentos. Por ejemplo, para llamar a la función concat con dos argumentos que serán concatenados:

  ```
  {fn concat("Hot", "Java")};
  ```

  El nombre del usuario actual de la base de datos puede obtenerse mediante:

  ```
  {fn user()};
  ```

  Las funciones escalares pueden estar soportadas por diferentes DBMS con ligeras diferencias de sintaxis, y pueden no estar disponibles en todos los drivers. Varios métodos de `DatabaseMetaData` nos listarán las funciones que están soportadas. Por ejemplo, el método getNumericFunctions devuelve una lista de los nombres de las funciones numéricas separadas por comas, el método getStringFunction nos devuelve los nombres de las funciones de cadena, y así varías más.

  EL driver o bien mapeará la llamada a la función ‘escapada’ en su propia sintaxis o implementará la función el mismo.

- `d, t` y `ts` para literales de fecha y tiempo

  Las DBMS difieren en la sintaxis que usan para los literales de fecha, tiempo y timestamp. JDBC soporta un formato estándar ISO para estos literales y usa una clausula de escape que el driver debe traducir a la representación del DBMS.

  Por ejemplo, una fecha se especifica en SQL JDBC mediante la sintaxis:

  ```
  {d `yyyy-mm-dd'}
  ```

  En esta sintaxis, `yyyy` es el año, `mm` es el mes y `dd` es el dia. El driver reemplazará la clausula de escape por la representaciòn propia equivalente de la DBMS. Por ejemplo, el driver reemplazaría `{d 1999-02-28}` por `'28-FEB-99'` si este es el formato apropiado para la base subyacente.

  Hay clausulas de escape análogas para `TIME y TIMESTAMP`

  ```
  {t `hh:mm:ss'}
  {ts `yyyy-mm-dd hh:mm:ss.f . . .'}
  ```

  La parte fraccional de los segundos `(.f . . .)` del `TIMESTAMP` puede omitirse.

- `call` ó `? = call` para procedimientos almacenados

  Si una database soporta procedimientos almacenados, estos pueden ser invocados desde JDBC mediante:

  {call procedure_name[(?, ?, . . .)]}

  o, cuando el procedimiento devuelve como resultado un parámetro

  ```
  {? = call procedure_name[(?, ?, . . .)]}
  ```

  Los corchetes indican que el material encerrado en ellos es opcional. Estos no forman parte de la sintaxis.

  Los argumentos de entrada pueden ser bien literales, bien parámetros. Ver la sección 7 “CallableStatement” de esta guía.

  Se puede llamar al método `DatabaseMetaData.supportsStoredProcedures` para ver si la base de datos soporta procedimientos almacenados.

- `oj` para joins de salida

  La sintaxis para un outer join es:

  ```
  {oj outer-join}
  ```

  donde outer-join es de la forma:

  ```
  table LEFT OUTER JOIN {table | outer-join} ON search-condition
  ```

  Las Outer joins son una característica avanzada, y solo puede chequearse la gramática SQL mediente una explicación de ella. JDBC provee tres métodos de DatabaseMetaData para determinar que tipos de outer joins soporta un driver: `supportsOuterJoins, supportsFullOuterJoins, y supportsLimitedOuterJoins`.

El método `Statement.setEscapeProcessing` activa o desactiva el procesamiento de escape. Por defecto la característica se encuentra activada. Un programador debería desactivar esta característica en tiempo de ejecución cuando el rendimiento ha de ser máximo, pero normalmente debe estar activado. Debería notarse que `setEscapeProcesing` no trabaja con objetos `PreparedStatement` por que la sentencia ya está preparada para enviar a la base de datos antes de poder ser llamada.

4.1.6 Uso del método execute

El método execute debería usarse solamente cuando es posible que una sentencia nos devuelva más de un objeto `Resultset`., mas de un update count o una combinación de ambos. Estas múltiples posibilidades para resultados, aunque raras, son posibles cuando se ejecutan ciertos procedimientos almacenados o por la ejecución dinámica de una string SQL desconocida (esto es, desconocida para el programador de la aplicación en tiempo de compilación). Por ejemplo, una usuario podría ejecutar un procedimiento almacenado (usando una objeto `CallableStatement` y este procedimiento podría ejecutar una actualización, después una select, luego una actualización, después una select y así. Normalmente, alguien que usa un procedimiento almacenado sabrá que se le va a devolver.

Porque el método execute maneja los casos que se salen de lo ordinario, no sorprende que los resultados devueltos requieren algun manejo especial. Por ejemplo, supongamos que se sabe que el procedimiento devuelve dos result sets.

Después de usar el método `execute` para ejecutar el procedimiento, se debe llamar al método `getResultSet` para conseguir el primer result set y después los métodos apropiados `getXXX` para recuperar los valores de él. Para conseguir el segundo result set, se necesita llamar al método `getMoreResults` y y despues a `getResultSet` de nuevo. Si se sabe que el procedimiento devuelve dos upadte counts, se llama primero al método `getUpdateCount`, seguido de `getMoreResults` y de nuevo `getUpdateCount`.

Aquellos casos en los que no se conoce que devolverá se nos presenta una situación más compleja. El método execute devuelve `true` si el resultado es un objeto `ResultSet` y `false` si es un `int` Java. Si devuelve un `int`, esto quiere decir que el resultado o bien es un update count o que la sentencia que ha ejecutado es un comando DDL. Lo primero que hay que hacer después de llamar execute es llmar o bien a `getResultSet` o `getUpdateCount`. Al método `getResultSet` se le llama para conseguir el primero de los dos o más objetos `ResultSet` y al método `getUpdateCount` para conseguir el primero de dos o más update counts.

Cuando el resultado de una sentencia SQL no es un result set, el método `getResultSet` devolverá `null`. Esto quiere decir que el resultado es un update count o que no hay más resultados. La única manera de encontrar que significa el valor `null` en este caso es llamar al método `getUpdateCount`, que devolverá un entero. Este entero será el número de filas afectadas por la sentencia ejecutada o –1 para indicar o bien que el resultado es un result set o bien que no hay más resultados. Si el método `getResultSet` ya ha devuelto null, el resultado no puede ser un objeto `ResultSet`, por lo que el valor devuelto de –1 tiene que ser que no hay más resultados. En otras palabras, no hay más resultados cuando lo siguiente es cierto:

```
((stmt.getResultSet() == null) && (stmt.getUpdateCount() == -1))
```

Si se ha llamado al método `getResultSet` y se ha procesado el objeto `ResultSet` devuelto, es necesario llamar al método `getMoreResults` para ver si hay más result sets o update counts.. Si `getMoreResults` devuelve true, entonces es necesario llamar de nuevo a `getResultSet` para recuperar el siguiente result set. Como ya se ha indicado anteriormente, si `getResultset` devuelve `null` hay que llamar a `GetUpdateCount` para buscar que significa ese `null` si un update count o que no hay más resultados.

Cuando getMoreResults devuelve false quiere decir que la sentencia SQL ha devuelto un update count o que no hay más resultados. Por tanto es necesario llamar al método getUpdateCount para encontrar cual es el caso. En esta situación, no habrá más resultados cuando lo siguiente es cierto:

((stmt.getMoreResults() == false) && (stmt.getUpdateCount() == -1))

El siguiente código muestra una forma de asegurarse que se ha accedido a todos los result sets y update counts de una llamada al método execute:

```
stmt.execute(queryStringWithUnknownResults);
while (true)  {
  int rowCount = stmt.getUpdateCount();
  if (rowCount > 0) {    //  this is an update count
    System.out.println("Rows changed = " + count);
    stmt.getMoreResults();
    continue;
  }
  if (rowCount == 0) {    // DDL command or 0 updates
    System.out.println(" No rows changed or statement was DDL
                                command");
    stmt.getMoreResults();
    continue;
  }

// if we have gotten this far, we have either a result set
// or no more results

  ResultSet rs = stmt.getResultSet;
  if (rs != null) {
    . . .  // use metadata to get info about result set columns
    while (rs.next())  {
    . . .  // process results
    stmt.getMoreResults();
    continue;
  }
  break;      // there are no more results
```

5. LA CLASE ResultSet

5.1 Vista Preliminar

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 result set, convirtiendo a ésta en la fila actúal.

El formato general de un result set es una tabla con cabeceras de columna y los valores correspondientes devueltos por la 'query'. Por ejemplo, si la 'query' es `SELECT a, b, c FROM Table1`, el resultado tendrá una forma semejante a :

```
a          b           c
--------   ---------   --------
12345      Cupertino   CA
83472      Redmond     WA
83492      Boston      MA
```

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`, la columna 2 como una `String` y la columna 3 como un array de bytes:

```
java.sql.Statement stmt = conn.createStatement();
ResultSet r = stmt.executeQuery("SELECT a, b, c FROM Table1");
while (r.next())
{
  // print the values for the current row.
  int i = r.getInt("a");
  String s = r.getString("b");
  float f = r.getFloat("c");
  System.out.println("ROW = " + i + " " + s + " " + f);
}
```

5.1.1 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.

En SQL, el cursor resultado para una tabla tiene nombre. Si una base de datos permite upadtes posicionados o deletes posicionados, el nombre del cursor es necesario y debe ser proporcionado como un parámetro del comando update o delete. El nombre del cursor puede obtenerse mediante una llamada al método `getCursorName`.

No todas las bases de datos soportan updates o deletes posicionados. Los métodos `DatabaseMetaData.supportsPositionedDelete` y `DatabaseMetaData.supportsPositionedUpdate` nos permiten descubrir si estas operaciones están soportadas en una conexión dada. Cuando lo están, el driver o la DBMS deben asegurarse que las filas seleccionadas están apropiadamente bloquedas y por tanto que estas operaciones no provoquen actualizaciones anomalas ni otros problemas de concurrencia.

5.1.2 Columnas

Los métodos getXXX 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.

Puede usarse o bien el nombre de la columna o el número de columna para referirse a esta. Por ejemplo: si la columna segunda de un objeto `RecordSet` rs se denomina "`title`" y almacena valores de cadena, cualquiera de los dos ejemplos siguientes nos devolverá el valor almacenado en la columna.

```
String s = rs.getString("title");
String s = rs.getString(2);
```

Nótese que las columnas se numeran de izquierda a derecha comenzando con la columna 1. Además los nombres usados como input en los métodos `getXXX` son insensibles a las mayúsculas.

La opción de usar el nombre de columna fue provista para que el usuario que especifica nombres de columnas en una 'query' pueda usar esos nombres como argumentos de los métodos `getXXX`. 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.

En algunos casos, es posible para una query SQL devolver un result set con más de una columna con el mismo nombre. Si se usa el nombre de columna como argumento en un método `getXXX`, éste devolverá el valor de la primera columna que coincida con el nombre. Por eso, si hay múltiples columnas con el mismo nombre, se necesita usar un índice de columna para asegurarse que se recupera el valor de la columna correcta. Esto puede ser ligeramente más eficiente que usar los números de columna.

Información acerca de las columnas en un `ResultSet` es accesible mediante el método `ResultSet.getMetaData`. El objeto `ResultSetMetaData` devuelto nos da el numero, tipo y propiedades de las columnas de los objetos `ResultSet`.

Si se conoce el nombre de una columna, pero no su indice, puede usarse el método `findColumn` para encontrar el número de columna.

5.1.3 Tipos de datos y conversiones.

Para los métodos getXXX, el driver JDBC intenta convertir los datos subyacentes a tipos de datos Java . Por ejemplo, si el método getXXX es getString y los tipos de datos de la base de datos en la base subyacente es VARCHAR, el driver JDBC convertirá VARCHAR en String de Java. El valor devuelto por getString será un objeto Java de tipo String.

La siguiente tabla muestra que tipos JDBC está permitido devolver para un método getXXX y que tipos JDBC (tipos genéricos de SQL) se recomiendan para recuperarlos. Una x indica un método getXXX legal para un tipo de dato particular. Por ejemplo, cualquier método getXXX excepto getBytes o getBinaryStream puede usarse para recuperar valores de tipo LONGVARCHAR, pero se recomienda usar getAsciiStream o getUnicodeStream, dependiendo de que tipo de dato se devuelve. El método getObject devolverá cualquier tipo de dato como un Object Java y es util cuando los tipos de datos de la DBMS subyacente son abstractos específicos de ésta o cuando una aplicación genérica necesita aceptar cualquier tipo de datos.

El uso de los métodos ResultSet.getXXX recuperan tipos de datos comunes JDBC

Una “x” indica que el método getXXX puede legalmente usarse para recuperar el tipo JDBC dado.

Una “**X**” indica que el método getXXX es el recomendado para recuperar el tipo de dato dado.

5.1.4 Uso de Streams valores muy grandes de filas

ResultSet hace posible el recuperar datos arbitrariamente grandes de tipo LONGVARBINARY o LONGVARCHAR. Los métodos getBytes y getString devuelven valores grandes (hasta los límites impuestos por el valor devuelto por Statement.getMaxFieldSize).

	TINYINT	SMALLINT	INTEGER	BIGINT	REAL	FLOAT	DOUBLE	DECIMAL	NUMERIC	BIT	CHAR	VARCHAR	LONGVARCHAR	BINARY	VARBINARY	LONGVARBINARY	DATE	TIME	TIMESTAMP
getByte	**X**	x	x	x	x	x	x	x	x	x	x	x	x						
getShort	x	**X**	x	x	x	x	x	x	x	x	x	x	x						
getInt	x	x	**X**	x	x	x	x	x	x	x	x	x	x						
getLong	x	x	x	**X**	x	x	x	x	x	x	x	x	x						
getFloat	x	x	x	x	**X**	x	x	x	x	x	x	x	x						
getDouble	x	x	x	x	x	**X**	**X**	x	x	x	x	x	x						
getBigDecimal	x	x	x	x	x	x	x	**X**	**X**	x	x	x	x						
getBoolean	x	x	x	x	x	x	x	x	x	**X**	x	x	x						
getString	x	x	x	x	x	x	x	x	x	x	**X**	**X**	x	x	x	x	x	x	x
getBytes														**X**	**X**	x			
getDate											x	x	x				**X**		x

getTime											x	x	x					**X**	x
getTimestamp											x	x	x				x		**X**
getAsciiStream											x	x	**X**	x	x	x			
getUnicodeStream											x	x	**X**	x	x	x			
getBinaryStream														x	x	**X**			
getObject	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x

De todos modos, puede ser conveniente recuperar datos muy grandes en 'pedazos' de tamaño fijo. Esto se hace mediante la clase `ResultSet` que devuelve 'streams' java.io.Input desde los cuales los datos pueden ser leidos en 'pedazos'. Nòtese que estas corrientes deben ser accedidas inmediatamente porque se cierran automáticamente con la llamada al siguiente método `getXXX` de `ResultSet`. (Este comportamiento está impuesto por las restricciones de implementación de acceso a grandes blob).

El API JDBC tiene tres métodos diferentes para recuperar streams, cada uno con un valor diferente de retorno.

- `getBinaryStream` devuelve una corriente que simplemente suminiistra bytes en 'bruto' desde la base de datos sin ninguna conversión.
- `getAsciiStream` devuelve una corriente con caracteres ASCII
- `getUnicodeStream` devuelve una corriente con caracteres Unicode [1] de 2 bytes.

Notar que esto difiere de las corrientes Java que devuelven bytes sn tipo y pueden (por ejemplo) usarse para ambos caracteres ASCII y Unicode.

A continuación veamos un ejemplo del uso de getAsciiStream:

```
java.sql.Statement stmt = con.createStatement();
ResultSet r = stmt.executeQuery("SELECT x FROM Table2");
// Now retrieve the column 1 results in 4 K chunks:
byte buff = new byte[4096];
while (r.next()) {
  Java.io.InputStream fin = r.getAsciiStream(1);
  for (;;) {
    int size = fin.read(buff);
    if (size == -1) { // at end of stream
          break;
    }
    // Send the newly-filled buffer to some ASCII output stream:
```

---

[1] Unicode

Unicode es un esquema de codificación de caracteres que utiliza 2 bytes por *cada* carácter. ISO (International Standards Organization) define un número dentro del intervalo 0 a 65.535 ($2^{16} – 1$) por cada carácter y símbolo de cada idioma (más algunos espacios vacíos para futuras ampliaciones). En todas las versiones de 32 bits de Windows, el Modelo de objetos componentes (COM), que es la base de las tecnologías OLE y ActiveX, utiliza Unicode. Unicode es totalmente compatible con

```
            output.write(buff, 0, size);
        }
    }
```

5.1.5 Valores resultado NULL

Para determinar si un valor resultado dado es JDBC NULL, primero debe intentarse leer la columna y usar el método `ResultSet.wasNull` para descubrir si el valor devuelto es JDBC NULL

Cuando se ha leido un JDBC NULL usando uno de los métodos `ResultSet.getXXX`, el método `wasNull` devuelve algo de lo siguiente:

- Un valor null de Java para aquellos métodos getXXX que devuelven objetos Java (tales como `getString, getBigDecimal, getBytes, getDate, getTime, getTimestamp, getAsciiStream, getUnicodeStream, getBinaryStream, getObject`).
- Un valor cero para `getByte, getShort, getInt, getLong, getFloat y getDouble`.
- Un valor false para `getBoolean`.

5.1.6 Result sets opcionales o múltiples.

Normalmente cuando se ejecuta una sentencia SQL o bien se usa `excuteQuery` (que devuelve un único `ResultSet`) o bien `executeUpdate` (que puede usarse para cualquier tipo de sentencia de modificación de la base de datos y que devuelve un contador de las filas que han sido afectadas. De cualquier modo, bajo ciertas circunstancias una aplicación puede no saber si una sentencia devolverá un result set hasta que ésta no haya sido ejecutada. Además, ciertos procedimientos almacenados pueden devolver varios result sets y/o update counts.

Para acomodarse a estas situaciones, JDBC provee de un mecanismo por el cual una aplicación puede ejecutar una sentencia y luego procesar una colección arbitraria de result sets y update counts. Este mecanismo se basa en una primera llamada a un método general `execute` y luego llamar a otros tres métodos `getResultSet`, `getUpdateCount` y `getMoreResults`. Estos métodos permiten a una aplicación explorar los resultados de las sentencias y determinar si dan como resultado un result set o un update count.

No se necesita hacer nada para cerrar un ResultSet. Se cierra automáticamente cuando por la `Statement` que la crea cuando se cierra esta, y se reutiliza cuando cuando se recupera el próximo resultado de una secuencia de múltiples resultados.

6. LA CLASE PreparedStatement

6.1 Vista Preliminar

La interfase 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 `setXXX` 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.

6.1.1 Creación de objetos PreparedStatement

El siguiente ejemplo, donde con es un objeto Connection, crea un objeto PreparedStatement conteniendo una sentencia SQL con dos 'encajes' para parámetros IN.

```
PreparedStatement pstmt = con.prepareStatement(
  "UPDATE table4 SET m = ? WHERE x = ?");
```

El objeto `pstmt` contiene ahora la sentencia "`UPDATE table4 SET m= ? WHERE x = ?` ", que ya ha sido enviada a la base de datos y ha sido preparada para su ejecución.

6.1.2 Pasar parámetros IN

Antes de que un objeto `PreparedStatement` sea ejecutado debe fijarse el valor de cada encaje '?'. Se hace esto mediante la llamada a un método `setXXX`, donde XXX es el tipo apropiado para el parámetro. Por ejemplo, si el parámetro tiene un tipo Java `long`, el método a usar será `setLong`. El primero de los argumentos del método `setXXX` es la posición ordinal del parámetro a fijar, y el segundo argumento es el valor que queremos que adquiera el parámetro. Por ejemplo, lo siguiente fija el primer parámetro a 123456789 y el segundo a 10000000.

```
pstmt.setLong(1, 123456789);
pstmt.setLong(2, 100000000);
```

Una vez que el valor ha sido fijado para una sentencia dada, puede usarse para múltiples ejecuciones de esa sentencia hasta que se limpie mediante el método ClearParameters.

En el modo por defecto para una conexión (modo auto-commit activo), cada sentencia es conmitada o rechazada automáticamente cuando se completa.

El mismo objeto `PreparedStatement` puede ejecutarse múltiples veces si la base de datos subyacente y el driver guardan las sentencias abiertas después que hayan sido 'conmitadas'. A menos que se de este caso, no hay un punto en el que intentar mejorar el rendimiento mediante el uso de objetos `PreparedStatement` en lugar de objetos `Statement`.

Usando pstmt, el objeto `PreparedStatement` creado anteriormente, el siguiente ejemplo ilustra como fijar los parámetros de los dos 'encajes' y ejecutar `pstmt` 10 veces. Como se ha mencionado anteriormente, la base de datos no debe cerrar `pstmt`. En este ejemplo, el primer parámetro se fija a "Hi" y permanece constante. El segundo parámetro se fija a un valor diferente en cada ejecución mediante el bucle for comenzando en 0 y terminando en 9.

```
pstmt.setString(1, "Hi");
for (int i = 0; i < 10; i++) {
    pstmt.setInt(2, i);
    int rowCount = pstmt.executeUpdate();
}
```

6.1.3 Conformidad de tipos de datos en parámetros IN

El XXX en los métodos `setXXX` son tipos Java. Estos son los tipos implícitos de JDBC (tipos genéricos SQL) porque el driver mapeará el tipo Java en su correspondiente tipo JDBC (ver la sección 8 para más información sobre el mapeo de tipos), y envían ese tipo JDBC a la base de datos. El siguiente ejemplo fija el segundo parámetro del objeto `PreparedStatement pstmt` a 44 con un tipo Java `short`.

```
pstmt.setShort(2, 44);
```

El driver enviará 44 a la base de datos como un JDBC SMALLINT que es el mapeo estándar para un short Java.

Es responsabilidad del programador asegurarse que el tipo Java en cada parámetro IN mapeado a un tipo de JDBC es compatible con el tipo de dato JDBC esperado por la base de datos. Consideremos el caso en el que la base de datos espera un tipo de datos SMALLINT. SI se usa el método setByte , el driver enviará un JDBC TINYINT a la base de datos. Esto probablemente funcionará porque muchas bases de datos convierten un tipo relacionado en otro, y, generalmente, un tipo TINYINT puede ser usado donde un SMALLINT. De cualquier modo, una aplicación que trabaje para la mayoría de bases de datos posibles, es preferible que use tipos Java que se correspondan con el tipo exacto JDBC esperado por la base de datos. Si el tipo esperado es SMALLINT, usar setShort en vez de setByte y esto hará la aplicación más portable.

6.1.4 Usar setObject

Un programador puede convertir explicitamente un parámetro de entrada en un tipo particular JDBC mediante el uso de `setObject`. Este método puede tomar un tercer argumento que especifica el tipo JDBC objetivo. EL driver convertirá el `Object` Java al tipo especificado JDBC anten de enviarlo a la base de datos.

Si no se da el tipo JDBC, el driver simplemente mapeará el `Object` Java a su tipo JDBC por defecto (usando la tabla de la sección 8) y lo enviará a la base de datos. Esto es similar a lo que ocurre con los métodos `setXXX` regulares. En ambos casos, el driver mapea el tipo Java del valor al tipo JDBC apropiado antes de enviarlo a la base de datos. La diferencia está en que los métodos `setXXX` usan el mapeo estándar de los tipos Java a tipos JDBC, mientras que el método usa el mapeo desde typos Object de Java a tipos JDBC (ver la tabla en la sección 8.6.4).

La capacidad de el método `setObject` para aceptar cualquier objeto Java permite a una aplicación ser genérica y aceptar entradas para un parámetro en tiempo de ejecución. En esta situación el tipo de entrada no es conocido cuando la aplicación es compilada. Mediante el uso de `setObject`, la aplicación puede aceptar cualquier tipo de objeto Java como entrada al tipo JDBC esperado por la base de datos. La tabla de la sección 8.6.5 muestra todas las posibles conversiones qiue `setObject` puede realizar.

6.1.5 Envío de JDBC NULL como un parámetro IN

El método `setNull` permite a los programadores enviar valores JDBC NULL a la base de datos como un parámetro `IN`. Notese de cualquier modo, que debe especificarse el tipo del parámetro.

También se enviará un JDBC NULL a la base de datos cuando un valor Java `null` se pasa mediante un método `setXXX` (si los acepta los objetos Java como argumentos). El método `setObject`, en cualquier caso, puede tomar un valor null únicamente si se ha especificado el tipo JDBC.

6.1.6 Envio de parámetros IN muy grandes.

Los métodos `setBytes` y `setString` son capaces de enviar cantidades ilimitadas de datos. De cualquier forma, a veces los programadores prefiern pasar los grandes blobs de datos en pequeños 'pedazos'. Esto puede realizarse fijando un parámetro IN a una corriente Java. Cuando se ejecuta la sentencia, el driver JDBC realizará repetidas llamadas a esta corriente de entrada, leyendo sus contenidos y transmitiendo estos contenidos con los datos actuales de los parámetros.

JDBC suministra tres métodos para fijar parámetros IN a corrientes de entrada. `setBinaryStream` para corrientes que contienen bytes, `setAsciiStream` para corrientes que contienen caracteres ASCII y `setUnicodeStream` para corrientes que contienen caracteres Unicode. Estos métodos toman un argumento mas que los otros métodos setXXX porque debe especificarse la longitud de la corriente. Esto es necesario porque algunas bases de datos necesitan conocer el tamaño total de los datos a transferir antes de enviarlos.

El siguiente ejemplo ilustra el uso de una corriente para enviar el contenido de un fichero en un parámetro IN

```
java.io.File file = new java.io.File("/tmp/data");
int fileLength = file.length();
java.io.InputStream fin = new java.io.FileInputStream(file);
java.sql.PreparedStatement pstmt = con.prepareStatement(
  "UPDATE Table5 SET stuff = ? WHERE index = 4");
pstmt.setBinaryStream (1, fin, fileLength);
pstmt.executeUpdate();
```

Cuando la sentencia se ejecuta, la corriente de entrada fin será llamada repetidamente hasta completar los datos.

7. LA CLASE CallableStatement

7.1 Vista Preliminar

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 mas 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) o ámbos (parámetros INOUT). 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 encuenta contenido en ellos es opcional, no ofroma 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, alquien 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.

7.1.1 Crear objetos CallableStatement

Los objetos `CallableStatement` se crean con el método `prepareCall` de `Connection`. El siguiente ejemplo crea una instancia de `CallableStatement` que contiene una llamada al procedimiento almacenado `getTestData`, con dos argumentos y no devuelve resultados.

```
CallableStatement cstmt = con.prepareCall(
                "{call getTestData(?, ?)}");
```

donde los encajes '?' son parámetros IN, OUT ó INOUT dependiendo del procedimiento `getTestData`.

7.1.2 Parámetros IN y OUT

El paso de valor para cualquier parámetro IN de un objeto `CallableStatement` se realiza mediante los métodos `setXXX` heredados de `PreparedStatement`. El tipo de el valor a pasar se determina por el método `setXXX` a usar (`setFloat` para pasar un valor `float`, y así).

Si el procedimiento almacenado devuelve parámetros OUT, el tipo JDBC de cada parámetro OUT debe ser registrado antes de que el objeto `CallableStatement` sea ejecutado (Esto es necesario porque algunas DBMS necesitan el tipo JDBC). El registro del tipo JDBC se realiza mediante el método `registerOutParameters`. Después que la sentencia ha sido ejecutada, los métodos `getXXX` de `CallableStatement` recuperan los valores de los parámetros. El método correcto `getXXX` a usar es el tipo Java que corresponde al tipo JDBC registrado para el parámetro. (El mapeo estándar para los tipos JDBC a tipos Java se muestra en la tabla de la sección 8.6.1). En otras palabras `registerOutParameter` usa un tipo

JDBC (por tanto coincide con el tipo con el tipo JDBC que la base de datos devolverá) y `getXXX` 'casts' este a un tipo Java.

Para ilustrar esto, el siguiente ejemplo registra los parámetros OUT, ejecuta el procedimiento almacenado llamado por `cstmt` y recupera los valores devueltos en los parámetros OUT. El método `getByte` recupera un byte Java de el primer parámetro, y `getBigDecimal` recupera un objeto `BigDecimal` (con tres dígitos después del punto decimal) del segundo parámetro OUT:

```
CallableStatement cstmt = con.prepareCall(
                     "{call getTestData(?, ?)}");
cstmt.registerOutParameter(1, java.sql.Types.TINYINT);
cstmt.registerOutParameter(2, java.sql.Types.DECIMAL, 3);
cstmt.executeQuery();
byte x = cstmt.getByte(1);
java.math.BigDecimal n = cstmt.getBigDecimal(2, 3);
```

De modo distinto a ResultSet, CallableStatement no tiene un mecanismo especial para recuperar grandes valores OUT incrementalmente.

7.1.3 Parámetros INOUT

Son parámetros que suminstran entradas así como aceptan salidas. Estos requieren llamar a los métodos apropiados `setXXX` (heredados de `PreparedStatement`) además de llamar al método `registerOutParameter`. Los métodos `setXXX` fijan los valores como parámetros de entrada y `registerOutParameter` registra sus tipos JDBC como parámetros de salida. El método `setXXX` suminstra un valor Java que el driver convierte en un valor JDBC antes de enviarlo a la base de datos.

El tipo JDBC del valor IN y el tipo JDBC para suminstrado al método `registerOutParameter` debe ser el mismo. Luego, para recuperar el valor de salida, se usa el método apropiado `getXXX`. Por ejemplo, un parámetro cuyo tipo Java es `byte` debería usar el método `setByte` para asignar el valor de entrada, debería suplir un TINYINT como tipo JDBC para `registerOutParameter`, y debería usar `getByte` para recuperar el valor de salida (Seccion 8, da más informaciòn y contiene tablas de tipos mapeados).

El siguiente ejemplo asume que existe un procedimiento almacenado `reviseTotal` con un único parámetro INOUT. El método `setByte` fija el valor del parámetro a 25 que es el que el driver enviará a la base de datos como un JDBC TINYINT. Después `registerOutParameter` registrará el parámetro como un JDBC TINYINT. Luego que el procedimiento sea ejecutado se devolverá un nuevo JDBC TINYINT y el método `getByte` lo recuperará como un nuevo valor `byte` Java.

```
CallableStatement cstmt = con.prepareCall(
    "{call reviseTotal(?)}");
cstmt.setByte(1, 25);
cstmt.registerOutParameter(1, java.sql.Types.TINYINT);
cstmt.executeUpdate();
byte x = cstmt.getByte(1);
```

7.1.4 Recuperar parámetros OUT después de resultados

Dadas las limitaciones impuestas pro algunas DBMS, se recomienda en aras de la máxima portabilidad, que todos los resultados generados por la ejecución de un objeto `CallableStatement` deberían recuperarse antes que los parámetros OUT usando los métodos `CallableStatement.getXXX`.

Si un objeto `CallableStatement` devuelve múltiples objetos `ResultSet` (mediante una llamada al método `execute`), todos los resultados deben recuperarse antes que los parámetros OUT. En este caso, debe asegurarse de que todos los resultados han sido accedidos, los métodos de `Statement` `getResultSet`, `getUpdateCount` y `getMoreResults` necesitan ser llamados hasta que no haya más resultados.

Después de hecho esto, los valores de los parámetros OUT pueden ser recuperados mediante los métodos `CallableStatement.getXXX`.

7.1.5 Recuperar valores NULL en parámetros OUT

El valor devuelto en un parámetro OUT puede ser JDBC NULL. Cuando esto ocurre, le valor JDBC NULL se convertirá de forma que el valor devuelto por el método `getXXX` sea null, 0 o false dependiendo del tipo del método `getXXX` usado.Como con los objetos `ResultSet`, la única manera de saber si un valor de 0 o false fue originalmente NULL JDBC es testear el método `wasNull`, que devuelve true si el último valor leído por un método getXXX fue JDBC NULL y false en caso contrario. La sección 5 contiene más información al respecto.

9. EJEMPLO DE CODIGO

```
      // The following code can be used as a template.  Simply
      // substitute the appropriate url, login, and password, and then
      substitute the
      // SQL statement you want to send to the database.



      //-----------------------------------------------------------------
      ---------
      //
      // Module:       SimpleSelect.java
      //
      // Description:  Test program for ODBC API interface.  This java
      application
      // will connect to a JDBC driver, issue a select statement
      // and display all result columns and rows
      //
      // Product:      JDBC to ODBC Bridge
      //
      // Author:       Karl Moss
      //
      // Date: February, 1996
      //
      // Copyright:    1990-1996 INTERSOLV, Inc.
      // This software contains confidential and proprietary
      // information of INTERSOLV, Inc.
      //-----------------------------------------------------------------
      ---------

import java.net.URL;
import java.sql.*;

class SimpleSelect {

        public static void main (String args[]) {
        String url   = "jdbc:odbc:my-dsn";
        String query = "SELECT * FROM emp";

        try {

                // Load the jdbc-odbc bridge driver

                Class.forName ("sun.jdbc.odbc.JdbcOdbcDriver");

                DriverManager.setLogStream(System.out);

                // Attempt to connect to a driver.  Each one
                // of the registered drivers will be loaded until
                // one is found that can process this URL

                Connection con = DriverManager.getConnection (
                        url, "my-user", "my-passwd");
```

```
            // If we were unable to connect, an exception
            // would have been thrown.  So, if we get here,
            // we are successfully connected to the URL

            // Check for, and display and warnings generated
            // by the connect.

            checkForWarning (con.getWarnings ());

            // Get the DatabaseMetaData object and display
            // some information about the connection

            DatabaseMetaData dma = con.getMetaData ();

            System.out.println("\nConnected to " + dma.getURL());
            System.out.println("Driver       " +
                    dma.getDriverName());
            System.out.println("Version      " +
                    dma.getDriverVersion());
            System.out.println("");

            // Create a Statement object so we can submit
            // SQL statements to the driver

            Statement stmt = con.createStatement ();

            // Submit a query, creating a ResultSet object

            ResultSet rs = stmt.executeQuery (query);

            // Display all columns and rows from the result set

            dispResultSet (rs);

            // Close the result set

            rs.close();

            // Close the statement

            stmt.close();

            // Close the connection

            con.close();
    }
    catch (SQLException ex) {

            // A SQLException was generated.  Catch it and
            // display the error information.  Note that there
            // could be multiple error objects chained
            // together
```

```
  System.out.println ("\n*** SQLException caught ***\n");

  while (ex != null) {
          System.out.println ("SQLState: " +
                          ex.getSQLState ());
          System.out.println ("Message:  " + ex.getMessage ());
          System.out.println ("Vendor:   " +
                          ex.getErrorCode ());
          ex = ex.getNextException ();
          System.out.println ("");
          }
  }
  catch (java.lang.Exception ex) {

          // Got some other type of exception.  Dump it.

          ex.printStackTrace ();
  }
}

//-----------------------------------------------------------------------
// checkForWarning
// Checks for and displays warnings.  Returns true if a warning
// existed
//-----------------------------------------------------------------------

private static boolean checkForWarning (SQLWarning warn)
          throws SQLException  {
  boolean rc = false;

  // If a SQLWarning object was given, display the
  // warning messages.  Note that there could be
  // multiple warnings chained together

  if (warn != null) {
          System.out.println ("\n *** Warning ***\n");
          rc = true;
          while (warn != null) {
                  System.out.println ("SQLState: " +
                          warn.getSQLState ());
                  System.out.println ("Message:  " +
                          warn.getMessage ());
                  System.out.println ("Vendor:   " +
                          warn.getErrorCode ());
                  System.out.println ("");
                  warn = warn.getNextWarning ();
          }
  }
  return rc;
}
```

```
	//-----------------------------------------------------------------
	// dispResultSet
	// Displays all columns and rows in the given result set
	//-----------------------------------------------------------------

	private static void dispResultSet (ResultSet rs)
	  throws SQLException
	{
	  int i;

	  // Get the ResultSetMetaData.  This will be used for
	  // the column headings

	  ResultSetMetaData rsmd = rs.getMetaData ();

	  // Get the number of columns in the result set

	  int numCols = rsmd.getColumnCount ();

	  // Display column headings

	  for (i=1; i<=numCols; i++) {
	        if (i > 1) System.out.print(",");
	        System.out.print(rsmd.getColumnLabel(i));
	  }
	  System.out.println("");

	  // Display data, fetching until end of the result set

	  boolean more = rs.next ();
	  while (more) {

	        // Loop through each column, getting the
	        // column data and displaying

	        for (i=1; i<=numCols; i++) {
	              if (i > 1) System.out.print(",");
	              System.out.print(rs.getString(i));
	        }
	        System.out.println("");

	        // Fetch the next result set row

	        more = rs.next ();
	  }
	}
}
```