JDBC

1. **JDBC**
2. **Bases de Datos**
3. **Conectividad JDBC**
    - Acceso de JDBC a Bases de Datos
        - Modelo de 2 Capas
        - Modelo de 3 Capas
    - Tipos de Drivers
        - Puente JDBC-ODBC
        - Java Binario
        - 100% Java / Protocolo Nativo
        - 100% Java / Protocolo Independiente
4. **Primera Aproximación a JDBC**
5. **Transacciones**
6. **Información de la Base de Datos**
7. **Tipos SQL en Java**
8. **Modelo Relacional de Objetos**

9. **Modelo de Conexión**

10. **JDBC y Servlets**

11. **Introducción a SQL**

    - El Modelo Relacional
    - Creación de Tablas
    - Recuperar Información
    - Almacenar Información

12. **Código Independiente y Portable**

JDBC (*Java DataBase Connectivity*) es un API de Java que permite al programador ejecutar instrucciones en lenguaje estándar de acceso a Bases de Datos, **SQL** (*Structured Query Language*, lenguaje estructurado de consultas), que es un lenguaje de muy alto nivel que permite crear, examinar, manipular y gestionar Bases de Datos relacionales. Para que una aplicación pueda hacer operaciones en una Base de Datos, ha de tener una conexión con ella, que se establece a través de un *driver*, que convierte el lenguaje de alto nivel a sentencias de Base de Datos. Es decir, las tres acciones principales que realizará JDBC son las de establecer la conexión a una base de datos, ya sea remota o no; enviar sentencias SQL a esa base de datos y, en tercer lugar, procesar los resultados obtenidos de la base de datos.

2.- Base de datos

Una Base de Datos es una serie de tablas que contienen información ordenada en alguna estructura que facilita el acceso a esas tablas, ordenarlas y seleccionar filas de las tablas según criterios específicos. Las bases de datos generalmente tienen *índices* asociados a alguna de sus columnas, de forma que el acceso sea lo más rápido posible.

Las Bases de Datos son, sin lugar a dudas, las estructuras más utilizadas en ordenadores; ya que son el corazón de sistemas tan complejos como el censo de una nación, la nómina de empleados de una empresa, el sistema de facturación de una multinacional, o el medio por el que nos expiden el billete para las próximas vacaciones.

En el caso, por ejemplo, del registro de trabajadores de una empresa, se puede imaginar una tabla con los nombres de los empleados y direcciones, y sueldos, retenciones y beneficios. Para organizar esta información, se puede empezar con una tabla que contenga los nombres de los empleados, su dirección y su número de teléfono. También se podría incluir la información relativa a su sueldo, categoría, última subida de salario, etc.

¿Podría todo esto colocarse en una sola tabla? Casi seguro que no. Los rangos de salario para diferentes empleados probablemente sean iguales, por lo que se podría optimizar la tabla almacenando solamente el tipo de salario en la tabla de *empleados* y los rangos de salario (en euros) en otra tabla, indexada a través del tipo de salario. Por ejemplo

Key	Nombre	Tipo de Salario
1	Pérez	2
2	García	1
3	Cabrera	2
4	López	3
5	Gómez	1

Tipo de Salario	Mínimo	Máximo
1	1100	1200
2	1200	1500
3	1500	1800

Los datos de la columna *Tipo de Salario* están referidos a la segunda tabla. Se pueden imaginar muchas categorías para estas tablas secundarias, como por ejemplo la provincia de residencia y los tipos de retención de Hacienda, o si tiene seguro de vida, vivienda propia,

coche, apartamento en la playa, casa en el campo, etc. Cada tabla tiene una primera columna que sirve de clave para las demás columnas, que ya contienen datos. La construcción de tablas en las bases de datos es tanto un arte como una ciencia, y su estructura está referida por su *forma normal*. Las tablas se dice que están en primera, segunda o tercera forma normal, o de modo abreviado como 1NF, 2NF o 3NF.

- Cada celda de la tabla debe tener solamente un valor (nunca un conjunto de valores). (**1NF**)

- 1NF y cada columna que no es clave depende completamente de la columna clave. Esto significa que hay una relación uno a uno entre la clave primaria y las restantes celdas de la fila. (**2NF**)

- 2NF y todas las columnas que no son clave son mutuamente independientes. Esto significa que no hay columnas de datos que contengan datos calculados a partir de los datos de otras columnas. (**3NF**)

Actualmente todas las bases de datos se construyen de forma que todas sus tablas están en la Tercera Forma Normal (3NF); es decir, que las bases de datos están constituidas por un número bastante alto de tablas, cada una de ellas con relativamente pocas columnas de información.

A la hora de extraer datos de las tablas, se realizan consultas contra ella. Por ejemplo, si se quiere generar una tabla de empleados y sus rangos de salario para algún tipo de plan especial de la empresa, esa tabla no existe directamente en la base de datos, así que debe construirse haciendo una consulta a la base de datos, y se obtendría una tabla que contendría la siguiente información:

Nombre	Mínimo	Máximo
Pérez	1200	1500
García	1100	1200
Cabrera	1200	1500
López	1500	1800
Gómez	1100	1200

O quizá ordenada por el incremento de salario:

Nombre	Mínimo	Máximo
García	1100	1200
Gómez	1100	1200
Pérez	1200	1500
Cabrera	1200	1500
López	1500	1800

Para generar la tabla anterior se habría de realizar una consulta a la base de datos que tendría la siguiente forma:

```
SELECT DISTINCTROW Empleados.Nombre, TipoDeSalario.Minimo,
    TipoDeSalario.Maximo FROM Empleados INNER JOIN
TipoDeSalario ON
    Empleados.SalarioKey = TipoDeSalario.SalarioKey
    ORDER BY TipoDeSalario.Minimo;
```

El lenguaje en que se ha escrito la consulta es SQL, actualmente soportado por casi todas las bases de datos a lo largo del mundo. Los estándares de SQL han sido varios a lo largo de los años y muchas de las bases de datos para PC soportan alguno de esos tipos. El estándar **SQL-92** es el que se considera origen de todas las actualizaciones. Hay que tener en cuenta, que hay posteriores versiones de SQL, perfeccionadas y extendidas para explotar características únicas de bases de datos particulares; así que no conviene separarse del estándar básico si se pretende hacer una aplicación que pueda atacar a cualquier tipo de base de datos. Al final del capítulo, el lector puede encontrar una pequeña revisión del lenguaje SQL.

Desde que los PC se han convertido en una herramienta presente en la mayor parte de las oficinas, se han desarrollado un gran número de bases de datos para ejecutarse en ese tipo de plataformas; desde bases de datos muy elementales como *Microsoft Works*, hasta otras ya bastante sofisticadas como *Approach*, *dBase*, *Paradox*, *Access* y *Foxbase*.

Otra categoría ya más seria de bases de datos para PC son aquellas que usan la plataforma PC como cliente para acceder a un servidor. Estas bases de datos son *IBM DB/2*, *Microsoft SQL Server*, *Oracle*, *Sybase*, *SQLBase*, *Informix*, *XDB* y *Postgres*. Todas estas bases de datos soportan varios dialectos similares de SQL, y todas parecen, a primera vista, intercambiables. La razón de que no sean

intercambiables, por supuesto, es que cada una está diseñada con unas características de rendimiento distintas, con un interfaz de usuario y programación diferente. Aunque todas ellas soportan SQL y la programación es similar, cada base de datos tiene su propia forma de recibir las consultas SQL y su propio modo de devolver los resultados. Aquí es donde aparece el siguiente nivel de estandarización, de la mano de **ODBC** (*Open DataBase Conectivity*).

La idea es que se pueda escribir código independientemente de quien sea el propietario de la base de datos a la que se quiera acceder, de forma que se puedan extraer resultados similares de diferentes tipos de bases de datos sin necesidad de tocar el código del programa. Si se consiguiese escribir alguna forma de trazadores para estas bases de datos que tuviesen un interfaz similar, el objetivo no sería difícil de alcanzar.

Microsoft hizo su primer intento en 1992, con la especificación que llamaron *Object Database Connectivity*; y que se suponía iba a ser la respuesta a la conexión a cualquier tipo de base de datos desde Windows. Como en toda aplicación informática, esta no fue la única versión, sino que hubo varias hasta llegar a la de 1994, que era más rápida y más estable, además de ser la primera de las versiones de 32 bits. Y, por si fuera poco, ODBC comenzó a desplazarse a otras plataformas diferentes de Windows, acaparando además de los PC también el mundo de las estaciones de trabajo. Tanto es así, que ahora casi todos los fabricantes de bases de datos proporcionan un *driver* ODBC para acceder a su base de datos.

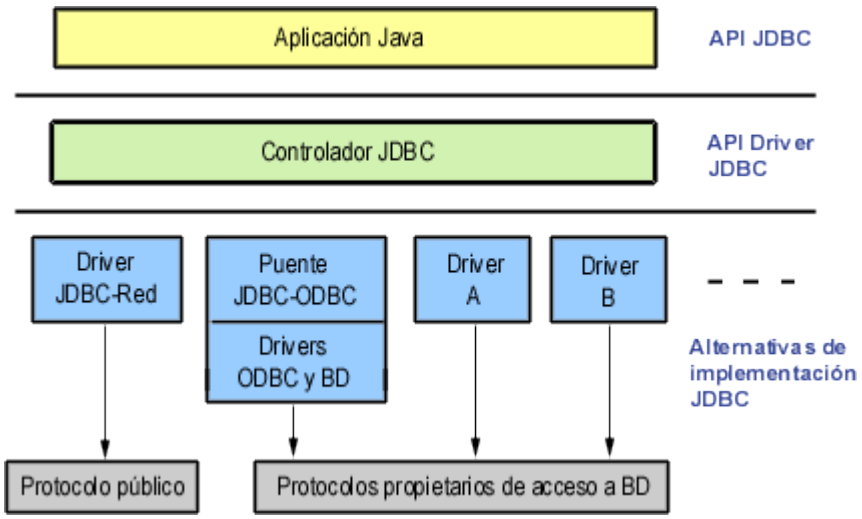
Sin embargo, ODBC dista mucho de ser la panacea que en un principio se podía pensar y que Microsoft se encargó de hacer creer. Muchos fabricantes de bases de datos soportan ODBC como un *interfaz alternativo* al suyo estándar, y la programación en ODBC no es nada, pero que nada trivial; incluyendo toda la parafernalia de la programación para Windows con *handles*, punteros y opciones que son duras de asimilar. Finalmente, ODBC no es un estándar libre, ha sido desarrollado y es propiedad de Microsoft, lo cual, dados los vientos que soplan en este competitivo mundo de las compañías de software, hace su futuro difícil de predecir.

3.- Conectividad JDBC

Para la gente del mundo Windows, JDBC es para Java lo que **ODBC** es para Windows. Windows en general no sabe nada acerca de las bases de datos, pero define el estándar ODBC consistente en un conjunto de primitivas que cualquier *driver* o fuente ODBC debe ser capaz de entender y manipular. Los programadores que a su vez deseen escribir programas para manejar bases de datos genéricas en Windows utilizan las llamadas ODBC.

Con JDBC ocurre exactamente lo mismo: JDBC es una especificación de un conjunto de clases y métodos de operación que permiten a cualquier programa Java acceder a sistemas de bases de datos de forma homogénea. Lógicamente, al igual que ODBC, la aplicación de Java debe tener acceso a un driver JDBC adecuado. Este driver es el que implementa la funcionalidad de todas las clases de acceso a datos y proporciona la comunicación entre el API JDBC y la base de datos real.

La necesidad de JDBC, a pesar de la existencia de ODBC, viene dada porque ODBC es un interfaz escrito en lenguaje C, que al no ser un lenguaje portable, haría que las aplicaciones Java también perdiesen la portabilidad. Y además, ODBC tiene el inconveniente de que se ha de instalar manualmente en cada máquina; al contrario que los drivers JDBC, que al estar escritos en Java son automáticamente instalables, portables y seguros.



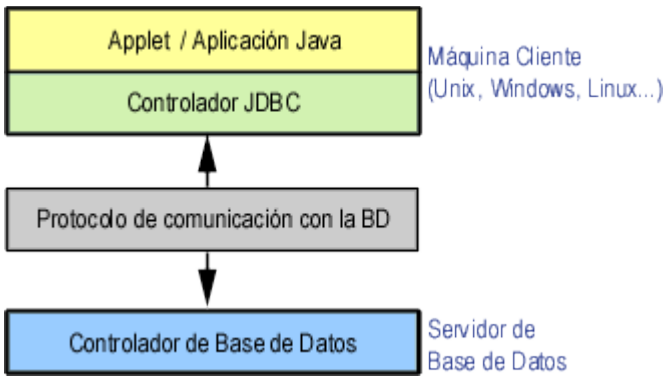
Toda la conectividad de bases de datos de Java se basa en sentencias SQL, por lo que se hace imprescindible un conocimiento adecuado de SQL para realizar cualquier clase de operación de bases de datos. Aunque, afortunadamente, casi todos los entornos de desarrollo Java ofrecen componentes visuales que proporcionan una funcionalidad suficientemente potente sin necesidad de que sea necesario utilizar SQL, aunque para usar directamente el JDK se haga imprescindible. La especificación JDBC requiere que cualquier driver JDBC sea compatible con al menos el nivel «de entrada» de ANSI SQL 92 (*ANSI SQL 92 Entry Level*).

Acceso de JDBC a Bases de Datos

El API JDBC soporta dos modelos diferentes de acceso a Bases de Datos, los modelos de dos y tres capas.

Modelo de dos capas

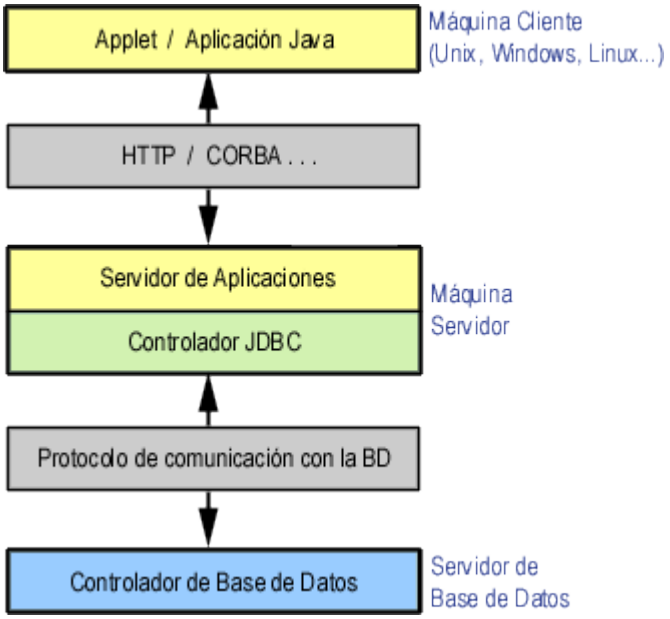
Este modelo se basa en que la conexión entre la aplicación Java o el applet que se ejecuta en el navegador, se conectan directamente a la base de datos.



Esto significa que el driver JDBC específico para conectarse con la base de datos, debe residir en el sistema local. La base de datos puede estar en cualquier otra máquina y se accede a ella mediante la red. Esta es la configuración de típica *Cliente/Servidor*: el programa cliente envía instrucciones SQL a la base de datos, ésta las procesa y envía los resultados de vuelta a la aplicación.

Modelo de tres capas

En este modelo de acceso a las bases de datos, las instrucciones son enviadas a una capa intermedia entre Cliente y Servidor, que es la que se encarga de enviar las sentencias SQL a la base de datos y recoger el resultado desde la base de datos. En este caso el usuario no tiene contacto directo, ni a través de la red, con la máquina donde reside la base de datos.



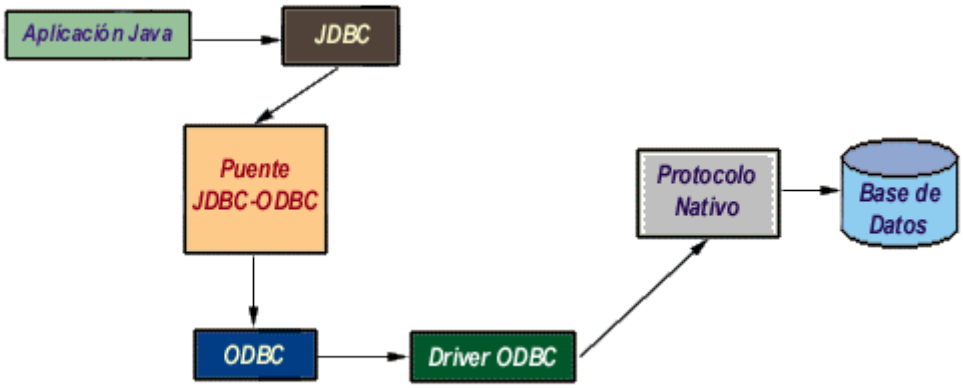
Este modelo presenta la ventaja de que el nivel intermedio mantiene en todo momento el control del tipo de operaciones que se realizan contra la base de datos, y además, está la ventaja adicional de que los drivers JDBC no tienen que residir en la máquina cliente, lo cual libera al usuario de la instalación de cualquier tipo de driver.

Tipos de drivers

Un driver JDBC puede pertenecer a una de cuatro categorías diferentes en cuanto a la forma de operar.

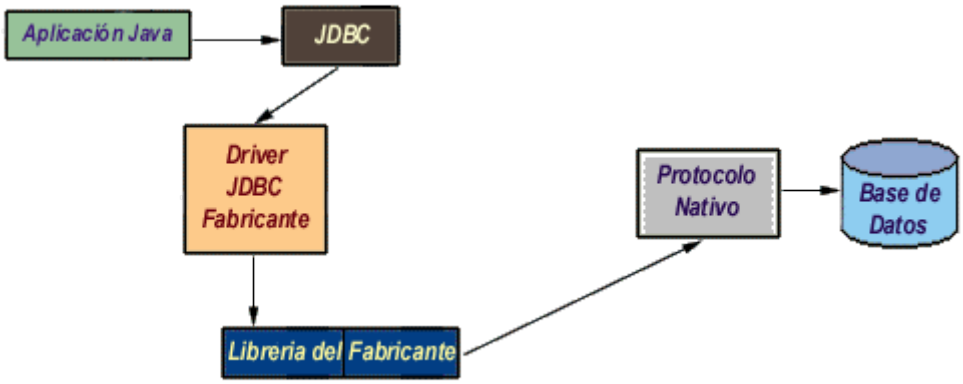
Puente JDBC-ODBC

La primera categoría de drivers es la utilizada por Sun inicialmente para popularizar JDBC y consiste en aprovechar todo lo existente, estableciendo un puente entre JDBC y ODBC. Este driver convierte todas las llamadas JDBC a llamadas ODBC y realiza la conversión correspondiente de los resultados.



La ventaja de este driver, que se proporciona con el JDK, es que Java dispone de acceso inmediato a todas las fuentes posibles de bases de datos y no hay que hacer ninguna configuración adicional aparte de la ya existente. No obstante, tiene dos desventajas muy importantes; por un lado, la mayoría de los drivers ODBC a su vez convierten sus llamadas a llamadas a una librería nativa del fabricante DBMS, con lo cual la lentitud del driver JDBC-ODBC puede ser exasperante, al llevar dos capas adicionales que no añaden funcionalidad alguna; y por otra parte, el puente JDBC-ODBC requiere una instalación ODBC ya existente y configurada.

Lo anterior implica que para distribuir con seguridad una aplicación Java que use JDBC habría que limitarse en primer lugar a entornos Windows (donde está definido ODBC) y en segundo lugar, proporcionar los drivers ODBC adecuados y configurarlos correctamente. Esto hace que este tipo de drivers esté totalmente descartado en el caso de aplicaciones comerciales, e incluso en cualquier otro desarrollo, debe ser considerado como una solución transitoria, porque el desarrollo de drivers totalmente en Java hará innecesario el uso de estos puentes.



Este *driver* se salta la capa ODBC y habla directamente con la librería nativa del fabricante del sistema DBMS (como pudiera ser *DB-Library* para Microsoft SQL Server o *CT-Lib* para Sybase SQL Server). Este driver es un driver *100% Java* pero aún así necesita la existencia de un código binario (la librería DBMS) en la máquina del cliente, con las limitaciones y problemas que esto implica.

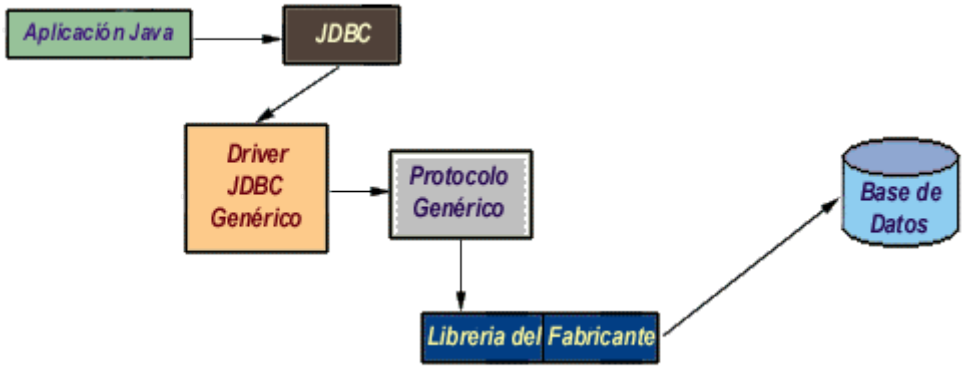
100% Java/Protocolo nativo

Es un driver realizado completamente en Java que se comunica con el servidor DBMS utilizando el protocolo de red nativo del servidor. De esta forma, el driver no necesita intermediarios para hablar con el servidor y convierte todas las peticiones JDBC en peticiones de red contra el servidor. La ventaja de este tipo de driver es que es una solución *100% Java* y, por lo tanto, independiente de la máquina en la que se va a ejecutar el programa.

Igualmente, dependiendo de la forma en que esté programado el driver, puede no necesitar ninguna clase de configuración por parte del usuario. La única desventaja de este tipo de *drivers* es que el cliente está ligado a un servidor DBMS concreto, ya que el protocolo de red que utiliza MS SQL Server por ejemplo no tiene nada que ver con el utilizado por DB2, PostGres u Oracle. La mayoría de los

fabricantes de bases de datos han incorporado a sus propios drivers JDBC del segundo o tercer tipo, con la ventaja de que no suponen un coste adicional.

100% Java/Protocolo independiente



Esta es la opción más flexible, se trata de un *driver 100% Java / Protocolo independiente*, que requiere la presencia de un intermediario en el servidor. En este caso, el driver JDBC hace las peticiones de datos al intermediario en un protocolo de red independiente del servidor DBMS. El intermediario a su vez, que está ubicado en el lado del servidor, convierte las peticiones JDBC en peticiones nativas del sistema DBMS. La ventaja de este método es inmediata: el programa que se ejecuta en el cliente, y aparte de las ventajas de los drivers *100% Java*, también presenta la independencia respecto al sistema de bases de datos que se encuentra en el servidor.

De esta forma, si una empresa distribuye una aplicación Java para que sus usuarios puedan acceder a su servidor MS SQL y posteriormente decide cambiar el servidor por Oracle, PostGres o DB2, no necesita volver a distribuir la aplicación, sino que únicamente debe reconfigurar la aplicación residente en el servidor que se encarga de transformar las peticiones de red en peticiones nativas. La única desventaja de este tipo de drivers es que la aplicación intermediaria es una aplicación independiente que suele tener un coste adicional por servidor físico, que hay que añadir al coste del servidor de bases de datos.

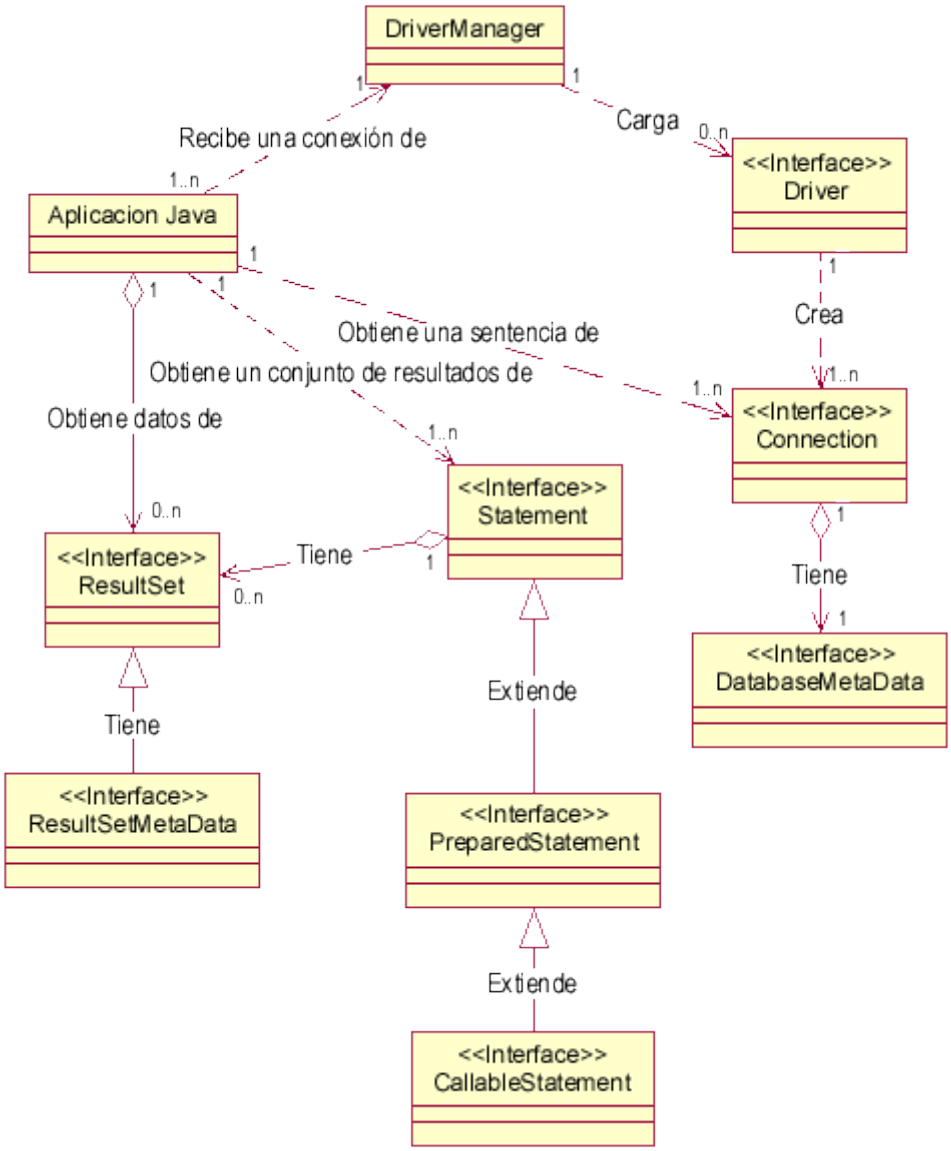
En el Tutorial todos los ejemplos estarán referidos al puente JDBC-ODBC y a la conectividad JDBC con servidores SQL comerciales Si el lector quiere seguir experimentando sin necesidad de adquirir

paquetes comerciales, en el caso de Windows puede utilizar el conocido *mSQL* (mini SQL), un servidor SQL escrito por David J. Hughes y que es de distribución *shareware*, siendo gratuita para fines no comerciales y de investigación; y en el caso de Linux, aunque también se puede utilizar *mSQL*, el autor recomienda ***PostGres***, que es un sistema de base de datos relacional que une las estructuras clásicas con los conceptos de programación orientada a objetos.

Lógicamente, el lector no debe esperar de *mSQL* una potencia, rendimiento o compatibilidad con ANSI SQL como la que tienen los servidores SQL comerciales, aunque en el caso de *PostGres* la funcionalidad es al menos igual a otras bases de datos profesionales existentes en el mercado; y aunque, como casi todos los productos Linux, carece de un interfaz cómodo, sí permite realizar las tareas de administración y explotación con bastante comodidad.

4.- Aproximación a JDBC

JDBC define ocho interfaces para operaciones con bases de datos, de las que se derivan las clases correspondientes. La figura siguiente, en formato OMT, con nomenclatura UML, muestra la interrelación entre estas clases según el modelo de objetos de la especificación de JDBC.



La clase que se encarga de cargar inicialmente todos los drivers JDBC disponibles es **DriverManager**. Una aplicación puede utilizar **DriverManager** para obtener un objeto de tipo conexión, **Connection**, con una base de datos. La conexión se especifica

siguiendo una sintaxis basada en la especificación más amplia de los URL, de la forma

jdbc:subprotocolo//servidor:puerto/base de datos

Por ejemplo, si se utiliza mSQL el nombre del subprotocolo será **msql**. En algunas ocasiones es necesario identificar aún más el protocolo. Por ejemplo, si se usa el puente JDBC-ODBC no es suficiente con **jdbc:odbc**, ya que pueden existir múltiples drivers ODBC, y en este caso, hay que especificar aún más, mediante **jdbc:odbc:fuente de datos**.

Una vez que se tiene un objeto de tipo **Connection**, se pueden crear sentencias, **statements**, ejecutables. Cada una de estas sentencias puede devolver cero o más resultados, que se devuelven como objetos de tipo **ResultSet**.

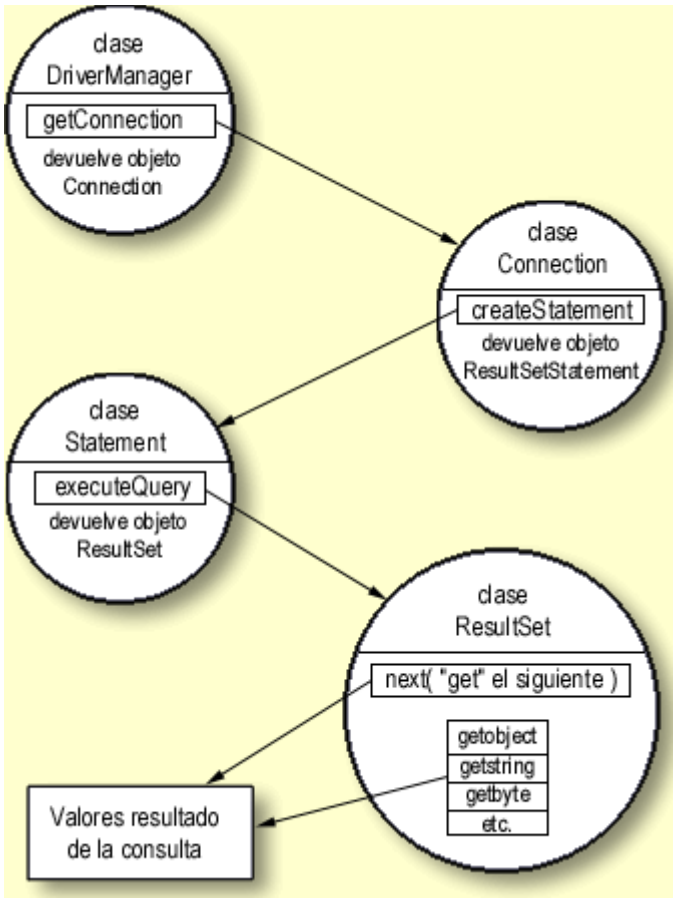
Y la tabla siguiente muestra la misma lista de clases e interfaces junto con una breve descripción.

Clase/Interface	*Descripción*
Driver	Permite conectarse a una base de datos: cada gestor de base de datos requiere un driver distinto
DriverManager	Permite gestionar todos los drivers instalados en el sistema
DriverPropertyInfo	Proporciona diversa información acerca de un driver
Connection	Representa una conexión con una base de datos. Una aplicación puede tener más de una conexión a más de una base de datos
DatabaseMetadata	Proporciona información acerca de una Base de Datos, como las tablas que contiene, etc.
Statement	Permite ejecutar sentencias SQL sin parámetros
PreparedStatement	Permite ejecutar sentencias SQL con parámetros de entrada/TD>

CallableStatement	Permite ejecutar sentencias SQL con parámetros de entrada y salida, típicamente procedimientos almacenados
ResultSet	Contiene las filas o registros obtenidos al ejecutar un SELECT
ResultSetMetadata	Permite obtener información sobre un **ResultSet**, como el número de columnas, sus nombres, etc.

La primera aplicación que se va a crear simplemente crea una tabla en el servidor, utilizando para ello el puente JDBC-ODBC, siendo la fuente de datos un servidor SQL Server. Si el lector desea utilizar otra fuente ODBC, no tiene más que cambiar los parámetros de *getConnection()* en el código fuente. El establecimiento de la conexión es, como se puede es fácil suponer, la parte que mayores problemas puede dar en una aplicación de este tipo. Si algo no funciona, cosa más que probable en los primeros intentos, es muy recomendable activar la traza de llamadas ODBC desde el panel de control. De esta forma se puede ver lo que está haciendo exactamente el *driver* JDBC y por qué motivo no se está estableciendo la conexión.

El siguiente diagrama relaciona las cuatro clases principales que va a usar cualquier programa Java con JDBC, y representa el esqueleto de cualquiera de los programas que se desarrollan para atacar a bases de datos.



La aplicación siguiente es un ejemplo en donde se aplica el esquema anterior, se trata de instalación java2101.java, crea una tabla y rellena algunos datos iniciales.

```
import java.sql.*;

class java2101 {
  static public void main( String[] args ) {
    Connection conexion;
    Statement sentencia;
    ResultSet resultado;

    System.out.println( "Iniciando programa." );

    // Se carga el driver JDBC-ODBC
    try {
      Class.forName( "sun.jdbc.odbc.JdbcOdbcDriver" );
   } catch( Exception e ) {
      System.out.println( "No se pudo cargar el puente
JDBC-ODBC." );
      return;
    }

    try {
      // Se establece la conexión con la base de datos
```

```
      conexion = DriverManager.getConnection
( "jdbc:odbc:Tutorial","","" );
      sentencia = conexion.createStatement();
      try {
         // Se elimina la tabla en caso de que ya existiese
         sentencia.execute( "DROP TABLE AGENDA" );
      } catch( SQLException e ) {};

      // Esto es código SQL
      sentencia.execute( "CREATE TABLE AMIGOS ("+
        " NOMBRE VARCHAR(15) NOT NULL, " +
        " APELLIDOS VARCHAR(30) NOT NULL, " +
        " CUMPLE DATETIME) " );
      sentencia.execute( "INSERT INTO AMIGOS " +
        "VALUES('JOSE','GONZALEZ','03/15/1973')" );
      sentencia.execute( "INSERT INTO AMIGOS " +
        "VALUES('PEDRO','GOMEZ','08/15/1961')" );
      sentencia.execute( "INSERT INTO AMIGOS " +
        "VALUES('GONZALO','PEREZ', NULL)" );
    } catch( Exception e ) {
      System.out.println( e );
      return;
    }
  System.out.println( "Creacion finalizada." );
  }
}
```

Las partes más interesantes del código son las que se van a revisar a continuación, profundizando en cada uno de los pasos.

Lo primero que se hace es importar toda la funcionalidad de JDBC, a través de la primera sentencia ejecutable del programa.

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

Las siguientes líneas son las que cargan el puente JDBC-ODBC, mediante el método *forName()* de la clase **Class**.

```
try {
  Class.forName( "sun.jdbc.odbc.JdbcOdbcDriver" );
  } catch( Exception e ) {
    System.out.println( "No se pudo cargar el puente JDBC-
ODBC." );
    return;
  }
```

En teoría esto no es necesario, ya que **DriverManager** se encarga de leer todos los *drivers* JDBC compatibles, pero no siempre ocurre así, por lo que es mejor asegurarse. El método *forName()* localiza, lee y enlaza dinámicamente una clase determinada. Para *drivers* JDBC, la sintaxis que JavaSoft recomienda de *forName()* es

nombreEmpresa.nombreBaseDatos.nombreDriver, y el *driver* deberá estar ubicado en el directorio *nombreEmpresa\nombreBaseDatos\nombreDriver.class* a partir del directorio indicado por la variable de entorno *CLASSPATH*. En este caso se indica que el puente JDBC-ODBC que se desea leer es precisamente el de Sun.

Si por cualquier motivo no es posible conseguir cargar **JdbcOdbcDriver.class**, se intercepta la excepción y se sale del programa. En este momento es la hora de echar mano de la información que puedan proporcionar las trazas ODBC.

La carga del driver también se puede especificar desde la línea de comandos al lanzar la aplicación:

```
java -Djdbc.drivers=sun.jdbc.odbc.JdbcOdbcDriver ElPrograma
```

A continuación, se solicita a **DriverManager** que proporcione una conexión para una fuente de datos ODBC. El parámetro **jdbc:odbc:Tutorial** especifica que la intención es acceder a la fuente de datos con nombre **Tutorial***, Data Source Name* o DSN, en la terminología ODBC.

```
conexion = DriverManager.getConnection
("jdbc:odbc:Tutorial","","" );
```

El segundo y tercer parámetro son el nombre del usuario y la clave con la cual se intentará la conexión. En este caso el acceso es libre, para acceder como administrador del sistema en el caso de un servidor MS SQL se usa la cuenta *sa* o *system administrator*, cuya cuenta de acceso no tiene clave definida; en caso de acceder a un servidor MS Access, la cuenta del administrador es *admin* y también sin clave definida. Esta es la única línea que con seguridad habrá de cambiar el programador para probar sus aplicaciones. *getConnection* admite también una forma con un único parámetro (el URL de la base de datos), que debe proporcionar toda la información de conexión necesaria al *driver* JDBC correspondiente. Para el caso JDBC-ODBC, se puede utilizar la sentencia equivalente:

DriverManager.getConnection ("jdbc:odbc:SQL;UID=sa;PWD=");

Para el resto de los *drivers* JDBC, habrá que consultar la documentación de cada *driver* en concreto.

Inmediatamente después de obtener la conexión, en la siguiente línea

```
sentencia = conexion.createStatement();
```

se solicita que proporcione un objeto de tipo *Statement* para poder ejecutar sentencias a través de esa conexión. Para ello se dispone de los métodos *execute(String sentencia)* para ejecutar una petición SQL que no devuelve datos o *executeQuery(String sentencia)* para ejecutar una consulta SQL. Este último método devuelve un objeto de tipo *ResultSet*.

Una vez que se tiene el objeto *Statement* ya se pueden lanzar consultas y ejecutar sentencias contra el servidor. A partir de aquí el resto del programa realmente es SQL «adornado»: en la línea:

```
sentencia.execute( "DROP TABLE AMIGOS" );
```

se ejecuta *DROP TABLE AMIGOS* para borrar cualquier tabla existente anteriormente. Puesto que este ejemplo es una aplicación «de instalación» y es posible que la tabla *AMIGOS* no exista, dando como resultado una excepción, se aísla la *sentencia.execute()* mediante un *try* y un *catch*.

La línea siguiente ejecuta una sentencia SQL que crea la tabla *AMIGOS* con tres campos: *NOMBRE, APELLIDOS* y *CUMPLE*. De ellos, únicamente el tercero, correspondiente al cumpleaños, es el que puede ser desconocido, es decir, puede contener valores nulos.

```
sentencia.execute( "CREATE TABLE AMIGOS ("+
        " NOMBRE VARCHAR(15) NOT NULL, " +
        " APELLIDOS VARCHAR(30) NOT NULL, " +
        " CUMPLE DATETIME) " );
```

Y ya en las líneas siguientes se ejecutan sentencias *INSERT* para rellenar con datos la tabla. En todo momento se ha colocado un *try ... catch* exterior para interceptar cualquier excepción que puedan dar las sentencias. En general, para *java.sql* está definida una clase especial de excepciones que es *SQLException*. Se obtendrá una excepción de este tipo cuando ocurra cualquier error de proceso de JDBC, tanto si es a nivel JDBC como si es a nivel inferior (ODBC o de protocolo).

Por ejemplo, si en lugar de *GONZALO* en la línea correspondiente a la última inserción en la Base de Datos, se intenta añadir un nombre nulo (*NULL*), se generará una excepción *SQLException* con el mensaje

[Microsoft][ODBC SQL Server Driver][SQL Server]Attempt to insert the value NULL into column 'NOMBRE', table 'master.dbo.AGENDA'; column does not allow nulls. INSERT fails.

que en el caso de *Microsoft Access* sería:

[Microsoft][ODBC Microsoft Access 97 Driver] The field 'AGENDA.NOMBRE' can't contain a Null value because the Required property for this field is set to True. Enter a value in this field.

En román paladino, el hecho de que la columna *NOMBRE* esté definida como *NOT NULL*, hace que no pueda quedarse vacía.

Ahora se verán los pasos que hay que dar para obtener información a partir de una base de datos ya creada. Como se ha dicho anteriormente, se utilizará *executeQuery* en lugar de *execute* para obtener resultados. Se sustituyen las líneas que contenían esa sentencia por :

```
resultado = sentencia.executeQuery( "SELECT * FROM
AMIGOS" );
while( resultado.next() ) {
  String nombre = resultado.getString( "NOMBRE" );
  String apellidos = resultado.getString( "APELLIDOS" );
  String cumple = resultado.getString( "CUMPLE" );
  System.out.println( "El aniversario de D. " + nombre + "
"
    + apellidos + ", se celebra el " + cumple );
  }
```

En este caso, en la primera línea se utiliza *executeQuery* para obtener el resultado de *SELECT * FROM AMIGOS*. Mediante *resultado.next()* la posición se situará en el «siguiente» elemento del resultado, o bien sobre el primero si todavía no se ha utilizado. La función *next()* devuelve *true* o *false* si el elemento existe, de forma que se puede iterar mediante *while (resultado.next())* para tener acceso a todos los elementos.

A continuación, en las líneas siguientes se utilizan los métodos *getXXX()* de resultado para tener acceso a las diferentes columnas. El acceso se puede hacer por el nombre de la columna, como en las dos primeras líneas, o bien mediante su ubicación relativa, como en la última línea. Además de *getString()* están disponibles *getBoolean(), getByte(), getDouble(), getFloat(), getInt(), getLong(), getNumeric(),*

getObject(), getShort(), getDate(), getTime() y *getUnicodeStream()*, cada uno de los cuales devuelve la columna en el formato correspondiente, si es posible.

Después de haber trabajado con una sentencia o una conexión es recomendable cerrarla mediante *sentencia.close()* o *conexión.close()*. De forma predeterminada los *drivers* JDBC deben hacer un *COMMIT* de cada sentencia. Este comportamiento se puede modificar mediante el método *Connection.setAutoCommit(boolean nuevovalor)*. En el caso de que se establezca *AutoCommit* a *false*, será necesario llamar de forma explícita a *Connection.commit()* para guardar los cambios realizados o *Connection.rollback()* para deshacerlos.

Como el lector habrá podido comprobar hasta ahora, no hay nada intrínsecamente difícil en conectar Java con una base de datos remota. Los posibles problemas de conexión que puede haber (selección del *driver* o fuente de datos adecuada, obtención de acceso, etc.), son problemas que se tendrían de una u otra forma en cualquier lenguaje de programación.

El objeto **ResultSet** devuelto por el método *executeQuery()*, permite recorrer las filas obtenidas, no proporciona información referente a la estructura de cada una de ellas; para ello se utiliza **ResultSetMetaData**, que permite obtener el tipo de cada campo o columna, su nombre, si es del tipo autoincremento, si es sensible a mayúsculas, si se puede escribir en dicha columna, si admite valores nulos, etc.

Para obtener un objeto de tipo **ResultSetMetaData** basta con llamar al método *getMetaData()* del objeto **ResultSet**.

En la lista siguiente aparecen algunos de los métodos más importantes de **ResultSetMetaData**, que permiten averiguar toda la información necesaria para formatear la información correspondiente a una columna, etc.

getCatalogName()
Nombre de la columna en el catálogo de la base de datos

getColumnName()
Nombre de la columna

getColumnLabel()
Nombre a utilizar a la hora de imprimir el nombre de la columna

getColumnDisplaySize()
Ancho máximo en caracteres necesario para mostrar el contenido de la columna

getColumnCount()
Número de columnas en el **ResultSet**

getTableName()
Nombre de la tabla a que pertenece la columna

getPrecision()
Número de dígitos de la columna

getScale()
Número de decimales para la columna

getColumnType()
Tipo de la columna (uno de los tipos SQL en **java.sql.Types**)

getColumnTypeName()
Nombre del tipo de la columna

isSigned()
Para números, indica si la columna corresponde a un número con signo

isAutoIncrement()
Indica si la columna es de tipo autoincremento

isCurrency()
Indica si la columna contiene un valor monetario

isCaseSensitive()
Indica si la columna contiene un texto sensible a mayúsculas

isNullable()
Indica si la columna puede contener un `NULL` SQL. Puede devolver los valores
`columnNoNulls`, `columnNullable` o `columnNullableUnknown`, miembros finales
estáticos de **ResultSetMetaData** (constantes)

isReadOnly()
Indica si la columna es de solo lectura

isWritable()
Indica si la columna puede modificarse, aunque no lo garantiza

isDefinitivelyWritable()
Indica si es absolutamente seguro que la columna se puede modificar

isSearchable()
Indica si es posible utilizar la columna para determinar los criterios de búsqueda de un SELECT

getSchemaName()
Devuelve el texto correspondiente al esquema de la base de datos para esa columna

En general pues, los objetos que se van a poder encontrar en una aplicación que utilice JDBC, serán los que se indican a continuación.

Connection

Representa la conexión con la base de datos. Es el objeto que permite realizar las consultas SQL y obtener los resultados de dichas consultas. Es el objeto base para la creación de los objetos de acceso a la base de datos.

DriverManager

Encargado de mantener los drivers que están disponibles en una aplicación concreta. Es el objeto que mantiene las funciones de administración de las operaciones que se realizan con la base de datos.

Statement

Se utiliza para enviar las sentencias SQL simples, aquellas que no necesitan parámetros, a la base de datos.

PreparedStatement

Tiene una relación de herencia con el objeto **Statement**, añadiéndole la funcionalidad de poder utilizar parámetros de entrada. Además, tiene la particularidad de que la pregunta ya ha sido compilada antes de ser realizada, por lo que se denomina *preparada*. La principal ventaja, aparte de la utilización de parámetros, es la rapidez de ejecución de la pregunta.

CallableStatement

Tiene una relación de herencia cn el objeto PreparedStatement. Permite utilizar funciones implementadas directamente sobre el sistema de gestión de la base de datos. Teniendo en cuenta que éste posee información adicional sobre el uso de las estructuras internas, índices, etc.; las funciones se realizarán de forma más eficiente. Este tipo de operaciones es muy utilizada en el caso de ser funciones muy complicadas o bien que vayan a ser ejecutadas varias veces a lo largo del tiempo de vida de la aplicación.

ResultSet

Contiene la tabla resultado de la pregunta SQL que se haya realizado. En párrafos anteriores se han comentado los métodos que proporciona este objeto para recorrer dicha tabla.

5.- Transacciones

En párrafos anteriores se ha tratado de la creación y uso de sentencias SQL, que siempre se obtenían llamando a un método de un objeto de tipo **Connection**, como *createStatement()* o *prepareStatement()*. El uso de transacciones, también se controla mediante métodos del objeto **Connection**. Como ya se ha dicho, **Connection** representa una conexión a una Base de datos dada, luego representa el lugar adecuado para el manejo de transacciones, dado que estas afectan a todas las sentencias ejecutadas sobre una conexión a la base de datos.

Por defecto, una conexión funciona en modo *autocommit*, es decir, cada vez que se ejecuta una sentencia SQL se abre y se cierra automáticamente una transacción, que sólo afecta a dicha sentencia. Es posible modificar esta opción mediante *setAutoCommit()*, mientras que *getAutoCommit()* indica si se está en modo *autocommit* o no. Si no se está trabajando en modo *autocommit* será necesario que se cierren explícitamente las transacciones mediante *commit()* si tienen éxito, o *rollback()*, si fallan; nótese que, tras cerrar una transacción, la próxima vez que se ejecute una sentencia SQL se abrirá automáticamente una nueva, por lo que no existe ningún método del tipo que permita iniciar una transacción.

Es posible también especificar el nivel de aislamiento de una transacción, mediante *setTransactionIsolation()*, así como averiguar cuál es el nivel de aislamiento de la actual mediante *getTransactionIsolation()*. Los niveles de aislamiento se representan mediante las constantes que se muestran en la lista siguiente, en la cual se explica muy básicamente el efecto de cada nivel de aislamiento.

TRANSACTION_NONE

No se pueden utilizar transacciones.

TRANSACTION_READ_UNCOMMITTED

Desde esta transacción se pueden llegar a ver registros que han sido modificados por otra transacción, pero no guardados, por lo que podemos llegar a trabajar con valores que nunca llegan a guardarse realmente.

TRANSACTION_READ_COMMITTED

Se ven solo las modificaciones ya guardadas hechas por otras transacciones.

TRANSACTION_REPEATABLE_READ

Si se leyó un registro, y otra transacción lo modifica, guardándolo, y lo volvemos a leer, seguiremos viendo la información que había cuando lo leímos por primera vez. Esto proporciona un nivel de consistencia mayor que los niveles de aislamiento anteriores.

TRANSACTION_SERIALIZABLE

Se verán todos los registros tal y como estaban antes de comenzar la transacción, no importa las modificaciones que otras transacciones hagan, ni que lo hayamos leído antes o no. Si se añadió algún nuevo registro, tampoco se verá.

Además de manejar transacciones, el objeto **Connection** también proporciona algunos otros métodos que permiten especificar características de una conexión a una base de datos; por ejemplo, los métodos *isReadOnly()* y *setReadOnly()* permiten averiguar si una conexión a una base de datos es de sólo lectura, o hacerla de sólo lectura. El método *isClosed()* permite averiguar si una conexión está cerrada o no, y *nativeSQL()* permite obtener la cadena SQL que el driver mandaría a la base de datos si se tratase de ejecutar la cadena SQL especificada, permitiendo averiguar qué es exactamente lo que se le envía a la base de datos.

Información de la Base de Datos

Falta aún una pieza importante a la hora de trabajar con la conexión a la base de datos mediante **Connection**, y es la posibilidad de poder interrogar sobre las características de una base de datos; por ejemplo, puede se interesante saber si la base de datos soporta cierto nivel de aislamiento en una transacción, como la TRANSACTION_SERIALIZABLE, que muchos gestores no soportan. Para esto está otro de los interfaces que proporciona JDBC, **DatabaseMetaData**, al que es posible interrogar sobre las características de la base de datos con la que se está trabajando. Es posible obtener un objeto de tipo **DatabaseMetaData** mediante el método *getMetaData()* de **Connection**.

DatabaseMetaData proporciona diversa información sobre una base de datos, y cuenta con varias docenas de métodos, a través de los cuales es posible obtener gran cantidad de información acerca de una tabla; por ejemplo, *getColumns()* devuelve las columnas de una tabla, *getPrimaryKeys()* devuelve la lista de columnas que forman la clave primaria, *getIndexInfo()* devuelve información acerca de sus índices, mientras que *getExportedKeys()* devuelve la lista de todas las claves ajenas que utilizan la clave primaria de esta tabla, y *getImportedKeys()* las claves ajenas existentes en la tabla. El método *getTables()* devuelve la lista de todas las tablas en la base de datos, mientras que *getProcedures()* devuelve la lista de procedimientos almacenados. Muchos de los métodos de **DatabaseMetaData** devuelven un objeto de tipo **ResultSet** que contiene la información deseada. El listado que se presenta a continuación, muestra el código necesario para obtener todas las tablas de una base de datos.

```
String nombreTablas = "%";        // Listamos todas las
tablas
String tipos[] = new String[1];   // Listamos sólo tablas
tipos[0] = "TABLE";
DatabaseMetaData dbmd = conexion.getMetaData();
ResultSet tablas = dbmd.getTables
( null,null,nombreTablas,tipos );

boolean seguir = tablas.next();
while( seguir ) {
  // Mostramos sólo el nombre de las tablas, guardado
  // en la columna "TABLE_NAME"
  System.out.println(
    tablas.getString( tablas.findColumn
( "TABLE_NAME" ) ) );
    seguir = tablas.next();
  };
```

Hay todo un grupo de métodos que permiten averiguar si ciertas características están soportadas por la base de datos; entre ellos, destacan *supportsGroupBy()* indica si se soporta el uso de GROUP BY en un SELECT, mientras que *supportsOuterJoins()* indica si se pueden llevar a cabo *outer-joins*. El método *supportsTransactions()*, comentado antes, indica si cierto tipo de transacciones está soportado o no. Otros métodos de utilidad son *getUserName()*, que devuelve el nombre del usuario actual; *getURL()*, que devuelve el URL de la base de datos actual.

DatabaseMetaData proporciona muchos otros métodos que permiten averiguar cosas tales como el máximo número de columnas utilizable en un SELECT, etc. En general, casi cualquier pregunta sobre las capacidades de la base de datos se puede contestar llamando a los distintos métodos del objeto **DatabaseMetaData**, que merece la pena que el lector consulte cuando no sepa si cierta característica está soportada.

Tipos SQL en Java

Muchos de los tipos de datos estándar de SQL ’92, no tienen un equivalente nativo en Java. Para superar esta deficiencia, se deben mapear los tipos de datos SQL en Java, utilizando las clases JDBC para acceder a los tipos de datos SQL. Es necesario saber cómo recuperar adecuadamente tipos de datos Java; como *int*, *long*, o *string*, a partir de sus contrapartidas SQL almacenadas en base de datos. Esto puede ser especialmente importante si se está trabajando con datos numéricos, que necesiten control decimal con precisión, o con fechas SQL, que tienen un formato muy bien definido.

El mapeo de los tipos de datos Java a SQL es realmente sencillo, tal como se muestra en la tabla que acompaña a este párrafo. Observe el lector que los tipos que comienzan por "java" no son tipos básicos, sino clases que tienen métodos para trasladar los datos a formatos utilizables, y son necesarias estas clases porque no hay un tipo de datos básico que mapee directamente su contrapartida SQL. La creación de estas clases debe hacerse siempre que se necesite almacenar un tipo de dato SQL en un programa Java, para poder utilizar directamente el dato desde la base de datos.

Java	SQL
String	VARCHAR
boolean	BIT
byte	TINYINT
short	SMALLINT
int	INTEGER
long	BIGINT
float	REAL
double	DOUBLE
byte[]-byte array: imagenes, sonidos...	VARBINARY (BLOBs)
java.sql.Date	DATE
java.sql.Time	TIME
java.sql.Timestamp	TIMESTAMP
java.math.BigDecimal	NUMERIC

El tipo de dato *byte[]*, es un array de bytes de tamaño variable. Esta estructura de datos guarda datos binarios, que en SQL son VARBINARY y LONG-VARBINARY. Estos tipos se utilizan para almacenar imágenes, ficheros de documentos, y cosas parecidas. Para almacenar y recuperar este tipo de información de la base de datos, se deben utilizarlos métodos para streams que proporciona JDBC: *setBinaryStream()* y *getBinaryStream()*.

La conversión de tipos en el sentido contrario puede no estar tan clara, ya que hay tipos SQL cuya tipo Java correspondiente puede no ser evidente, como VARBINARY, o DECIMAL, etc. La tabla siguiente muestra los tipos Java correspondientes a cada tipo SQL.

SQL	Java
CHAR	String
VARCHAR	String
LONGVARCHAR	String
NUMERIC	java.math.BigDecimal
DECIMAL	java.math.BigDecimal
BIT	boolean
TINYINT	byte
SMALLINT	short
INTEGER	int
BIGINT	long
REAL	float
FLOAT	double
DOUBLE	double
BINARY	byte[]
VARBINARY	byte[]
LONGVARBINARY	byte[]
DATE	java.sql.Date
TIME	java.sql.Time
TIMESTAMP	java.sql.Timestamp

Existe una constante para cada tipo de dato SQL, declarada en **java.sql.Types**; por ejemplo, el tipo al tipo TIMESTAMP le corresponde la constante **java.sql.Types.TIMESTAMP**.

Además, JDBC proporciona clases Java nuevas para representar varios tipos de datos SQL: estas son **java.sql.Date**, **java.sql.Time** y **java.sql.Timestamp**.

6.- Modelos

Modelo Relacional de Objetos

Este modelo intenta fundir la orientación a objetos con el modelo de base de datos relacional. Como muchos de los lenguajes de programación actuales, como Java, son orientados a objetos, una estrecha integración entre los dos podría proporcionar una relativamente sencilla abstracción a los desarrolladores que programan en lenguajes orientados a objetos y que también necesitan *programar* en SQL. Esta integración, además, debería casi eliminar la necesidad de una constante traslación entre las tablas de la base de datos y las estructuras del lenguaje orientado a objetos, que es una tarea muy ardua.

A continuación se muestra un ejemplo muy simple para presentar la base del Modelo. Supóngase que se crea la siguiente Tabla en una base de datos:

apellido	nombre	teléfono	num_empleado
González	José	95 498 1112	00001
Gómez	Pedro	95 498 1113	00012
Pérez	Gonzalo	95 498 1114	00045
López	Alejandro	95 498 1115	00023

Con una relativa facilidad, se puede mapear esta tabla en un objeto Java; que, tal como se muestra en el siguiente trozo de código:

```
class Empleado {
  int Clave;
  String Nombre;
  String Apellido;
  String Telefono;
  int Num_Empleado;

  Clave = Num_Empleado;
  }
```

Para recuperar esta tabla desde la base de datos a Java, simplemente se asignarían las columnas respectivas al objeto **Empleado** que se crearía previamente a la recuperación de cada fila, tal como se muestra a continuación:

```
Empleado objEmpleado = new Empleado();
objEmpleado.Nombre = resultset.getString( "nombre" );
objEmpleado.Apellido = resultset.getString( "apellido" );
objEmpleado.Telefono = resultset.getString( "telefono" );
objEmpleado.Num_Empleado = resultset.getInt( "num_empleado"
);
```

Con una base de datos más grande, incluso con enlaces entre las tablas, el número de problemas se dispara, incluyendo la escalabilidad debida a los múltiples JOINs en el modelo de datos y los enlaces cruzados entre las claves de la tablas. Pero, afortunadamente, ya hay productos disponibles que permiten crear este tipo de puentes entre los modelos relacional y orientado a objetos; es más, hay varias de estas soluciones que están siendo desarrolladas para trabajar específicamente con Java.

Uno de los ejemplos, para *Linux*, de este tipo de herramientas es la base de datos **PostGres**, que es un sistema de base de datos relacional que unse las estructuras clásicas de estos sistemas con los conceptos de programación orientada a objetos, es decir, se trataría de una base de datos objeto-relacional. En PostGres, por ejemplo, las tablas de denominan clases, las filas se denominan instancias y las columnas se denominan atributos. Además, un concepto que aparece en PostGres y que viene claramente de la programación orientada a objetos es la *Herencia*, de forma que cuando se crea una nueva clase heredada de otra, la clase creada adquiere todas las características de la clase de la que proviene, más las características que se definan en la nueva clase. Por poner un ejemplo, si se tiene una tabla creada con la sentencia que se indica a continuación:

```
CREATE TABLA tabla1 (
  campo1 text
  campo2 int )
```

A continuación, se puede crear una segunda tabla con la sentencia SQL a continuación definida:

```
CREATE TABLA tabla2 (
  campo3 int )
INHERITS( tabla1 )
```

Como resultado de esta sentencia SQL, la nueva tabla tendrá los atributos *campo1*, *campo2* y *campo3*.

Modelo de Conexión

La conexión a bases de datos relacionales a través de JDBC realmente es muy simple, tal como ya se ha podido comprobar. No obstante, en este apartado es proporcionar, o intentarlo al menos, una plantilla que se pueda reutilizar y personalizar para aprender a manipular bases de datos con Java. Una vez que el ejemplo sea comprendido por el lector, no habrá problemas a la hora de saber qué hacer y cómo hacerlo. Y cuando se necesite más información, la documentación que proporciona JavaSoft sobre JDBC la tiene.

De todas las cosas que hay que tener en mente, la conexión de Java con la base de datos relacional es la primera de las preocupaciones. En el ejemplo java2102.java se muestra cómo se hace, y establece la conexión. También contiene varios métodos que procesan sentencias SQL habituales de una forma simple y segura: la conexiones son abiertas y cerradas con cada sentencia SQL. Si el lector está construyendo aplicaciones en las que se prevean grandes flujos de transacciones, tendrá que establecer una estrategia más adecuada; por ejemplo, si se pretenden realizar actualizaciones sobre un registro una vez que se haya accedido a él, probablemente sea mejor mantener abierta la conexión con la base de datos hasta que hayan concluido todas las actualizaciones. Como la conexión a una base de datos es un objeto, se puede mantener en una variable tanto tiempo como se necesite; con lo cual, la aplicación será capaz de procesar las actualizaciones mucho más rápidamente, pero corriendo el riesgo de que otros usuarios tengan bloqueado el acceso hasta que las conexiones que estén bloqueadas se concluyan.

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

public class java2102 extends Thread {
  public static final int PUERTO = 6700;
  ServerSocket socketEscucha;

  public java2102() {
    try {
      socketEscucha = new ServerSocket( PUERTO );
    } catch( IOException e ) {
      System.err.println( e );
    }
    this.start();
  }

  public void run() {
    try {
      while( true ) {
        Socket socketCliente = socketEscucha.accept();
        SQLConexion c = new SQLConexion( socketCliente );
```

```
      }
    } catch( IOException e ) {
      System.err.println( e );
    }
  }

  public static void main( String[] argv ) {
    new java2102();
  }
}

class SQLConexion extends Thread {
  protected Socket cliente;
  protected BufferedReader in;
  protected PrintStream out;
  protected String consulta;

  public SQLConexion( Socket socketCliente ) {
    cliente = socketCliente;
    try {
      in =
     new BufferedReader( new InputStreamReader
( cliente.getInputStream() ) );
      out = new PrintStream( cliente.getOutputStream() );
    } catch( IOException e ) {
      System.err.println( e );
      try {
        cliente.close();
      } catch( IOException e2 ) {};
         return;
    }
    this.start();
  }

  public void run() {
    try {
      consulta = in.readLine();
      System.out.println( "Lee la consulta <" + consulta +
">" );
      ejecutaSQL();
    } catch( IOException e ) {}
    finally {
      try {
        cliente.close();
      } catch( IOException e ) {};
    }
  }

  public void ejecutaSQL() {
    Connection conexion;   // Objeto de conexión a la base
de datos
    Statement sentencia;   // Objeto con la sentencia SQL
    ResultSet resultado;   // Objeto con el resultado de la
consulta SQL
```

```
    ResultSetMetaData resultadoMeta;
    boolean mas;           // Indicador de si hay más filas
    String driver = "jdbc:odbc:Tutorial";
    String usuario = "";
    String clave = "";
    String registro;
    int numCols, i;

    try {
      Class.forName( "sun.jdbc.odbc.JdbcOdbcDriver" );
      conexion = DriverManager.getConnection
( driver,usuario,clave );

      sentencia = conexion.createStatement();
      resultado = sentencia.executeQuery( consulta );

      mas = resultado.next();
      if( !mas ) {
        out.println( "No hay mas filas." );
        return;
      }

      resultadoMeta = resultado.getMetaData();
      numCols = resultadoMeta.getColumnCount();
      System.out.println( numCols + " columnas en el
resultado.");
      while( mas ) {
        // Se construye la cadena de respuesta
        registro = "";
        for( i=1; i <= numCols; i++ ) {
          registro = registro.concat( resultado.getString
(i)+"  " );
        }
        out.println( registro );
        System.out.println( registro );
        mas = resultado.next();
      }

      resultado.close();
      sentencia.close();
      conexion.commit();
      conexion.close();
    } catch( Exception e ) {
      System.out.println( e.toString() );
    }
  }
}
```

El ejemplo, evidentemente, asume que hay una base de datos disponible para usar y su esquema es muy simple, ya que utiliza la base de datos del primer ejemplo del capítulo, con solamente tres campos. Mucho del desarrollo de este capítulo ha sido desarrollado en *Linux* utilizando la base de datos *PostGres*, y luego, exactamente el

mismo código, solamente cambiando la conexión a la base de datos, ejecutado utilizando *Microsoft Acc*ess, para capturar la ejecución con el API del último JDK.

La parte cliente del ejemplo anterior, es la que se ha codificado en el ejemplo java2103.java, implementado como applet, que permite introducir una consulta en el campo de texto, que será enviada al servidor implementado en el ejemplo anterior, que a su vez, enviará la consulta a la base de datos y devolverá el resultado al applet, que mostrará la información resultante de su consulta en la parte inferior del applet. En estos dos ejemplos, se muestran los fundamentos de la interacción con bases de datos, de una forma un poco más complicada, de tal modo que no desde el mismo programa se ataca a la base de datos; esto proporcionará al lector una visión más amplia de la capacidad y potencia que se encuentra bajo la conjunción de Java y las Bases de Datos.

```
import java.io.*;
import java.net.*;
import java.applet.*;
import java.awt.*;
import java.awt.event.*;

public class java2103 extends Applet {
  static final int puerto = 6700;
  String cadConsulta = "No hay consulta todavia";
  String cadResultado = "No hay resultados";
  Button boton;
  TextArea texto;
  List lista;

  public void init() {
    setLayout( new GridLayout( 5,1 ) );
    texto = new TextArea( 20,40 );
    lista = new List();
    boton = new Button( "Ejecutar Consulta" );
    boton.addActionListener( new MiActionListener() );

    add( new Label( "Escribir la consulta aqui..." ) );
    add( texto );
    add( boton );
    add( new Label( "y examinar los resultados aqui" ) );
    add( lista );

    resize( 800,800 );
    }

  void abreSocket() {
    Socket s = null;
    try {
      s = new Socket( getCodeBase().getHost(),puerto );
```

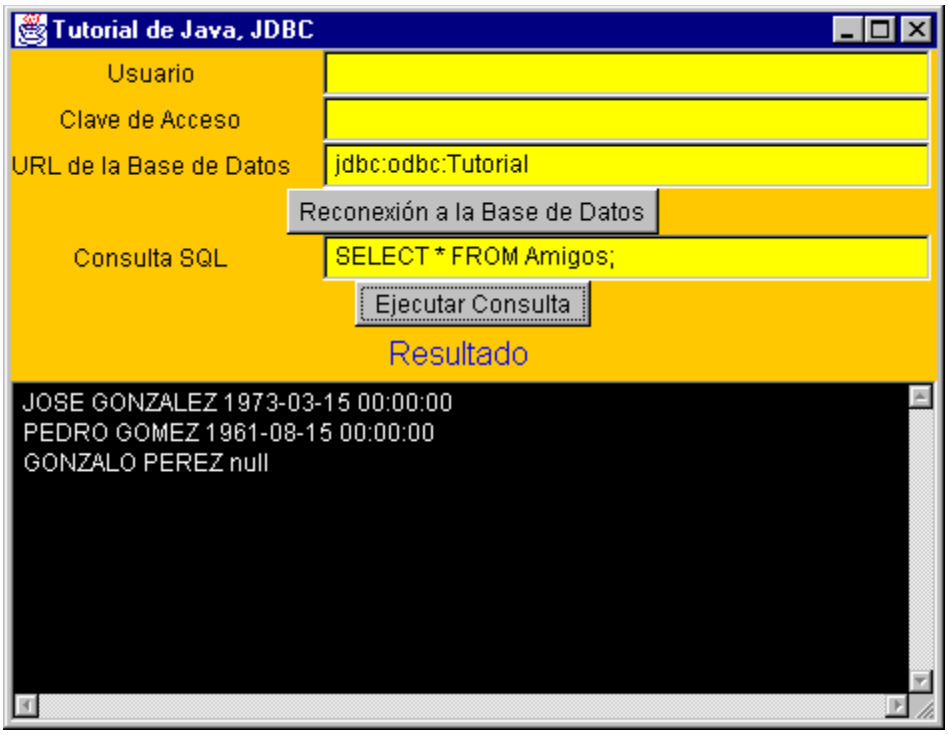
```
      BufferedReader sinstream =
        new BufferedReader(new InputStreamReader
(s.getInputStream()));
      PrintStream soutstream = new PrintStream
( s.getOutputStream() );

      soutstream.println( texto.getText() );
      lista.removeAll();
      cadResultado = sinstream.readLine();
      while( cadResultado != null ) {
        lista.add( cadResultado );
        cadResultado = sinstream.readLine();
        }
    } catch( IOException e ) {
      System.err.println( e );
    } finally {
    try {
      if( s != null )
        s.close();}
    catch( IOException e ) {}
    }
    }

  class MiActionListener implements ActionListener {
    public void actionPerformed( ActionEvent evt ) {
      abreSocket();
    }
  }
}
```

Una vez comprendidas las ideas básicas que se presentaban en los ejemplos anteriores, el lector podrá atreverse ya a escribir programas que manejen esquemas de bases de datos mucho más complicados. No obstante, si el lector tiene una buena base en el conocimiento de bases de datos relacionales, estará satisfecho con esta simplicidad.

El ejemplo java2104.java, es un simple interfaz para atacar a cualquier tipo de base de datos. Solamente se ha codificado el driver JDBC a utilizar, pero a través de la ventana que se presenta, se puede acceder a cualquier base de datos y cualquier tabla. La figura muestra la ventana, una vez que se ha accedido a la base de datos que se ha estado utilizando.



Y el código completo del ejemplo, que se ha intentado comentar profusamente para que la comprensión sea sencilla es el que se reproduce a continuación.

```
import java.net.URL;
import java.awt.*;
import java.sql.*;
import java.awt.event.*;

public class java2104 extends Frame implements
MouseListener {
  // Se utiliza el itnerfaz MouseListener para poder
capturar
  // los piques de ratón

  // Estos son los objetos que se van a utilizar en la
aplicación
  Button botConexion = new Button( "  Conexión a la Base de
Datos  " );
  Button botConsulta = new Button( " Ejecutar Consulta " );

  TextField txfConsulta = new TextField( 40 );
  TextArea txaSalida = new TextArea( 10,75 );
  TextField txfUsuario = new TextField( 40 );
  TextField txfClave = new TextField( 40 );
  TextField txfUrl = new TextField( 40 );
  String strUrl = "";
  String strUsuario = "";
  String strClave = "";
```

```
  // El objeto Connection es parte del API de JDBC, y debe
ser lo
  // primero que se obtenga, ya que representa la conexión
efectiva
  // con la Base de Datos
  Connection con;

  public static void main( String args[] ) {
    java2104 ventana = new java2104();

    // Se recoge el evento de cierre de la ventana
    ventana.addWindowListener( new WindowAdapter() {
      public void windowClosing( WindowEvent evt ) {
        System.exit( 0 );
      }
    } );

    ventana.setSize( 450,300 );
    ventana.setTitle( "Tutorial de Java, JDBC" );
    ventana.pack();
    ventana.setVisible( true );
  }

  // Constructor de la clase, que es el que construye el
interfaz
  // que se va a mostrar en la ventana
  public java2104() {
    // Se hacen todos los campos de texto editables para
que se
    // puedan introducir datos, y no se permite que se
escriba en
    // el área de texto que se va a utilizar como salida de
los
    // resultados de las acciones del usuario y las
respuestas que
    // se obtengan de la base de datos a las consultas que
se
    // realicen
    txfConsulta.setEditable( true );
    txfUsuario.setEditable( true );
    txfUrl.setEditable( true );
    txaSalida.setEditable( false );

    // Se va a utilizar el GridBagLayout, que aunque
complicado
    // en su uso, tiene la flexibilidad que se necesita en
este
    // caso concreto
    GridBagLayout gridbag = new GridBagLayout();
    GridBagConstraints gbCon = new GridBagConstraints();
    // Lo fijamos como el layout a utilizar
    setLayout( gridbag );

    // Se fija el color y la fuente de caracteres a usar
```

```
    setFont( new Font( "Helvetica",Font.PLAIN,12 ) );
    setBackground( Color.orange );

    // No se han fijado los setConstraints para el Label,
para que
    // se asuman los de defecto. El campo de texto
txfUsuario es
    // el último componente en su fila, a través de gbCon,
y
    // luego se añade al interfaz de usuario
    gbCon.weightx = 1.0;
    gbCon.weighty = 0.0;
    gbCon.anchor = GridBagConstraints.CENTER;
    gbCon.fill = GridBagConstraints.NONE;
    gbCon.gridwidth = GridBagConstraints.REMAINDER;
    add( new Label( "Usuario" ) );
    gridbag.setConstraints( txfUsuario,gbCon );
    add( txfUsuario );
    add( new Label( "Clave de Acceso" ) );
    gridbag.setConstraints( txfClave,gbCon );
    add( txfClave );
    add( new Label( "URL de la Base de Datos" ) );
    gridbag.setConstraints( txfUrl,gbCon );
    add( txfUrl );
    // Ahora viene la fila en que está el botón de Conexión
a la
    // base de datos, fijamos los constraints para que eso
sea así
    // y lo añadimos
    gridbag.setConstraints( botConexion,gbCon );
    add( botConexion );

    // Ahora registramos el botón para que reciba los
eventos del
    // raton a través del interfaz MouseListener
    botConexion.addMouseListener( this );

    // Ahora viene la zona que permite introducir el texto
de la
    // consulta que se quiere realizar y el botón que va a
permitir
    // su envío al driver JDBC
    add( new Label( "Consulta SQL" ) );
    gridbag.setConstraints( txfConsulta,gbCon );
    add( txfConsulta );
    gridbag.setConstraints( botConsulta,gbCon );
    add( botConsulta );
    botConsulta.addMouseListener( this );

    // Ahora se coloca una etiqueta en su propia línea para
rotular
    // el área de texto en la que se van a presentar los
resultados
    // de las consultas que se realicen
```

```
    Label labResultado = new Label( "Resultado" );
    labResultado.setFont( new Font
( "Helvetica",Font.PLAIN,16 ) );
    labResultado.setForeground( Color.blue );
    gridbag.setConstraints( labResultado,gbCon );
    gbCon.weighty = 1.0;
    add( labResultado );

    // Ahora se cambia la forma de extensión de la ventana,
para que
    // si se agranda la ventana tenga la mayor parte de
espacio
    // posible en la zona de texto en donde se presentan
los
    // resultados
    gridbag.setConstraints( txaSalida,gbCon );
    txaSalida.setForeground( Color.white );
    txaSalida.setBackground( Color.black );
    add( txaSalida );
    }

  public void mouseClicked( MouseEvent evt ) {
    // Cuando se pulsa el botón Consulta, se recoge el
contenido del
    // campo de texto txfConsulta y se le pasa al método
Select, que
    // es el que va a realizar la consulta y devolver el
resultado
    // que se va a presentar en la zona de salida
    if( evt.getComponent() == botConsulta ) {
      System.out.println( txfConsulta.getText() );
      txaSalida.setText( Select( txfConsulta.getText() ) );
    }

    // Si se pulsa el botón de Conexión, se intenta
establecer la
    // conexión con la base de datos indicada en el campo
de texto
    // correspondiente a URL, con el usuario y clave que se
hayan
    // indicado en los campos correspondientes
    if( evt.getComponent() == botConexion ) {
      // Se fijan las variables globales de usaurio, clave
y url a
      // los valores que se hayan introducido en los campos
      strUsuario = txfUsuario.getText();
      strClave = txfClave.getText();
      strUrl = txfUrl.getText();

      // La creación de la conexión con la base de datos
lanza una
      // excepción en caso de que haya problemas al
establecer esa
```

```
      // conexión con los aprámetros que se le indiquen,
por ello
      // es imprescindible colocar el método getConnection
en un
      // bloque try-catch. Si se produce algún problema y
se lanza
      // la excepción, aparecerá reflejada en la consola y
en el
      // área que se ha destinado en la ventana a ver los
resultados
      try {
        // Ahora se intenta crear una nueva instancia del
driver que se
        // va a utilizar. Hay varias formas de especificar
el driver que
        // se quiere, e incluso se puede dejar que sea el
propio
        // DriverManager de JDBC que seleccione el que
considere más
        // adecuado para conectarse a una fuente de datos
determinada
        Class.forName( "sun.jdbc.odbc.JdbcOdbcDriver" );
        // La conexión aquí se realiza indicando la URL de
la base de
        // datos y el usuario y clave que dan acceso a ella
        con = DriverManager.getConnection
( strUrl,strUsuario,strClave );
        // Si la conexión ha sido satisfactoria, cambiamos
el rótulo
        // del botón de conexión, para que indique que si
se pulsa lo
        // que se realiza será la "Reconexión"
        botConexion.setLabel( "Reconexión a la Base de
Datos" );
        txaSalida.setText( "Conexion establecida con
"+strUrl );
      } catch( Exception e ) {
        // Se presenta la información correspondiente al
error, tanto
        // en la consola como en la zona de salida de la
ventana
        e.printStackTrace();
        txaSalida.setText( e.getMessage() );
      }
    }
  }

  // Se implementan vacíos el resto de los métodos del
interfaz de
  // eventos del ratón, MouseListener. Si se quiere evitar,
también es
  // posible utilizar MouseAdapter, que tiene
implementados, pero sin
  // acciones asignadas, todos estos métodos
```

```
  public void mouseEntered( MouseEvent evt ) {}
  public void mouseExited( MouseEvent evt ) {}
  public void mousePressed( MouseEvent evt ) {}
  public void mouseReleased( MouseEvent evt ) {}

  // Este es el método que realiza la consulta
  public String Select( String consulta ) {
    String resultado="";
    int cols;
    int pos;

    // Hay varios métos que se van a emplear y que lanzan
excepciones
    // en caso de que haya algún problema con la consulta,
o si se
    // rompe la conexión, etc
    try {
      // En primer lugar, se instancia la clase Statement,
que es
      // necesaria para ejecutar la consulta. La clase
Connection
      // devuelve un objeto Statement que se enlaza a la
conexión
      // abierta para pasar de nuevo el objeto Statement.
Así es
      // como la instancia "sentencia" se enlaza a la
conexión actual
      // con la base de datos
      Statement sentencia = con.createStatement();

      // El objeto resultSet también es enlazado con la
conexión a la
      // base de datos a través de la clase Statement, que
contiene el
      // método executeQuery, que devuelve un objeto
ResultSet.
      ResultSet rs = sentencia.executeQuery( consulta );

      // Ahora se utiliza el método getMetaData en el
resultado para
      // devolver un objeto MetaData, que contiene el
método getColumnCount
      // usado para determinar cuántas columnas de datos
están presentes
      // en el resultado.
      cols = ( rs.getMetaData() ).getColumnCount();

      // Aquí se utiliza el método next de la instancia
"rs" de
      // ResultSet para recorrer todas las filas, una a
una. Hay formas
      // más optimizadas de hacer esto, utilizando la
característica
      // inputStream del driver JDBC
```

```
      while( rs.next() ) {
        // Se recorre ahora cada una de las columnas de la
fila, es
        // decir, cada celda, una a una
        for( pos=1; pos <= cols; pos++ ) {
          // Este es el método general para obetener un
resultado. el
          // método getString intentará moldear el
resultado a un String.
          // En este caso solamente se recoge el resultado
y se le añade
          // un espacio y todo se añade a la variable
"resultado"
          resultado += rs.getString( pos )+" ";
        }

        // Para cada fila que se revise, se le añade un
retorno de
        // carro, para que la siguiente fila empiece en
otra línea
        resultado += "\n";
      }

      // Se cierra la "sentencia". En realidad se cierran
todos los
      // canales abiertos para la consulta pero la conexión
con la
      // base de datos permanece
      sentencia.close();
    } catch( Exception e ) {
      e.printStackTrace();
      resultado = e.getMessage();
    }
  // Antes de salir, se devuelve el resultado obtenido
  return resultado;
  }
}
```

7.- Servlets y JDBC

En esta parte del capítulo dedicado a JDBC, se va a explorar el mundo de los *servlets* en el contexto de JDBC. Si el lector no está cómodo con el uso de *servlets* o con los conocimientos que posee, en el Tutorial se trata a fondo este tema, al que puede recurrir y luego regresar a este punto de su lectura.

Hasta ahora se ha presentado la parte cliente del uso de JDBC, y en lo que se pretende adentrar al lector es en el uso de JDBC en el servidor para generar páginas Web a partir de la información obtenida de una base de datos. Si el lector está familiarizado con CGI, podrá comprobar que los *servlets* son mucho mejores a la hora de generar páginas Web dinámicamente, por varias razones: velocidad, eficiencia en el uso de recursos, escalabilidad, etc.

La arquitectura de los servlets hace que la escritura de aplicaciones que se ejecuten en el servidor sea relativamente sencilla y, eso sí, sean aplicaciones muy robustas. La principal ventaja de utilizar servlets es que se puede programar sin dificultad la información que va a proporcionar entre peticiones del cliente. Es decir, se puede tener constancia de lo que el usuario ha hecho en peticiones anteriores e implementar funciones de tipo *rollback* o *cancel transaction* (suponiendo que el servidor de base de datos las soporte). Además, cada instancia del servlet se ejecuta dentro de un hilo de ejecución Java, por lo que se pueden controlar las interacciones entre múltiples instancias; y al utilizar el identificador de sincronización, se puede asegurar que los servlets del mismo tipo esperan a que se produzca la misma transacción, antes de procesar la petición; esto puede ser especialmente útil cuando mucha gente intenta actualizar al mismo tiempo la base de datos, o si hay mucha gente pendiente de consultas a la base de datos cuando ésta está en pleno proceso de actualización.

El ejemplo java2105.java, es un servlet, que junto con la página web asociada, java2105.html, y las dos bases de datos que utiliza, conforma un servicio completo de noticias o artículos. Hay un número determinado de usuarios que tienen autorización para enviar artículos, y otro grupo de usuarios que pueden ver los últimos artículos. Por supuesto, que también se podría almacenar la fecha y la hora en que se colocan los artículos, y posiblemente también fuese

útil la categorización, es decir, colocarlos dentro de una categoría como deportes, internacional, nacional, local, etc. De este modo, lo que aquí se esboza como la simple gestión de artículos, podría ser la semilla de un verdadero *servicio de noticias*. Pero el autor no pretende llegar a eso, sino simplemente presentar la forma en que se puede aunar la fuerza de JDBC y los servlets para poder acceder a una base de datos, introduciendo algunas características como la comprobación de autorización, etc.; por ello, se ha huido conscientemente del uso de la palabra noticias, aunque el lector puede implementar sin demasiada dificultad.

Como el lector es una persona lista, seguro que almacenará toda esta información en una base de datos de la forma mejor posible; de forma que se podría escribir un servlet que acumulase la información que le llegue en ficheros para luego proporcionarla, pero el manejo de todos estos archivos puede resultar engorrosa, y las búsquedas enlentecer el funcionamiento; así que, una base de datos es la mejor de las soluciones.

Además, también se quiere almacenar las contraseñas para el acceso al sistema de artículos en una base de datos, en vez de en la configuración del servidor Web. Hay dos razones fundamentales para ello; por una lado, porque es previsible que mucha gente utilice este servicio, y por otro lado, que debe ser posible asociar cada envío con una persona en particular. La inclusión en una base de datos ayudará a manejar gran cantidad de usuarios y además a controlar quien envía artículos. Una mejora que se puede hacer al sistema es el desarrollo de un applet JDBC que permita añadir y quitar usuarios, o asignar privilegios a quien haya enviado algún artículo al Sistema.

Las siguientes líneas de código muestran las sentencias utilizadas en la creación de las tablas e índices que se van a utilizar en la aplicación que se desarrollará para completar el Sistema de Artículos. El programa se ha pensado para atacar una base de datos *Access*, utilizando el puente JDBC-ODBC, si el lector desea portarlo a otro driver JDBC, tendría que asegurarse de que las sentencias SQL que se utilizan están soportadas por él.

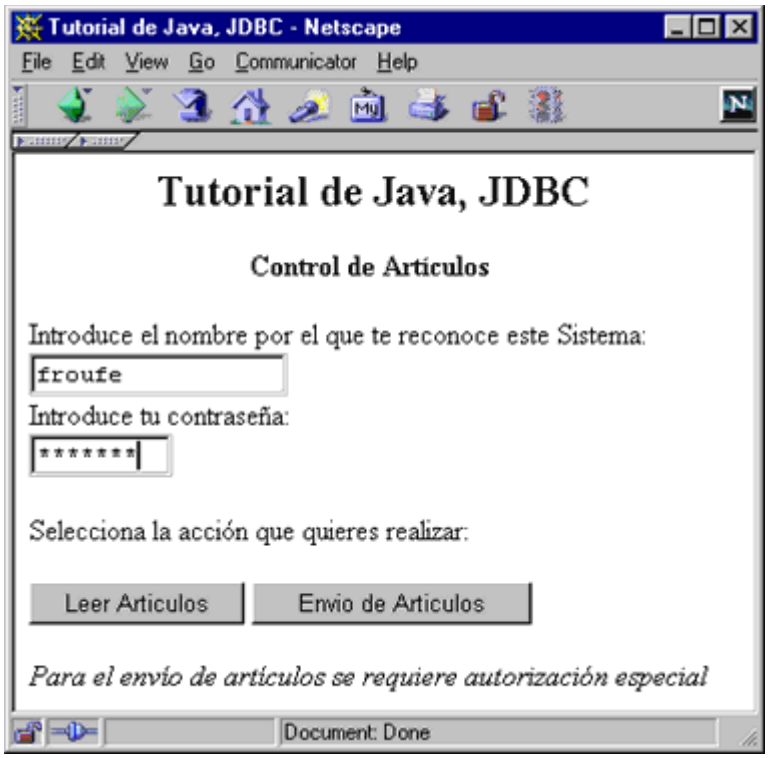
```
CREATE TABLE usuarios (
  usuario VARCHAR(16) NOT NULL,
  nombre VARCHAR (60) NOT NULL,
  empresa VARCHAR (60) NOT NULL,
  admitirEnvio CHAR(1) NOT NULL,
  clave VARCHAR (8) NOT NULL );
CREATE INDEX usuario_key ON usuarios(usuario) WITH PRIMARY;
```

```
CREATE TABLE articulos (
  titulo VARCHAR(255) NOT NULL,
  usuario VARCHAR(16) NOT NULL,
  cuerpo memo );
CREATE INDEX articulo_key ON articulos(usuario);
CREATE INDEX titulo_key ON articulos(titulo,usuario);
```

Como se puede observar, solamente hay dos tablas. Si se quisiesen implementar las categorías, sería necesario incorporar una nueva tabla, o añadir un campo más a la tabla de artículos. La clave primaria de la tabla de usuarios es el identificador de cada usuario, y en la tabla de artículos hay un índice compuesto formado por el título del artículo y el usuario que lo envió. Se usa un tipo MEMO (soportado pro Access, que puede ser BLOB en otros motores), para guardar el cuerpo del artículo. Una mejora, en caso de convertirlo en un sistema de noticias, consistiría en añadir la fecha en que se ha enviado la noticia, como ya se ha comentado en un párrafo anterior, guardarla en un campo en la tabla de artículos y añadir el campo al índice compuesto de esa tabla, con lo cual se podrían presentar las noticias correspondientes a una fecha determinada sin aparente dificultad.

En la tabla de usuarios se guarda el identificador por el que el sistema va a reconocer al usuario, junto con la contraseña que elija para comprobar su identificación, más su nombre completo, la empresa a que pertenece y si ese usuario tiene autorización para el envío de artículos al Sistema. Si un usuario no dispone de autorización para el envío, solamente podrá acceder a la lectura de artículos. Y por supuesto, si alguien no aparece en esta tabla, no tendrá acceso alguno al Sistema.

Lo primero que hay que desarrollar es la página que va a dar acceso al sistema, para ver la forma de procesar los datos que van a llegar. La página es muy simple, java2105.html, y a continuación se reproduce la imagen que presenta en el navegador y el código *html* utilizado para generarla.

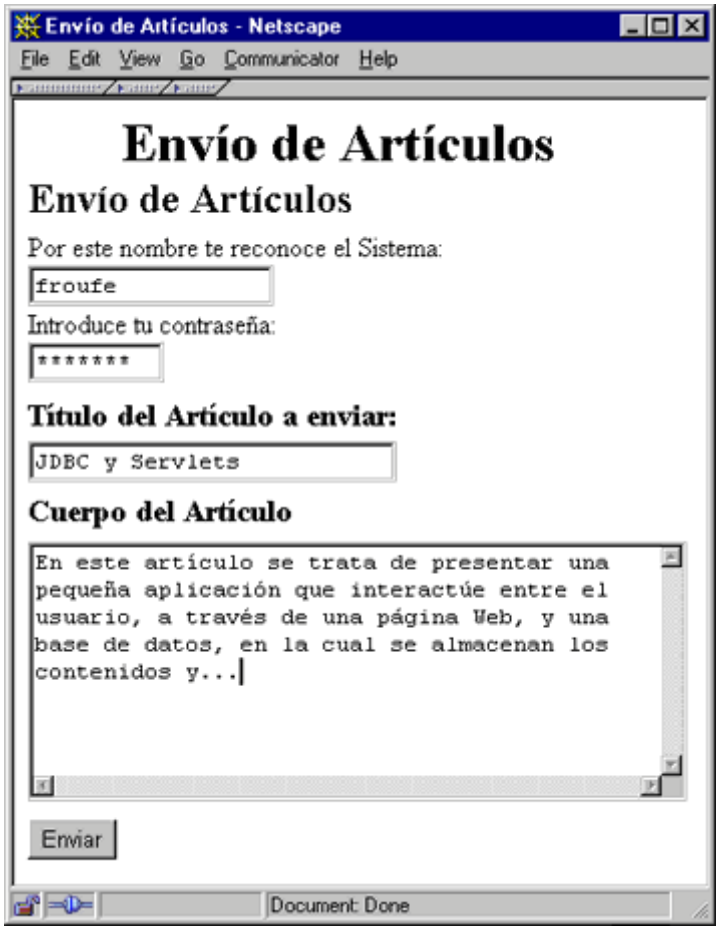



```
<HTML>
<HEAD>
  <TITLE>Tutorial de Java, JDBC</TITLE>
</HEAD>
<BODY>
<FORM ACTION="http://breogan:8080/servlet/java2105"
  ENCTYPE="x-www-form-encoded" METHOD="POST">
  <H2><CENTER>Tutorial de Java, JDBC</CENTER></H2>
  <P><CENTER><B>Control de Artículos</B></CENTER></P>

  <P>Introduce el nombre por el que te reconoce este
Sistema:<BR>
  <INPUT NAME="usuario" TYPE="text" SIZE="16"
MAXLENGTH="16"><BR>
  Introduce tu contraseña:<BR>
  <INPUT NAME="clave" TYPE="password" SIZE="8"
MAXLENGTH="8"></P>
  Selecciona la acción que quieres realizar:
  <P>
  <INPUT NAME="accion" TYPE="submit" VALUE="Leer
Articulos">
  <INPUT NAME="accion" TYPE="submit" VALUE="Envio de
Articulos">
  </P>
  </FORM>
<I>Para el envío de artículos se requiere autorización
especial</I>
</BODY>
</HTML>
```

Lo primero que se necesita aclarar es cómo se van a procesar los datos del formulario. La página no puede ser más sencilla, tal como se puede ver en la captura de su visualización en el navegador, con dos botones para seleccionar la acción a realizar. Si se quiere enviar un artículo, no es necesario introducir el nombre y clave en esta página, ya que el servlet enviará una nueva página para la introducción del contenido del artículo, y ya sobre esa sí que se establecen las comprobaciones de si el usuario está autorizado o no a enviar artículos al sistema.

La página que envía el servlet, es la que reproduce la imagen siguiente. Esta página se genera en el mismo momento en que el usuario solicita la inserción de un artículo. En caso de que haya introducido su identificación en la pagina anterior, el servlet la colocará en su lugar, sino, la dejará en blanco.



La imagen reproduce la página ya rellena, lista para la inserción de un nuevo artículo en el sistema. Cuando se pulsa el botón de envío de artículo y el servlet recibe la petición de inserción del artículo en el sistema, es cuando realiza la comprobación de autorización, por una

lado de si es un usuario reconocido para el sistema y, en caso afirmativo, si está autorizado al envío de artículo, o solamente puede leerlos.

Debe recordar el lector, que los nombres de los campos de entrada de datos se pueden llamar con *getParameter()* y recoger la información que contienen. Los parámetros *nombre* y *clave* en el formulario se mapean en las variables *usuario* y *clave* en el servlet, que serán las que se utilicen para realizar las comprobaciones de acceso del usuario.

El código completo del servlet está en el fichero java2105.java, que se reproduce a continuación.

```
import java.io.*;
import java.net.*;
import java.sql.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class java2105 extends HttpServlet {
  String DBurl = "jdbc:odbc:Tutorial";
  String usuarioGet = "";
  String usuarioPost = "";
  String claveGet = "";
  String clavePost = "";
  Connection con;
  DatabaseMetaData metaData;

  // Este método es el que se encarga de establecer e
inicializar
  // la conexión con la base de datos
  public void init( ServletConfig conf ) throws
ServletException {
    SQLWarning w;

    super.init( conf );
    try {
      Class.forName( "sun.jdbc.odbc.JdbcOdbcDriver" );
      con = DriverManager.getConnection
( DBurl,usuarioPost,clavePost );
      if( (w = con.getWarnings()) != null ) {
        while( w != null ) {
          log( "SQLWarning: "+w.getSQLState()+'\t'+
            w.getMessage()+'\t'+w.getErrorCode()+'\t' );
            w = w.getNextWarning();
          }
        }
    } catch( ClassNotFoundException e ) {
      throw new ServletException( "init" );
```

```
    } catch( SQLException e ) {
      try {
        con = DriverManager.getConnection
( DBurl,usuarioGet,claveGet );
      } catch( SQLException ee ) {
        ee.printStackTrace();
        while( e != null ) {
          log( "SQLException: "+e.getSQLState()+'\t'+
            e.getMessage()+'\t'+e.getErrorCode() );
          e = e.getNextException();
          }
        throw new ServletException( "init" );
      }
    }
  }

  public void service( HttpServletRequest
req,HttpServletResponse res )
    throws ServletException,IOException {
    String usuario = req.getParameter( "usuario" );
    String autorizado;
    String accion = req.getParameter( "accion" );

    try {
      autorizado = autorizacion( req );
      if( accion.equals( "Leer Articulos" )
        && !autorizado.equals( "ACCESO DENEGADO" ) ) {
        leerArticulo( req,res );
      } else if( accion.equals( "Enviar" )
        && autorizado.equals( "POST" ) ) {
        enviarArticulo( req,res );
      } else if( accion.equals("Envio de Articulos" ) ) {
        if( usuario == null )
          usuario = " ";
        PrintWriter out = new PrintWriter
( res.getOutputStream() );
        out.println( "<HTML>" );
        out.println( "<HEAD><TITLE>Envío de
Artículos</TITLE></HEAD>" );
        out.println( "<BODY>" );
        out.println( "<CENTER><H1>Envío de
Artículos</H2></CENTER>" );
        out.println( "<FORM
ACTION=http://breogan:8080/servlet/java2105" );
        out.println( "METHOD=POST>" );
        out.println( "<H2>Envío de Artículos</H2>" );
        out.println( "<P>Por este nombre te reconoce el
Sistema: <BR>" );
        out.println( "<INPUT NAME=usuario TYPE=text
VALUE='"+usuario+"'" );
        out.println( "SIZE=16 MAXLENGTH=16><BR>" );
        out.println( "Introduce tu contraseña: <BR>" );
        out.println( "<INPUT NAME=clave TYPE=password
SIZE=8 MAXLENGTH=8>" );
```

```
        out.println( "</P>" );
        out.println( "<H3>Título del Artículo a
enviar:</H3>" );
        out.println( "<P><INPUT NAME=titulo TYPE=text
SIZE=25" );
        out.println( "MAXLENGTH=50></P>" );
        out.println( "<H3>Cuerpo del Artículo</H3>" );
        out.println( "<P><TEXTAREA NAME=cuerpo ROWS=10
COLS=50>" );
        out.println( "</TEXTAREA></P>" );
        out.println( "<P><INPUT NAME=accion TYPE=submit
VALUE='Enviar'>" );
        out.println( "</FORM>" );
        out.println( "</BODY></HTML>" );
        out.flush();
      } else {
        PrintWriter out = new PrintWriter
( res.getOutputStream() );
        out.println( "<html>" );
        out.println( "<head><title>Acceso
Denegado</title></head>" );
        out.println( "<body>" );
        out.println( "Se ha producido un error de
acceso:<br>" );
        out.println( "El usuario o clave que has
introducido no " );
        out.println( "son válidos o<br>" );
        out.println( "no tienes acceso a esta
funcionalidad." );
        out.println( "</body></html>" );
        out.flush();
      }
    } catch( SQLException e ) {
      while( e != null ) {
        log( "SQLException: "+e.getSQLState()+'\t'+
          e.getMessage()+'\t'+e.getErrorCode()+'\t' );
        e = e.getNextException();
        }
    // Aquí habría que insertar el código necesario para
restablecer la
    // conexión llamando a init() de nuevo y volviendo a
realizar la
    // llamada al método service(req,res)
    }
  }

  // Se cierra la conexión con la base de datos
  public void destroy() {
    try {
      con.close();
    } catch( SQLException e ) {
      while( e != null ) {
        log( "SQLException: "+e.getSQLState()+'\t'+
          e.getMessage()+'\t'+e.getErrorCode()+'\t' );
```

```
        e = e.getNextException();
        }
    } catch( Exception e ) {
      e.printStackTrace();
    }
  }

  // Este método ejecuta la consulata a la base de datos y
devuelve el
  // resultado, para formatear la salida y presentar el
resultado de
  // la consulta de artículos al usuario
  public void leerArticulo( HttpServletRequest
req,HttpServletResponse res )
    throws IOException,SQLException {
    Statement stmt = con.createStatement();
    String consulta;
    ResultSet rs;

    res.setStatus( res.SC_OK );
    res.setContentType( "text/html" );
    consulta = "SELECT
articulos.cuerpo,articulos.titulo," );
    consulta +=
articulos.usuario,usuarios.nombre,usuarios.empresa ";
    consulta += "FROM articulos,usuarios WHERE " );
    consulta += articulos.usuario=usuarios.usuario";
    rs = stmt.executeQuery( consulta );

    PrintWriter out = new PrintWriter( res.getOutputStream
() );
    out.println( "<HTML>" );
    out.println( "<HEAD><TITLE>Artículos
Enviados</TITLE></HEAD>" );
    out.println( "<BODY>" );

    while( rs.next() ) {
      out.println( "<H2>" );
      out.println( rs.getString(1) );
      out.println( "</H2><p>" );
      out.println( "<I>Enviado desde: "+rs.getString(5)
+"</I><BR> " );
      out.println( "<B>"+rs.getString(2)+"</B>, por
"+rs.getString(4) );
      out.println( "<HR>" );
      }
    out.println( "</BODY></HTML>" );
    out.flush();
    rs.close();
    stmt.close();
    }

  public void enviarArticulo( HttpServletRequest
req,HttpServletResponse res )
```

```
    throws IOException,SQLException {
    Statement stmt = con.createStatement();
    String consulta = "";
    String usuario = req.getParameter( "usuario" );

    PrintWriter out = new PrintWriter( res.getOutputStream
() );
    res.setStatus( res.SC_OK );
    res.setContentType( "text/html" );
    out.println( "<HTML>" );
    out.println( "<HEAD><TITLE>Envío
Realizado</TITLE></HEAD>" );
    out.println( "<BODY>" );

    consulta = "INSERT INTO articulos VALUES( '";
    consulta += req.getParameter( "titulo" )
+"','"+usuario+"','";
    consulta += req.getParameter("cuerpo")+"')";
    int result = stmt.executeUpdate( consulta );

    if( result != 0 ) {
      out.println( "Tu artículo ha sido aceptado e
insertado" );
      out.println( " correctamente." );
    } else {
      out.println( "Se ha producido un error en la
aceptación de tu " );
       out.println( "artículo.<BR>" );
      out.println( "Contacta con el Administrador de la
base de datos, " );
       out.println( "o consulta<BR>" );
      out.println( "el fichero <I>log</I> del servlet." );
    }
    out.println( "</BODY></HTML>" );
    out.flush();
    stmt.close();
  }

  // Devuelve la información del Servlet
  public String getServletInfo() {
    return "Servlet JDBC (Tutorial de Java), 1998";
  }

  public String autorizacion( HttpServletRequest req )
throws SQLException {
    Statement stmt = con.createStatement();
    String consulta;
    ResultSet rs;
    String valido = "";
    String usuario = req.getParameter( "usuario" );
    String clave = req.getParameter( "clave" );
    String permiso="";
```

```
    consulta = "SELECT admitirEnvio FROM usuarios WHERE
usuario = '"+usuario;
    consulta += "' AND clave = '"+clave+"'";
    rs = stmt.executeQuery( consulta );

    while( rs.next() ) {
      valido = rs.getString(1);
    }
    rs.close();
    stmt.close();

    if( valido.equals( "" ) ) {
      permiso = "ACCESO DENEGADO";
    } else {
      // Permiso sólo para lectura de artículos
      if( valido.equals( "N" ) ) {
        permiso = "GET";
      // Permiso para lectura y envío de artículos
      } else if( valido.equals( "S" ) ) {
        permiso = "POST";
      }
    }
  return permiso;
  }
}
```

A continuación se repasan los trozos de código más interesantes del servlet, tal como se ha hecho en muchos de los ejemplos del Tutorial. Una de las primeras cosas que hay que hace es reconocer cuando se ha recibido una petición correctamente, y para ello se utiliza el código que aparece en la línea siguiente:

```
res.setStatus( res.SC_OK );
```

La línea que sigue a ésta, es la que indica que el contenido es HTML, porque la intención del servlet es enviar respuestas en este formato al navegador. Esto se indica en la línea:

```
res.setContentType( "text/html" );
```

Otro trozo de código interesante es el utilizado para saber cual de los botones de la página inicial se ha pulsado, en donde se recurre al método *getParameter()* sobre el nombre del botón (**accion**), buscando los valores que se han asignado en el código fuente, y procesar el que se haya pulsado de los dos. Las líneas de código siguientes son las que realizan estas acciones.

```
String accion = req.getParameter( "accion" );
try {
  autorizado = autorizacion( req );
  if( accion.equals( "Leer Articulos" )
    && !autorizado.equals( "ACCESO DENEGADO" ) ) {
```

```
    leerArticulo( req,res );
  } else if( accion.equals( "Enviar" )
    && autorizado.equals( "POST" ) ) {
    enviarArticulo( req,res );
  } else if( accion.equals("Envio de Articulos" ) ) {
  . . .
  }
} catch( SQLException e ) {
  . . .
  }
```

Como se ha especificado en la página inicial de acceso un valor para cada uno de los botones, se sabe cuales deben buscarse y qué hacer para procesarlos. También se pueden recoger todos los parámetros que hay en un formulario a través del método *getParameterNames()*.

Una vez que se sabe el usuario y la clave, hay que comprobar esta información contra la base de datos. Para ello se utiliza una consulta para saber si el usuario figura en la base de datos. El código que realiza estas acciones es el que se reproduce en las siguientes líneas.

```
public String autorizacion( HttpServletRequest req ) throws
SQLException {
    Statement stmt = con.createStatement();
    String consulta;
    ResultSet rs;
    String valido = "";
    String usuario = req.getParameter( "usuario" );
    String clave = req.getParameter( "clave" );
    String permiso="";

    consulta = "SELECT admitirEnvio FROM usuarios WHERE
usuario = '"+usuario;
    consulta += "' AND clave = '"+clave+"'";
    rs = stmt.executeQuery( consulta );

    while( rs.next() ) {
      valido = rs.getString(1);
    }
    rs.close();
    stmt.close();

    if( valido.equals( "" ) ) {
      permiso = "ACCESO DENEGADO";
    } else {
      // Permiso sólo para lectura de artículos
      if( valido.equals( "N" ) ) {
        permiso = "GET";
      // Permiso para lectura y envío de artículos
      } else if( valido.equals( "S" ) ) {
        permiso = "POST";
      }
```

```
    }
  return permiso;
  }
```

Para realizar las consultas a la base de datos, es necesario crear una conexión con ella. Para hacerlo se utiliza el método *init()* de la clase **HttpServlet**, en donde se instancia la conexión con el servidor de base de datos, como se muestra en las líneas de código siguientes.

```
String DBurl = "jdbc:odbc:Tutorial";
. . .
try {
  Class.forName( "sun.jdbc.odbc.JdbcOdbcDriver" );
  con = DriverManager.getConnection
( DBurl,usuarioPost,clavePost );
. . .
```

Esta es solamente la parte de la conexión, hay más código implicado, pero ya se han visto bastantes ejemplos. No obstante, hay que tener en cuenta que la conexión se puede romper normalmente por tiempo, es decir, que si no se realizan acciones contra la base de datos en un tiempo determinado, la conexión se rompe. Así que, una de las cosas que debe controlar el lector, si va a utilizar este código es introducir la reconexión en la parte de código que trata la excepción de SQL..

Una vez creada la conexión, hay que realizar consultas para la lectura de datos. El siguiente código hace esto, consultando la base de datos y recogiendo la información de todos los artículos. Se puede incorporar fácilmente un nuevo campo con la fecha, para obtener los artículos ordenados por la fecha en que ha sido enviados, o realizar consultas diferentes para obtener los artículos ordenados por usuario, etc.

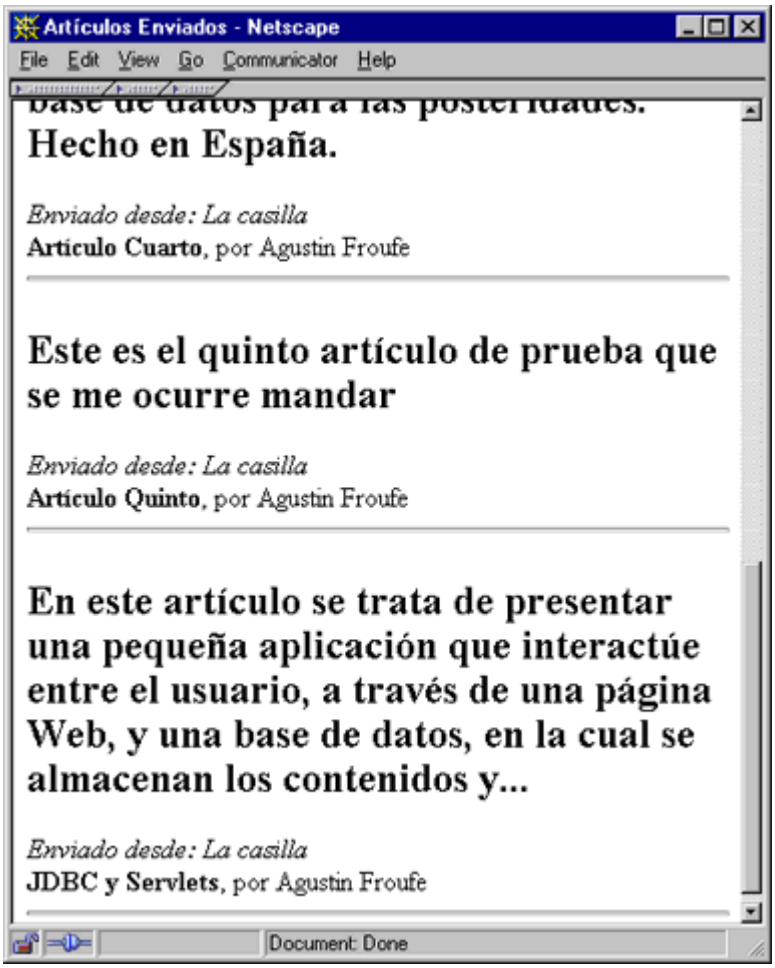
```
    consulta = "SELECT
articulos.cuerpo,articulos.titulo," );
    consulta +=
articulos.usuario,usuarios.nombre,usuarios.empresa ";
    consulta += "FROM articulos,usuarios WHERE " );
    consulta += articulos.usuario=usuarios.usuario";
    rs = stmt.executeQuery( consulta );

    PrintWriter out = new PrintWriter( res.getOutputStream
() );
    out.println( "<HTML>" );
    out.println( "<HEAD><TITLE>Artículos
Enviados</TITLE></HEAD>" );
    out.println( "<BODY>" );

    while( rs.next() ) {
      out.println( "<H2>" );
```

```
      out.println( rs.getString(1) );
      out.println( "</H2><p>" );
      out.println( "<I>Enviado desde: "+rs.getString(5)
+"</I><BR> " );
      out.println( "<B>"+rs.getString(2)+"</B>, por
"+rs.getString(4) );
      out.println( "<HR>" );
      }
    out.println( "</BODY></HTML>" );
    out.flush();
    rs.close();
    stmt.close();
    }
```

Aquí se llama al método *getString()* de cada columna de cada fila, ya que cada fila corresponde a un artículo. La figura siguiente muestra el resultado de la ejecución de una consulta de este tipo.



Otra parte interesante del código es la que se encarga del envío de los artículos y su inserción en la base de datos. Esto se realiza en las líneas que se muestran. Las cuestiones de autorización se encargan a otro método, así que en este no hay porqué considerarlas.

```
    consulta = "INSERT INTO articulos VALUES( '";
    consulta += req.getParameter( "titulo" )
+"','"+usuario+"','";
    consulta += req.getParameter("cuerpo")+"')";
    int result = stmt.executeUpdate( consulta );

    if( result != 0 ) {
      out.println( "Tu artículo ha sido aceptado e
insertado" );
      out.println( " correctamente." );
    } else {
      out.println( "Se ha producido un error en la
aceptación de tu " );
       out.println( "artículo.<BR>" );
      out.println( "Contacta con el Administrador de la
base de datos, " );
       out.println( "o consulta<BR>" );
      out.println( "el fichero <I>log</I> del servlet." );
    }
    out.println( "</BODY></HTML>" );
    out.flush();
    stmt.close();
  }
```

Con esto, se ha presentado al lector un ejemplo en el que se accede a la base de datos desde el servidor, y la parte cliente se limita a utilizar los recursos del servidor Web para acceder a la información de la base de datos. El ejemplo es específico para servlets HTTP, aunque se pueden escribir servlets que devuelvan tipos binarios en lugar de una página *html*; Por ejemplo, se puede fijar el tipo de contenido a '*image/gif*' y utilizar el controlador de **OutputStream** para escribir una imagen *gif* que se construya en el mismo momento al navegador. De este modo se pueden pasar imágenes que están almacenadas en la base de datos del servidor a la parte cliente, en este caso, el navegador.

8.- Introducción a SQL

Esta sección viene a intentar introducir un poco al lector que no conoce SQL, en su mundo y a entender mejor los ejemplos que se han propuesto. No se intenta hacer un manual de SQL, sino simplemente recordar algunas de las características y comandos de este lenguaje. Como JDBC requiere que los drivers sean compatibles con la versión estándar ANSI SQL-92, esa será la que se comente, aunque en la actualidad SQL sigue evolucionando y mejorando, pasando de ser un sublenguaje de manejo de datos a un verdadero lenguaje de programación, pero hay que tener siempre presente que una de las primeras frases que aparece en la especificación oficial de JDBC es:

"In order to pass JDBC compliance tests and to be called 'JDBC compliant', we require that a driver support at least ANSI SQL-92 Entry Level"

Lo cual es claramente imposible de cumplir con drivers para sistemas de manejo de bases de datos antiguos (DBMs) y para algunas versiones de SQL particularizadas por los fabricantes de algunos de los sistemas de bases de datos más evolucionados.

El Modelo Relacional

Aunque SQL está basado en el modelo relacional, no representa una implementación excesivamente rígida. En él, las unidades básicas son *tablas*, *columnas* y *filas*. Si se habla estrictamente en términos relacionales, la *relación* es mapeada en una *tabla* y proporciona una forma de relacionar los datos dentro de una tabla de forma simple. Una *columna* representa un dato presente en la tabla, mientras que una *fila* representa un registro, o entrada, de la tabla. Cada fila contiene un valor determinado para cada una de las columnas (un valor puede estar vacío, o indefinido, y ser considerado válido). La tabla se puede visualizar como una matriz, con las columnas en vertical y las filas en horizontal. La tabla siguiente *Empleados*, por ejemplo, se podría utilizar para almacenar los empleados de una empresa.

empleado	apellido	nombre	categoria
00001	González	José	Administrativo
00012	Gómez	Pédro	Ingeniero
00045	Pérez	Gonzalo	Perito
00023	López	Alejandro	Ingeniero

Unas cuantas reglas sintácticas básicas de SQL, sobre las que el lector debe prestar especial atención, o tener cuidado, son:

- SQL no es sensible a los espacios en blanco. Los retornos de carro, tabuladores y espacios en blanco no tienen ningún significado especial a la hora de la ejecución de sentencias. Las palabras clave y comandos están delimitados por comas (,), cuando es necesario, y se utilizan paréntesis para agruparlos.

- Cuando se realizan múltiples consultas a un mismo tiempo, se debe utilizar el punto y coma (;) para separar cada una de las consultas.

- Las consultas no son sensibles a mayúsculas o minúsculas. Sin embargo, hay que prestar mucha atención, porque aunque en las palabras clave no importe el que estén en mayúsculas o minúsculas, en las cadenas que se almacenen en las tablas como valores sí que permanece el que se hayan introducido en mayúsculas o minúsculas; y hay que tener esto presente, sobre todo a la hora de hacer comparaciones.

Aunque se pueden tener todos los datos en una sola tabla, no es habitual que esto sirva para todos los casos. Por ejemplo, si en la tabla anterior de *Empleados*, se quisiera ahora añadir información sobre los departamentos de la empresa y en cuáles están adscritos los empleados, se podría añadir a esa misma tabla; sin embargo, el propósito de la tabla *Empleados* es almacenar datos sobre los empleados, no sobre la empresa. La solución consiste en crear otra tabla, *Departamentos*, en donde guardar la información específica de los departamentos de la empresa. Para asociar un empleado con un departamento, solamente habría que añadir una columna a la tabla *Empleados* para guardar el nombre o número del departamento. Ahora que ya están colocados los empleados en cada departamento, se puede incorporar otra tabla para guardar información de los *Proyectos* en que están involucrados.

Empleados				
empleado	apellido	nombre	categoria	departamento
00023	López	Alejandro	Ingeniero	022
00012	Gómez	Pedro	Ingeniero	011
00045	Pérez	Gonzalo	Perito	022

Departamentos		
departamento	jefe (Empleados)	sede
011	00023	Sevilla
022	00012	Madrid

Proyectos			
proyecto	titulo	tipo	lider
001	Corre que te pillo	juego	00045
003	Cafetera	simulador	00012

Ahora está más claro cómo están separados lógicamente los datos. Este proceso es el que más tiempo requiere en el desarrollo de una base de datos, y representa el primer nivel en donde se declara el esquema de las relaciones. El *esquema* de la bases de datos es el contenedor de más alto nivel que la define como una colección de tablas, en donde cada tabla cae dentro de un esquema. Del mismo modo, un *catálogo* es una entidad que puede englobar a varios esquemas. Esta abstracción es precisamente la parte más necesaria dentro de un robusto sistema relacional de bases de datos (RDBMs).

La razón principal de una base de datos es el acceso a la información, es decir, facilitar que se pueda leer una tabla, que se pueda cambiar la información que contiene, y también que se puedan crear y destruir tablas. Además, permite que se establezcan niveles de seguridad, pro ejemplo, se puede incorporar otra tabla, llamada *Condifencial*, para guardar la dirección, teléfono y sueldo de cada empleado, que tendrá que estar colocada en un esquema separado para que solamente el departamento de tesorería pueda acceder a esos datos.

Si se observan las tablas anteriores, las tres se encuentran enlazadas. La tabla *Empleados* contiene una columna que contiene el número del departamento al que pertenece cada empleado. Este número de departamento también aparece en la tabla *Departamentos*, que describe cada departamento de la empresa. Las tablas *Empleados* y la nueva *Confidencial*, están relacionadas, pero todavía se necesita incorporar una entrada en una de las tablas que

permita acceder a la otra, para hacer una distinción con el número de empleado.

El enlace que se ha establecido, entre número de empleado y número de departamento, se conoce como *índice* (*key*). Un índice se utiliza para identificar información dentro de una tabla. Cada empleado individual o departamento, deberían tener un único índice para facilitar posteriores acciones sobre las tablas. Según el modelo relacional, el índice se supone único dentro de una tabla, ninguna otra entrada en la tabla debe tener el mismo índice primario, o *clave primaria*.

Una sola columna suele ser suficiente para identificar inequívocamente a una fila o entrada. Sin embargo, también se puede usar una combinación de filas para componer un índice primario. Por ejemplo, se puede utilizar una combinación del departamento y sede de ese departamento para componen su clave primaria. En SQL, las columnas definidas como claves primarias, deben estar definidas, no pueden ser nulas.

Lo mejor es, pues, repartir los datos en tablas donde se encuentren lógicamente asociados; aunque, puede ser que haya datos que deban estar en más de una tabla, como es en este caso el número de empleado, que debe estar en la tabla Empleados y en la tabla Confidencial. Se puede necesitar tener la seguridad de que si una fila existe en una tabla, es necesario que exista la correspondiente fila en la tabla relacionada; en el ejemplo, se puede decir que por cada entrada en la tabla de Empleados debe haber otra entrada en la tabla Confidencial. Esta asociación se puede solidificar con el uso de *claves ajenas*, donde una columna determinada en una tabla depende de otra columna en la tabla *padre*. En esencia, se está construyendo una columna virtual en una tabla, en base a una columna real de otra tabla. En el ejemplo, se ha enlazado la columna con el número del empleado de la tabla *Confidencial* al número de empleado de la tabla *Empleados*; se está indicando que el número de empleado es un índice en la tabla *Confidencial*, de ahí el término de *clave ajena*. Una clave primaria también puede contener una clave ajena si es necesario.

La forma en que se modelan los datos y las técnicas que se utilizan para construir claves primarias y ajenas, caen dentro del diseño de bases de datos, que esta pequeña introducción a SQL no puede abordar.

Creación de Tablas

La sentencia CREATE TABLE es la que se usa para crear una tabla. Es una operación importante pero muy sencilla. Hay algunas fuentes de datos que solamente admiten en las tablas elementos muy simples, como por ejemplo, las fuente de texto accedidas a través de ODBC. El formato de la sentencia es:

```
CREATE TABLE <nombre tabla> (<elemento columna> [,<elemento
columna]...)
```

El *elemento columna* se declara de la forma:

```
<nombre columna> <tipo de dato> [DEFAULT <expresión>]
    [<constante columna> [,<constante columna]...]
```

Siendo *constante columna* quien indica la forma o característica de la columna, que puede ser:

```
NOT NULL | UNIQUE | PRIMARY KEY
```

Siguiendo con el ejemplo, en este caso lo único que hay que tener presente es que es necesario definir la tabla de referencia, en este caso *Empleados*, antes de definir la tabla que hace referencia a ella, en este caso *Confidencial*. La tabla *Empleados* se crearía de la siguiente forma:

```
CREATE TABLE Empleados  (
    empleado CHAR(5) PRIMARY KEY,
    apellido VARCHAR(20) NOT NULL,
    nombre VARCHAR(20) NOT NULL,
    categoria VARCHAR(20) NOT NULL,
    departmento VARCHAR(20) );
```

En la creación de la tabla se utilizan dos tipos de datos para especificar cadenas: CHAR y VARCHAR. El sistema de base de datos utiliza exactamente la cantidad de espacio que se indique cuando se utiliza un tipo de datos CHAR; en caso de que se indique que una entrada es de tipo CHAR(**n**) y se rellene con una cena de tamaño inferior a **n**, los caracteres que resten se rellenan con espacios. Con VARCHAR, se almacena exactamente la cadena que se indique, el tamaño que se indica en la creación solamente sirve para fijar el tamaño máximo del valor que se puede almacenar.

También se utiliza la directiva NOT NULL que hace que se compruebe cada una de las entradas en la columna. La creación de la tabla *Confidencial* es igualmente sencilla, la única diferencia es la

incorporación de la palabra clave REFERENCES para que esta tabla pueda utilizar el atributo correspondiente al número de empleado de la tabla *Empleados* como su clave primaria.

```
CREATE TABLE Confidencial (
    empleado CHAR(5) PRIMARY KEY,
    direccion VARCHAR(50),
    telefono VARCHAR(12),
    sueldo DECIMAL,
    FOREIGN KEY( empleado ) REFERENCES Empleados
( empleado ) );
```

La sentencia para eliminar una tabla es DROP TABLE, que es igual de sencilla que la sentencia de creación, y su forma es:

```
DROP TABLE <nombre tabla>
```

Recuperar Información

La sentencia SELECT es la que se utiliza cuando se quieren recuperar datos de la información almacenada en un conjunto de columnas. Las columnas pueden pertenecer a una o varias tablas y se puede indicar el criterio que deben seguir las filas de información que se extraigan. Muchas de las cláusulas que permite esta sentencia son simples, aunque se pueden conseguir capacidades muy complejas a base de una gramática más complicada.

Desde luego, la mejor forma de entender el proceso de consulta a la base de datos y el uso de las cláusulas que modifican al comando SELECT, es pensar en términos de conjuntos matemáticos. SQL, al igual que todos los lenguajes de cuarta generación está diseñado para responder a cuestiones de *tipo ¿Qué quiero hacer?*, al contrario que los otros lenguajes de programación, como Java y C++, que intentan resolver cuestiones del tipo *¿Cómo lo hago?*.

El dominio de las consultas SQL no es una tarea sencilla, pero con un poco de sentido común y algo de intuición se pueden conseguir resultados muy eficientes, gracias al modelo relacional en el que se base SQL.

La sintaxis de la sentencia es:

```
SELECT [ALL | DISTINCT] <seleccion>
    FROM <tablas>
    WHERE <condiciones de seleccion>
```

```
    [ORDER BY <columna> [ASC | DESC]
        [,<columna> [ASC | DESC]]...]
```

La *seleccion* contiene normalmente una lista de columnas separadas por coma (,), o un asterisco (*) para seleccionarlas todas. Un ejemplo ejecutado contra una de las tablas creadas anteriormente podría ser:

```
SELECT * FROM Empleados;
```

Que devolvería el contenido completo de la tabla. Si solamente se quieren conocer los empleados del departamento 022, la consulta sería:

```
SELECT * FROM Empleados
    WHERE departamento = '022';
```

Para ordenar la lista resultante por apellidos, por ejemplo, se usaría la directiva `ORDER BY`:

```
SELECT * FROM Empleados
    WHERE departamento = '022'
    ORDER BY apellido;
```

Si lo que se quiere, además de que la lista esté ordenada por apellido, es ver solamente el número de empleado, se consultaría de la forma:

```
SELECT empleado FROM Empleados
    WHERE departamento = '022'
    ORDER BY apellido;
```

Si se quieren resultados de dos tablas a la vez, tampoco hay problema en ello, tal como se muestra en la siguiente sentencia:

```
SELECT Empleados.*, Confidencial.*
    FROM Empleados, Confidencial;
```

También se pueden realizar consultas más complicadas; por ejemplo*, mostrar el sueldo de los empleados del departamento 022*. Según las tablas, la información del sueldo se encuentra en la tabla *Confidencial*, y el departamento al que está adscrito un empleado se encuentra en la tabla *Empleados*. Para asociar una comparación de una tabla con otra, se puede utilizar las referencia la número de empleado en la tabla *Confidencial* desde la tabla *Empleados*. Se pueden especificar los empleados que pertenecen a un departamento y utilizar los números de empleados resultantes para obtener la información sobre su sueldo desde la tabla *Confidencial*.

```
SELECT c.sueldo
    FROM Empleados AS e, Confidencial AS c
```

```
    WHERE e.departmento = '022'
      AND c.empleado = e.empleado;
```

Aquí se ha declarado algo parecido a una variable con la cláusula AS. Ahora se pueden hacer referencias a campos específicos de la tabla utilizando un punto (.), como si se tratase de un objeto. Se puede, por ejemplo, determinar cuántos empleados de la empresa tienen un sueldo por encima de los 1200 euros.

```
SELECT salary
    FROM Confidencial
    WHERE sueldo > 1200;
```

Y también se pueden conocer los empleados del departamento 022 que cobran más de los 1200 euros.

```
SELECT c.sueldo
    FROM Empleados AS e, Confidencial AS c
    WHERE  e.departmento = '022'
      AND c.empleado = e.empleado
      AND c.sueldo > 1200;
```

También se pueden realizar un cierto número de funciones en SQL, incluyendo algunos cálculos matemáticos y estadísticos. Por ejemplo, se puede saber la media de sueldo de los empleados del departamento 022 de la siguiente forma:

```
SELECT AVG( c.sueldo )
    FROM Empleados as e, Confidencial as c
    WHERE e.departmento = '022'
      AND c.empleado = e.empleado;
```

Desde luego, las posibilidades que ofrece SQL exceden en mucho el alcance de este capítulo y los pocos ejemplos que se han presentado. Además, como el interés del autor es introducir a JDBC, tampoco es necesario el uso de ejemplos complejos. Si el lector tiene interés en aprender más sobre SQL, debería recurrir a uno de los muchos y buenos libros que hay publicados sobre ello.

Almacenar Información

La sentencia INSERT se utiliza cuando se quieren insertar filas de información en las columnas. Aquí también se pueden presentar diferentes capacidades, dependiendo del nivel de complejidad soportado. La sintaxis de la sentencia es:

```
INSERT INTO <nombre tabla>
    [(<nombre columna> [,<nombre columna]...)]
```

```
    VALUES (<expresion> [,<expresion>]...)
```

Por ejemplo, en la tabla de los Empleados se podría ingresar uno nuevo con la siguiente información:

```
INSERT INTO Empleados
    VALUES ( ’00066’, ’Garrido’, ’Juan’, ’Ingeniero’ ,
’022’ );
```

Si la gramática del *driver* utilizado lo soporta, se puede utilizar una cláusula `SELECT` para cargar varias columnas a la vez.

La sentencia `DELETE` es la que se emplea cuando se quieren eliminar filas de las columnas, y su gramática también es muy simple:

```
DELETE FROM <nombre tabla> WHERE <condicion busqueda>
```

Si no se especifica la cláusula `WHERE`, se eliminará el contenido de la tabla completamente, sin eliminar la tabla, por ejemplo:

```
DELETE FROM Empleados;
```

Vaciará completamente la tabla, dejándola sin ningún dato en las columnas, es decir, esencialmente lo que hace es borrar todas las columnas de la tabla. Especificando la cláusula `WHERE`, se puede introducir un criterio de selección para el borrado, por ejemplo:

```
DELETE FROM Empleados WHERE empleado=’00001’;
```

También se pueden borrar múltiples filas en esta operación, siempre que la cláusula `WHERE` permita seleccionar más de una fila, todas ellas serán eliminadas.

Para actualizar filas ya existentes en las columnas, se utiliza la sentencia `UPDATE`, cuya gramática es mínima:

```
UPDATE <nombre tabla>
    SET <nombre columna = ( <expresion> | NULL )
        [, <nombre columna = ( <expresion> | NULL )]...
    WHERE <condicion busqueda>
```

Este comando permite cambiar uno o más campos existentes en una fila. Por ejemplo, para cambiar el nombre de un empleado en la tabla de *Empleados*, se haría:

```
UPDATE Empleados
    SET nombre = 'Pedro Juan'
    WHERE empleado='00012';
```

Código Independiente y Portable

De nuevo, tenga el lector presente que lo que a continuación se refleja son simples opiniones y sugerencias, que solamente están destinadas a que los problemas que se presenten sean los menos posibles, cuando se intente programar con Java, y en este caso, con JDBC.

Uno de los objetivos fundamentales a la hora de diseñar JDBC fue obtener la máxima portabilidad posible entre distintos Sistemas de Gestión de Bases de Datos. Distintos gestores tienden a utilizar distinta sintaxis para las mismas cosas, como por ejemplo para especificar una fecha, ejecutar un procedimiento almacenado, etc. JDBC proporciona una serie de cláusulas de escape de modo que se pueda escribir una fecha, etc., de forma portable: será luego el driver el que se encargue de convertir la información al formato que requiere la base de datos. Todas las cláusulas de escape se escriben siempre entre llaves, "**{...}**".

Una fecha se especifica utilizando el formato **{d 'aaaa-mm-dd'}**, donde **d** indica que se está hablando de una fecha, y **aaaa** serán los cuatro dígitos correspondientes a un año, **mm** los dos dígitos del mes, y **dd** los dos dígitos del día. Una hora se escribirá utilizando el formato **{t 'hh:mm:ss'}**, donde se usa **hh** para la hora, **mm** para los minutos, y **ss** para los segundos. Para un valor de fecha/hora se utilizará **{ts 'aaaa-mm-dd hh:mm:ss.f . . .'**}, donde **f**, la parte fraccionaria de los segundos, es optativo.

No sólo se utilizan cláusulas de escape para las constantes de fecha y hora, también para llamar a un procedimiento almacenado hay una sintaxis especial que permite aislarse de la sintaxis concreta de cada base de datos: el formato será **{call nombre_proc[(?, ?, . . .)]}**, donde **call** indica que se está invocando un procedimiento almacenado, **nombre_proc** será su nombre, y los distintos parámetros, si los hay, se indicarán mediante una serie de caracteres de interrogación, **?**, encerrados entre paréntesis.

También para invocar una función SQL existe una cláusula de escape, con la sintaxis **{fn upper("Texto")}**, donde **fn** indica que estamos llamando a una función, de nombre **upper**, y a la que se le pasa el argumento "Texto".

También hay cláusulas de escape para las composiciones *externas (outer-joins*), utilizando la sintaxis **{oj outer-join}**, donde **outer-join** tiene la forma :

```
tabla LEFT OUTER JOIN {tabla | outer-join} ON
condición_de_búsqueda
```

A la hora de utilizar dentro de un SELECT la palabra reservada LIKE se pueden usar dos caracteres como comodín, (_) y (%). Para buscar en la base de datos un texto en que aparezcan estos dos caracteres, basta con poner delante de ellos un carácter especial que indique que en ese lugar se utilizan como cualquier otro carácter, no como comodines. Para ello, se debe indicar cuál es dicho carácter especial utilizando la sintaxis **{escape 'carácter'}**, por ejemplo

```
SELECT nombre FROM Empleados WHERE nombre LIKE '@_' {escape
'@'}
```

En este ejemplo, se indica que el carácter especial de escape es **@**, por lo que la cadena **LIKE '@_'** indica que se deben buscar todos los nombre que comiencen por '**_**', en lugar de usarlo como comodín.

Si se quiere escribir código JDBC para que sea portable e independiente de la máquina y motor de base de datos que se va a atacar, deberían, además de tener en cuenta las recomendaciones anteriores, seguirse unas sencillas reglas a la hora de escribir código. Algunas de ellas son las que se recogen en los siguientes párrafos.

- Presérvese siempre el uso de mayúsculas y minúsculas de la salida de los métodos *getTables()* y *getColumns()*

- Llamar al método *getIdentifierQuoteString()* y utilizar la cadena que devuelva para delimitar los identificadores

- Tener en cuenta que no todos los sistemas soportan SQL ’92 estrictamente

- Utilícese *setNull()*

```
      PreparedStatement stmt = con.prepareStatement(
        "update nombre set departamento = ? where empleado
= 00012" );
      stmt.setString( 1,"" ); // Erróneo!!
      stmt.setNull( 1 );
```

```
stmt.executeUpdate();
```

- Comprobar los límites de la sesión con *getMaxStatements() y getMaxConnections()*. Por ejemplo, Microsoft SQL restringe a una sola sentencia activa y hay algunos drivers que limitan las conexiones a una sola

- Utilizar las comillas ...pero no en todas partes

- Las pseudo-columnas nunca deben ir entre comillas. Por ejemplo, *filaid*, *regid*, *usuario*, *empleado*

- Tener cuidado con algunas cosas de SQL ’92:

```
"select * from Empleados where empleado = NULL"
// nunca devuelve filas!
"select * from Empleados where empleado is NULL"
// puede devolver filas
"select * from Empleados where empleado = ?"
// usar setNull para el parámetro
// nunca devolverá filas
```

- Tener cuidado con los Nombres, si *supportSchemasInDataManipulation()* indica que sí se pueden soportar esquemas, utilizar siempre un punto (.) entre el nombre del esquema y el nombre de la tabla. ¡No olvidar las comillas!

- A la hora de crear una tabla usar *getTypeInfo()* y buscar entre el resultado las mejores opciones. Por ejemplo, si se quiere crear una columna DECIMAL(8,0), buscar un SQL DECIMAL o SQL NUMERIC, luego SQL INTEGER, luego SQL DOUBLE o SQL FLOAT, luego SQL CHAR o SQL VARCHAR; e ignorar tipos que no coincidan con los requerimientos: moneda, autoincremento, etc.

- Utilizar parámetros en lugar de constantes en las sentencias SQL

- Algunas bases de datos no soportan constantes cadena para columnas LONG

- Usar *setObject()*, que reduce la complejidad del programa. Por ejemplo, si se usan buffers de cadenas, entonces es necesario *setObject()*

- Para parámetros de salida, no olvidarse de utilizar *registerOutParameter()*