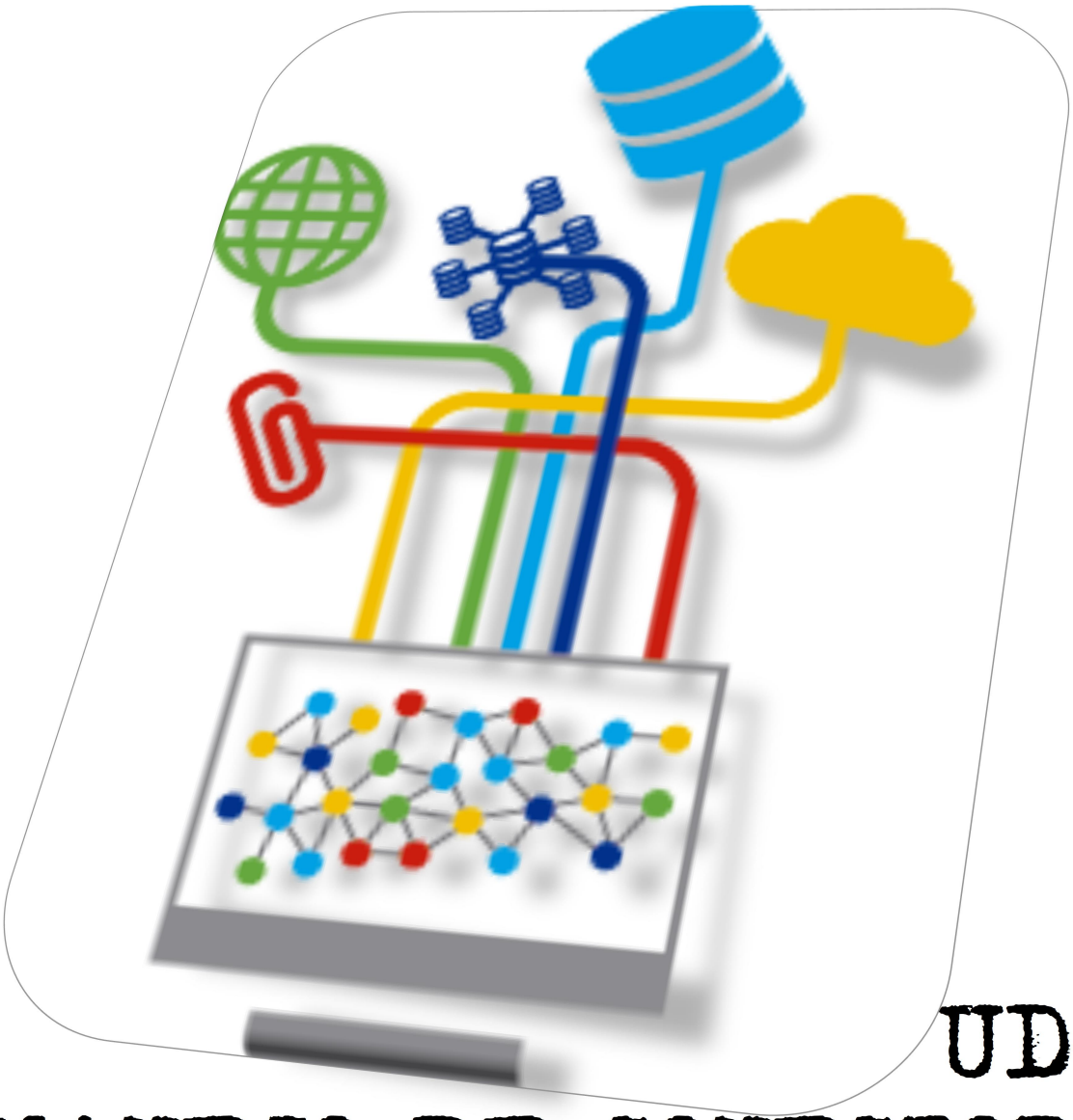
ACCESO A DATOS
2º CFGS D.A.M.



UD. 2
MANEJO DE CONECTORES

Qué es un conector

Los conectores son el software que se necesita para realizar las conexiones desde nuestro programa Java con una base de datos relacional.

El desfase Objeto-Relacional

Actualmente las bases de datos orientada a objetos están ganando cada vez más aceptación frente a las relacionales, ya que solucionan las necesidades de aplicaciones más sofisticadas que requieren el tratamiento de elementos más complejos.

Las bases de datos relacionales no están diseñadas para almacenar estos objetos ya que existe un desfase entre las construcciones típicas que proporciona el modelo de datos relacional y las proporcionadas por los ambientes de programación basados en objetos; es decir, al guardar los datos de un programa bajo el enfoque O.O. se incrementa la complejidad del programa dando lugar a más código y más esfuerzo de programación debido a la diferencia de esquemas entre los elementos a almacenar (objetos) y las características del repositorio de la base de datos (tablas). A esto se denomina **desfase objeto-relacional** (desajuste de impedancia) y se refiere a los problemas que ocurren debido a las diferencias entre el modelo de datos de la base de datos y el lenguaje de programación orientado a objetos.

Sin embargo, el paradigma relacional y el Orientado a Objetos pueden compatibilizarse. Cada vez que los objetos deben extraerse o almacenarse en una base de datos relacional se requiere un mapeo desde las estructuras provistas en el modelo de datos a las provistas por el entorno de programación.

Bases de datos embebidas

Al desarrollar pequeñas aplicaciones en las que no vamos a almacenar grandes cantidades de información no es necesario que utilicemos un SGBD como Oracle o MySQL. Podemos utilizar en su lugar una base de datos embebida donde el motor esté incrustado en la aplicación y sea exclusivo para ella. La Base de datos se inicia cuando se ejecuta la aplicación y termina cuando se cierra la aplicación.

SQLITE

Es un sistema gestor de base de datos multiplataforma escrito en C que proporciona un motor muy ligero. Las bases de datos se guardan en forma de ficheros por lo que es fácil trasladar la base de datos con la aplicación que la usa. Cuenta con una utilidad que nos permitirá ejecutar comandos SQL contra una base de datos SQLite en modo consola. Es un proyecto de dominio público.

La biblioteca implementa la mayor parte del estándar SQL-92 y se puede utilizar desde programas en C/C++, PHP, Visual Basic, Perl, Delphi, Java, Android, …

Su instalación es sencilla. Desde la página http://www.sqlite.org/download.html se puede descargar. Para Windows podemos descargar el fichero ZIP ***sqlite-tools-win32-x86-3150100.zip***, al descomprimirlo obtenemos un único fichero ejecutable (sqlite3exe). Al ejecutarlo desde la línea de comandos escribimos el nombre del fichero que contendrá la base de datos, si el fichero no existe se creará si existe cargará la base de datos. El siguiente ejemplo crea la base de datos ejemplo.db (en la carpeta D:\db\SQLite), todas las tablas que creemos en estta sesión se almacenarán en este fichero, para finalizar se escribe el comando .quit:

```
sqlite> BEGIN TRANSACTION;
sqlite> CREATE TABLE departamentos (
…> dept_no TINYINT(2) NOT NULL PRIMARY KEY,
…> denombre VARCHAR(15),
…> loc VARCHAR(15)
…> );
sqlite>
sqlite>
sqlite> INSERT INTO departamentos VALUES (10,'CONTABILIDAD','SEVILLA');
sqlite> INSERT INTO departamentos VALUES (20,'INVESTIGACIÓN','MADRID');
sqlite> INSERT INTO departamentos VALUES (30,'VENTAS','BARCELONA');
sqlite> INSERT INTO departamentos VALUES (40,'PRODUCCIÓN','BILBAO');
sqlite> COMMIT;
sqlite> .quit
```

Para más información sobre el manejo de bases de datos SQLite puedes leer la siguiente web:

https://usemossoftwarelibre.wordpress.com/cc/tutorial-sqlite-en-espanol/capitulo-4-tutorial-sqlite/

OTRAS BASES DE DATOS EMBEBIDAS

Apache Derby, es una base de datos relacional de código abierto, implementado en su totalidad en Java que forma parte del Apache DB subproject y está disponible bajo la licencia Apache, versión 2.0. Algunas ventajas de esta base de datos son: su tamaño reducido, basada en Java y soporte de los estándares SQL, ofrece un controlador integrado JDBC que permite incrustar Derby en cualquier solución basada en Java, soporta el modelo cliente-servidor utilizando el Derby Network Server. Además, es muy fácil de implementar y utilizar.

Descarga en: http://db.apache.org/derby/derby_downloads.html

HSQLDB, es un sistema gestor de bases de datos relacional escrito en Java. Es compatible con SQL ANSI-92 y SQL:2008. Puede manterr la base de datos en memoria o en ficheros en disco. Permite interidad referencial, procedimientos almacenados en Java, disparadores y tablas en disco de hasta 8Gb. Se distribuye con licencia BSD que es una licencia muy cercana al domino público.

Descarga en: http://sourceforge.net/projects/hsqldb/files/

H2, es un sistema gestor de Base de Datos relacional programado íntegramente en Java. Está disponible omo software de código libre bajo Licencia Pública de Mozilla o la Eclipse Public License.

Descarga en: http://www.h2database.com/html/main.html

FIREBIRD es un sistema gestor de bases de datos relacional de código abierto que no tiene licencias duales, por lo que es totalmente libre y se puede usar tanto en aplicaciones comerciales como de código abierto.

SQL Server Mobile Es una base de datos de Microsoft, compacta y con una gran variedad de funciones diseñada para admitir una amplia lista de dispositivos inteligentes y Tablets.

Oracle Embedded Ofrece un conjunto de soluciones al desarrollo software que aporta ciertas ventajas como la instalación silenciosa, configuración automática o funcionamiento desatendido.

DB4O (DataBase 4 Objects) es un motor de base de datos orientado a objetos Se puede utilizar de forma embebida o en aplicaciones cliente-servidor. Está disponible para entornos Java y .Net. Dispone de licencia dual GPL/comercial. Proporciona algunas características destacables como:

- Evitar el problema del desfase Objeto-Relacional
- No existir un leguaje SQL para la manipulación de datos, en su lugar existen métodos delegados.
- Se instala añadiendo un único fichero de librería (JAR para Java o DLL para .NET)
- Se crea un único fichero de base de datos con la extensión .YAP.

Descarga en:

http://www.java2s.com/Code/Jar/d/Downloaddb4o8018415484alljava5jar.htm

Al descomprimir el archivo db4o-*x.x*-java.zip veremos la carpeta lib que contiene los JAR que tenemos que agregar a nuestro proyecto para utilizar el motor de la base de datos.

Si usamos eclipse, podremos crear una librería con todos los JAR para ello seleccionamos nuestro proyecto y con la opción Build Paths > Add Libreries. Posteriormente, seleccionamos User Library desde la ventana a la que accedamos.

En la ventana siguiente, pulsamos User Libraries, New (donde daremos el nombre a la librería, por ejemplo Db4oLib. Y pulsamos el botón Add JARs, localizando todos los db4o-.jar de la carpeta lib y pulsamos el botón Abrir.

La librería estará incluida en nuestro proyecto.

Vamos a crear una base de datos de objetos Personas. Para ello, creamos el objeto Persona en nuestro IDE:

```
public class E1Persona {

   private String nombre;
   private String ciudad;

   public E1Persona (String nombre, String ciudad){
      this.nombre=nombre;
      this.ciudad=ciudad;
   }
   public E1Persona(){
      this.nombre=null;
      this.ciudad=null;
   }

   //GETTERS Y SETTERS
   public String getNombre(){return nombre;}
```

```
   public void setNombre(String nombre) {
      this.nombre = nombre;
   }

   public void setCiudad(String dir){this.ciudad=dir;}

   public String getCiudad() {
      return ciudad;
   }
}
```

Posteriormente creamos nuestra aplicación principal que cree la base de datos e inserte objetos Persona en ella:

```
import com.db4o.Db4oEmbedded;
import com.db4o.ObjectContainer;
import com.db4o.ObjectSet;

public class E1Principal {

   final static String BDPer = "D:/db/db4o/DBE1Persona.yap";

   public static void main(String[] args) {
      ObjectContainer
db=Db4oEmbedded.openFile(Db4oEmbedded.newConfiguration(),BDPer);

      //Creamos E1Personas
      E1Persona p1 = new E1Persona ("Juan", "Guadalajara");
      E1Persona p2 = new E1Persona ("María", "Madrid");
      E1Persona p3 = new E1Persona ("Enrique", "Málaga");
      E1Persona p4 = new E1Persona ("Manuel", "Sevilla");

      //Almacenar objetos Persona en la base de datos
      db.store(p1);
      db.store(p2);
      db.store(p3);
      db.store(p4);

      //Cerramos la base de datos
      db.close();
   }
}
```

Para realizar cualquier acción, ya sea insertar, modificar o realizar consultas debemos manipular una instancia de ObjectContainer donde se define el fichero de base de datos.

Si queremos recuperar objetos, podemos utilizar la interfaz Query-By-Example queryByExample(). El siguiente ejemplo muestra todos los objetos Persona existentes en la base de datos, si no existe ninguno el método size() aplicado al objeto ObjectSet devuelve 0:

```
import com.db4o.Db4oEmbedded;
import com.db4o.ObjectContainer;
import com.db4o.ObjectSet;
public class E1LeerDatos {
   final static String BDPer = "D:/db/db4o/DBE1Persona.yap";
```

```
   public static void main(String[] args) {
      ObjectContainer db=Db4oEmbedded.openFile(BDPer);
      //LEER TODOS LOS REGISTROS
      System.out.println("Todas las personas son:");
      E1Persona per = new E1Persona(null,null);
      ObjectSet<E1Persona> result = db.queryByExample(per);
      if(result.size()==0) System.out.println("No existen Registros de
Personas..");
      else{
         System.out.println("Número de registros: "+result.size());

         while(result.hasNext()){
            E1Persona p=result.next();
            System.out.println("Nombre: "+p.getNombre()+",
Ciudad:"+p.getCiudad());
         }
      }
      //LEER SÓLO LOS REGISROS DE AQUELLOS LLAMADOS Enrique
      System.out.println("Todos los Enrique son...");
      E1Persona perEnrique = new E1Persona("Enrique",null);
      ObjectSet<E1Persona> resultEnrique = db.queryByExample(perEnrique);
      if(resultEnrique.size()==0) System.out.println("No existen Registros
de Personas..");
      else{
         System.out.println("Número de registros: "+resultEnrique.size());

         while(resultEnrique.hasNext()){
            E1Persona p=resultEnrique.next();
            System.out.println("Nombre: "+p.getNombre()+",
Ciudad:"+p.getCiudad());
         }
      }
      //LEER SÓLO LOS REGISTROS DE AQUELLAS PERSONAS QUE VIVAN EN Madrid
      System.out.println("En Madrid viven...");
      E1Persona perMadrid = new E1Persona(null,"Madrid");
      ObjectSet<E1Persona> resultMadrid = db.queryByExample(perMadrid);
      if(resultMadrid.size()==0) System.out.println("No existen Registros
de Personas..");
      else{
         System.out.println("Número de registros: "+resultMadrid.size());

         while(resultMadrid.hasNext()){
            E1Persona p=resultMadrid.next();
            System.out.println("Nombre: "+p.getNombre()+",
Ciudad:"+p.getCiudad());
         }
      }
      //CIERRA LA BASE DE DATOS
      db.close();
   }
}
```

Si lo que queremos es modificar un objeto, utilizaremos el método store(), previa localización del mismo.

```
import com.db4o.Db4oEmbedded;
import com.db4o.ObjectContainer;
```

```
import com.db4o.ObjectSet;
public class E1ModificarDatos {
   final static String BDPer = "D:/db/db4o/DBE1Persona.yap";
   public static void main(String[] args) {
      ObjectContainer db=Db4oEmbedded.openFile(BDPer);

      //CAMBIAR LA CIUDAD DONDE VIVE JUAN POR TOLEDO
      E1Persona perJuan = new E1Persona("Juan",null);
      ObjectSet<E1Persona> resultJuan = db.queryByExample(perJuan);
      if(resultJuan.size()==0) System.out.println("No existen Juan...");
      else{
         E1Persona existe = (E1Persona) resultJuan.next();
         existe.setCiudad("Toledo");
         db.store(existe); //ciudad modificada
         //consultar los datos
         ObjectSet<E1Persona> result=db.queryByExample(new
E1Persona(null,null));
         while(result.hasNext()){
            E1Persona p=result.next();
            System.out.println("Nombre: "+p.getNombre()+",
Ciudad:"+p.getCiudad());
         }
      }
      //CIERRA LA BASE DE DATOS
      db.close();
   }
}
```

Si lo que queremos es eliminar objetos, utilizamos el método delete, previa localización del mismo:

```
import com.db4o.Db4oEmbedded;
import com.db4o.ObjectContainer;
import com.db4o.ObjectSet;
public class E1ModificarDatos {
   final static String BDPer = "D:/db/db4o/DBE1Persona.yap";
   public static void main(String[] args) {
      ObjectContainer db=Db4oEmbedded.openFile(BDPer);

      //ELIMINAR LOS REGISTROS DE JUAN
      E1Persona perJuan = new E1Persona("Juan",null);
      ObjectSet<E1Persona> resultJuan = db.queryByExample(perJuan);
      if(resultJuan.size()==0) System.out.println("No existen Juan...");
      else{
         E1Persona existe = (E1Persona) resultJuan.next();
         System.out.println("Registros a borrar: "+resultJuan.size());
         if (resultJuan.size()>1){
            while(resultJuan.hasNext()){
               E1Persona p=resultJuan.next();
               db.delete(p);
               System.out.println("Borrado...");
            }
         }else db.delete(existe); //sólo hay un registro
      }
      //CIERRA LA BASE DE DATOS
      db.close();
```

```
    }
}
```

Protocolos de Acceso a Base de Datos

Existen dos normas de conexión a una Base de Datos SQL:

- ODBC (Open Database Connectivity): Define una API (Application Program Interface-Interfaz para Programas de Aplicación) que pueden usar las aplicaciones para abrir una conexión con una base de datos, enviar consultas, actualizaciones y obtener resultados. Las aplicaciones pueden usar esta API para conectarse a cualquier servidor de base de datos compatible con ODBC.
- JDBC (Java Database Connectivity-Conectividad de Base de Datos con Java): Define una API que pueden usar los programas Java para conectarse a los servidores de bases de datos relacionales.

Muchos de los orígenes de datos no son bases de datos relacionales, algunos puede que ni si quiera sean bases de datos (ficheros planos, almacenes de correos electrónicos)…

OLE-DB (Object Linking and Embedding for Databases-Enlace e incrustación de objetos para bases de datos) de Microsoft es una API de C++ con objetivos parecidos a los de ODBC pero para orígenes de datos que no son bases de datos. OLE-DB proporciona estructuras para la conexión con orígenes de datos, ejecución de comandos y devolución de resultados en forma de conjunto de filas. Sin embargo se diferencia de ODBC en algunos aspectos. Los programas OLE-DB pueden negociar con los orígenes de datos para averiguar las interfaces que soportan. En ODBC los comandos siempre están en SQL, en OLE-DB pueden estar en cualquier lenguaje soportado por el origen de datos, puede que algunos orígenes soporten SQL o un subconjunto limitado de SQL y otros ofrezcan el acceso a los datos de los ficheros planos sin ninguna capacidad de consulta.

La API ADO (Active Data Objects – Objetos activos de datos) creada por Microsfot ofrece una interfaz sencilla de utilizar con la funcionalidad OLE-DB, que puede llamarse desde los lenguajes como VBScript y JScript.

Acceso a Datos mediante ODBC

ODBC es un estándar de acceso a bases de datos desarrollado por Microsoft con el objetivo de posibilitar el acceso a cualquier dato desde cualquier aplicación, sin importar que sistema gestor de bases de datos almacene los datos. Cada sistema de base de datos compatible con ODBC proporciona una biblioteca que se debe enlazar con el programa cliente. Cuando el programa cliente realiza una llamada a la API ODBC el código de la biblioteca se comunica con el servidor para realizar la acción solicitada y obtener los resultados.

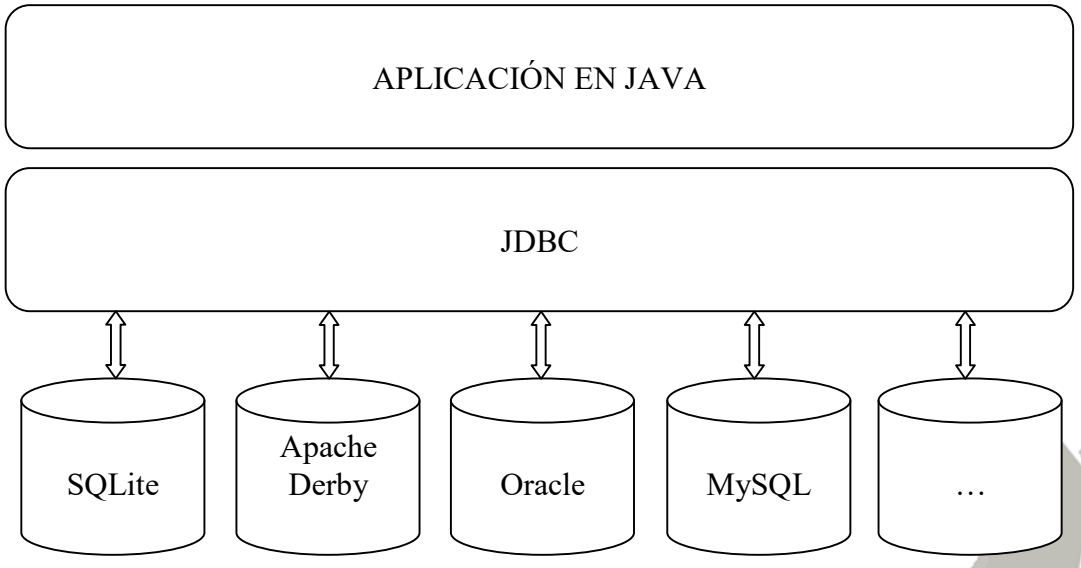
Pasos para usar un ODBC:

1. Configurar ODBC
   El programa asigna en primer lugar un entorno SQL con la función *SQLAllocHandle()*, después un manejador para la conexión a la base de datos basada en el entorno anterior, cuyo propósito es traducir las consultas de datos de la aplicación en comandos que el sistema de base de datos entienda. Estos manejadores son de varios tipos:
   - *SQLHENV*: define el entorno de acceso a los datos.
   - *SQLHDBC*: identifica el estado y configuración de la conexión (driver y origen de datos).
   - *SQLHSTMT*: declaración SQL y cualquier conjunto de resultados asociados.
   - *SQLHDESC*: recolección de metadatos utilizados para describir una sentencia SQL.
2. Abrir Conexión
   Usando *SQLDriverConnect*() o *SQLConnect*() y reservados los manejadores
3. Enviar órdenes SQL
   Mediante *SQLExecDirect()*
4. Desconectar la Base de Datos
   Desconectamos y liberamos la conexión y los manejadores del entorno SQL

La API ODBC usa una interface escrita en C y no es apropiada para su uso directo desde Java. Las llamadas desde Java a código C nativo tienen un número de inconvenientes en la seguridad, implementación, robustez y portabilidad de las aplicaciones. ODBC es difícil de aprender. Mezcla características elementales con otras más avanzadas y tiene complejas opciones incluso para las consultas más simples.

Acceso a Datos mediante JDBC

JDBC proporciona una librería estándar para acceder a fuentes de datos principalmente orientados a bases de datos relacionales que usan SQL. No sólo provee una interfaz sino que también define una arquitectura estándar, para que los fabricantes puedan crear los drivers que permitan a las aplicaciones Java el acceso a los datos. JDBC dispone de una interfaz distinta para cada base de datos, esto es conocido como driver (controlador o conector). Eso permite que las llamadas a los métodos Java de las clases JDBC se correspondan con el API de la base de datos.

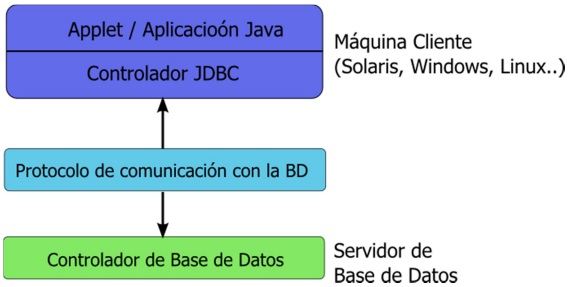


JDBC consta de un conjunto de clases e interfaces que nos permite escribir aplicaciones Java para gestionar las siguientes tareas con una base de datos relacional:

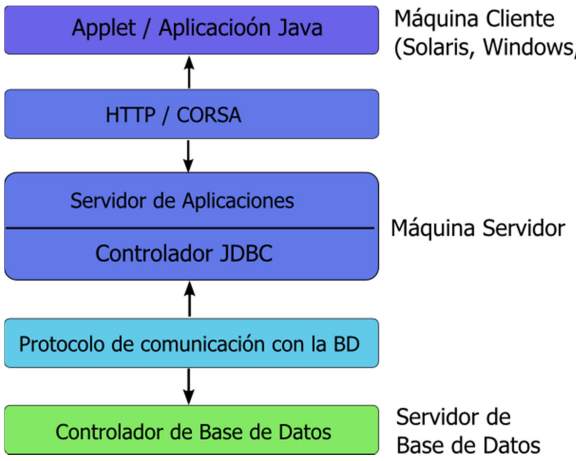
- Conectarse a la base de datos.
- Enviar consultas e instrucciones de actualización a la base de datos
- Recuperar y procesar los resultados recibidos de la base de datos en respuesta a consultas.

Modelos de acceso a base de datos

La API JDBC es compatible con los modelos tanto de dos como de tres capas para el acceso a la base de datos.



En el modelo de dos capas, un applet o una aplicación Java se comunican directamente con la base de datos, esto requiere un driver JDBC residiendo en el mismo lugar que la aplicación. Desde el programa Java se envían sentencias SQL al sistema gestor de la base de datos para que las procese y los resultados se envían de vuelta al programa. La base de datos puede encontrarse en otra máquina diferente a la de la aplicación y las solicitudes se hacen a través de la red (arquitectura cliente-servidor). El driver será el encargado de manejar la comunicación a través de la red de forma transparente al programa.



En el modelo tres capas, los comandos se envían a una capa intermedia que se encargará de enviar los comandos SQL a la base de datos y de recoger los resultados de la ejecución de las sentencias. Es decir, tenemos una aplicación corriendo en una máquina y accediendo a un driver de base de datos situado en otra máquina. En este caso los drivers no tienen que residir en la máquina cliente.

Un servidor de aplicaciones es una implementación de la especificación J2EE (Java Platform Enterprise Edition). J2EE es un entorno centrado en Java para desarrollar aplicaciones empresariales multicapa basadas en la web. Existen diversas implementaciones, cada una con sus propias características.

Tipos de drivers

Existen 4 tipos de conectores (drivers o controladores) JDBC:

- **JDBC-ODBC Bridge**: permite el acceso a bases de datos JDBC mediante un driver ODBC. Convierte las llamadas al API de JDBC en llamadas ODBC. Exige la instalación y configuración de ODBC en la máquina cliente.
- **Native**: controlador escrito parcialmente en Java y en código nativo de la base de datos. Traduce las llamadas al API de JDBC Java en llamadas

propias del motor de base de datos. Exige instalar en la máquina cliente código binario propio del cliente de base de datos y del sistema operativo.

- **Network**: controlador de Java puro que utiliza un protocolo de red (por ejemplo HTTP) para comunicarse con el servidor de base de datos. Traduce las llamadas al API de JDBC Java en llamadas propias del protocolo de red independiente de la base de datos y a continuación son traducidas por un software intermedio al protocolo usado por el motor de la base de datos. El driver JDB no comunica directamente con la base de datos, comunica con el software intermedio, que a su vez comunica con la base de datos. Son útiles para aplicaciones que necesitan interactuar con la base de datos, ya que usan el mismo driver JDBC sin importar la base de datos específica. No exige instalación en cliente.
- **Thin**: controlador de Java puro con protocolo nativo. Traduce las llamadas al API de JDBC Java en llamadas propias del protocolo de red usado pro el motor de base de datos. No exige instalación del cliente.

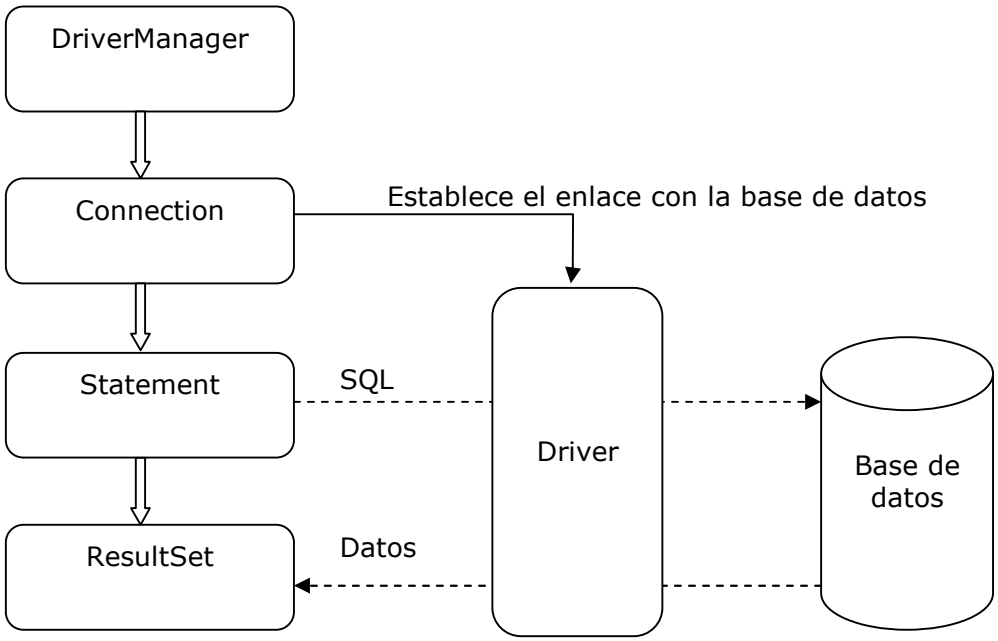
Sin duda alguna, los dos últimos tipos son la mejor forma para acceder a bases de datos JDBC. Y de éstas, en la mayoría de los casos, la mejor de todas es la última opción.

Cómo funciona JDBC

La API JDBC es compatible con los modelos tanto de dos como de tres capas para el acceso a la base de datos.

JDBC define varias interfaces que permite realizar operaciones con bases de datos; a partir de ellas se derivan las clases correspondientes. Éstas están definidas en el paquete java.sql. La siguiente tabla muestra las clases e interfaces más importantes:

CLASE E INTERFAZ	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.
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
CallableStatement	Permite ejecutar sentencias SQL con parámetros de entrada y salida, como llamadas a procedimientos almacenados
ResultSet	Contiene las filas resultantes de ejecutar una orden SELECT
ResultSetMetadata	Permite obtener información sobre un ResultSet, como el número de columnas, sus nombres, etc.



El funcionamiento de un programa con JDBC requiere los siguientes pasos:

1. Importar las clases necesarias.
2. Cargar el driver JDBC
3. Identificar el origen de datos
4. Crear un objeto Connection
5. Crear un objeto Statement
6. Ejecutar una consulta con el objeto Statement
7. Recuperar los datos del objeto ResultSet
8. Liberar el objeto ResultSet
9. Liberar el objeto Statement
10. Liberar el objeto Connection

Ejemplo práctico

Base de datos ejemplo en MySQL (Usuario: root, sin clave de usuario)

```
import java.sql.*;

public class E2MySQL {

   public static void main(String[] args) {

      try{
         //cargar el driver
         Class.forName("com.mysql.jdbc.Driver");

         //establecemos la conexion con la BD
```

```
        Connection conexion = DriverManager.getConnection("jdbc:mysql://localhost/ejemplo","root","");

        //preparamos la consulta
        Statement sentencia = conexion.createStatement();
        String sql = "SELECT * FROM departamentos";
        ResultSet resul = sentencia.executeQuery(sql);

        //Recorremos el resultado para visualizar cada fila
        //Se hace un  buclee mientras haya regisros y se van mostrando
        while (resul.next()){
           System.out.printf("%d, %s, %s, %n",
                resul.getInt(1),
                resul.getString(2),
                resul.getString(3));
        }

        resul.close(); //Cerrar ResultSet
        sentencia.close(); //Cerrar Statement
        conexion.close(); //Cerrar conexión

     }catch (ClassNotFoundException cn){
        cn.printStackTrace();
     }catch (SQLException e){
        e.printStackTrace();
     }
   }
}
```

Para poder probar el programa hemos de obtener el JAR que contiene el driver MySQL (en este ejemplo se ha utilizado el com.mysql.jdbc_5.1.5.jar), además hemos instalado el MySQL Workbench para la gestión de la base de datos y el servidor)

Para refrescar la memoria sobre MySQL, aquí podrás ver unos vídeos bastante simples sobre MySQL Workbench:

CREAR BASE DE DATOS CON WORKBENCH

https://www.youtube.com/watch?v=FXCzcd5PNiI

CREAR TABLAS CON WORKBENCH

https://www.youtube.com/watch?v=luMXHjgHMPc

CREAR TABLAS CON WORKBENCH DE FORMA GRÁFICA

https://www.youtube.com/watch?v=hfE0_Mme32k

Analicemos el código anterior:

Con el método forName() de la clase Class, **cargamos el driver**, que en esta ocasión es el com.mysql.jdbc.Driver

A continuación se **establece la conexión** con la base de datos (debiendo estar arrancado el servidor MySQL), para ello usamos la clase DriverManager con el método getConnection(). Podemos observar que al método getConnection le debemos pasar por parámetros la url de la base de datos, el usuario y el password. La url tiene el formato ***jdbc:mysql://nombre_host:puerto/nombre_basedatos***, donde:

- **jdbc:mysql** indica que estamos utilizando un driver JDBC para MySQL.
- **nombre_host** indica el nombre del servidor donde está la base de datos. Puede ser una IP o un nombre de máquina que esté en la red.
- **puerto** es el puerto predeterminado para las bases datos MySQL, por defecto es 3306. De no indicarse se asume este valor.
- **nombre_basedatos** es el nombre de la base de datos a la que nos vamos a conectar y que debe existir en MySQL.

A continuación debemos **realizar la consulta**, para ello utilizamos la interfaz Statement para crear la sentencia. Para ello debemos acceder al método createStatement() de un objeto Connection válido. La objeto obtenido (sentencia) tiene el método executeQuery() que sirve para realizar una consulta a la base de datos, pasándole un String en el que esta la consulta SQL.

El resultado de la ejecución de la sentencia nos devuelve un ResultSet, que es un objeto a modo de lista en la que está el resultado de la misma. Cada elemento de la lista es uno de los registros de la tabla. ResultSet no contiene todos los datos, sino que los va recopilando de la base de datos según se va pidiendo. Por ello, el método executeQuery() puede tardar poco, aunque recorre los elementos del ResultSet puede no serlo.

Para recorrer la lista de resultados usamos el puntero interno que posee Resultset, avanzando con el método next (resul.next).

getInt() y getString() nos devuelve los valores de los campos del registro. Le indicamos entre paréntesis la posición de la columna en la tabla (el nombre de la columna)

ResultSet tiene métodos para mover el puntero del objeto ResultSet como hemos visto con next(), pero también existen otros como first(), last(), previous(), beforeFirst() y getRow().

Por último **liberamos los recursos** y **cerramos la conexión** con los close() correspondientes.

Establecimiento de Conexiones

Vamos a conectarnos a otras bases de datos embebidas comentadas anteriormente como son SQLite o Access.

Conexión a SQLite

La librería que necesitamos en esta ocasión es la sqlite-jdbc-3.8.11.2.jar y podemos descargarla desde la URL https://bitbucket.org/xerial/sqlite-jdbc/downloads, que deberemos instalar en nuestro IDE.

En el programa java deberemos cargar el driver y realizar la conexión con la base de datos de la siguiente forma:

```
Class.forName("org.sqlite.JDBC");
Conection conexión=DriverManager.getConection("jdbc:sqlite:D:/db/SQLITE/ejemplo.db")
```

Conexión a Access

Las librerías que necesitamos en esta ocasión son: Commons-lang-2.6.jar, Commons-logging-1.1.1.jar, hsqldb.jar, jackcess-2.1.2.jar, ucanaccess-3.0.2.jar y ucanload.jar. y podemos descargarla desde la url:

https://sourceforge.net/projects/ucanaccess/files/, de donde nos descargaremos el fichero UcanAccess-30.4-bin.zip que contiene los .jar anteriores, los cuales deberemos instalar en nuestro IDE.

Par conectarnos, introduciremos el siguiente códio:

```
Class.forName("net.ucanaccess.jdbc.UcanaccessDriver");
Connection con = DriverManager.getConnection("jdbc:ucanaccess://mibasedatosaccess");
```

Por ejemplo:

```
Connection con = DriverManager.getConnection("jdbc:ucanaccess://./ejemplo.accdb")
```

Ejecución de Sentencias de Descripción de Datos

Es normal que cuando desarrollemos nuestra aplicación JDBC conozcamos la estructura de las tablas que componen nuestra base de datos. Pero es posible que esto no sea así en alguna ocasión. Para solucionar este problema, la información sobre la base de datos se puede obtener a través de los metaobjetos, que no son más que objetos que proporcionan información sobre la base de datos.

La interfaz DatabaseMetaData proporciona información sobre la base de datos a través de múltiples métodos de los cuales es posible obtener gran cantidad de información.

Para obtener más información sobre la clase DatabaseMetaData puedes acceder a:

https://docs.oracle.com/javase/8/docs/api/java/sql/DatabaseMetaData.html

Ejecución de Sentencias de Manipulación de Datos

Mediante la interfaz Statement accedemos a métodos para ejecutar sentencias SQL y así poder obtener resultados. De esta forma podemos usar los siguientes métodos:

- ResultSet executeQuery(String): utilizado para sentencias SQL que recuperan datos de un único objeto ResulSet, se utiliza para las sentencias SELECT.
- int executeUpdate(String): utilizado para sentencias que no devuelven un ResultSet como las sentencias de manipulación de datos (DML): INSERT, UPDATE y DELETE; y las sentencias de definición de datos (DDL): CREATE, DROP y ALTER. Éste método devuelve un int indicando el número de filas que fueron afectadas y en el caso de las DDL devuelve un 0.
- Boolean execute(String): se puede utilizar para ejecutar cualquier sentencia SQL. Devuelve true si se obtiene un ResultSet (para recuperar las filas será necesario llamar al método getResultSet()) y false se se trata de un recuento de actualizaciones o no hay resultados; en este caso se usara el método getUpdateCount() para recuperar el valor devuelto.

Veamos un ejemplo a continuación:

```
import java.sql.*;

public class E3EjemploExecute {
```

```
  public static void main(String[] args) throws ClassNotFoundException,
SQLException {
     //CONEXIÓN A MYSQL
     Class.forName("com.mysql.jdbc.Driver");
     Connection conexion =
DriverManager.getConnection("jdbc:mysql://localhost/ejemplo","root","");

     String sql="SELECT * FROM departamentos";
     Statement sentencia = conexion.createStatement();
     boolean valor = sentencia.execute(sql);

     if (valor){
        ResultSet rs=sentencia.getResultSet();
        while (rs.next()) {
           System.out.printf("%d, %s, %s, %n", rs.getInt(1),
rs.getString(2), rs.getString(3));
        }
        rs.close();
     }else{
        int f=sentencia.getUpdateCount();
        System.out.printf("Filas afectadas: %d %n",f);
     }
     sentencia.close();
     conexion.close();
  }
}
```

Si cambiamos la orden SQL por otra como `String sql= "UPDATE departamentos SET dnombre=LOWER(dnombre)";` la variable valor será false y la salida del programa será diferente.

A través de un objeto ResultSet se puede acceder al valor de cualquier columna de la fila actual por nombre o posición.

```
import java.sql.*;

public class E3EjemploExecute {

  public static void main(String[] args) throws ClassNotFoundException,
SQLException {
     //CONEXIÓN A MYSQL
     Class.forName("com.mysql.jdbc.Driver");
     Connection conexion =
DriverManager.getConnection("jdbc:mysql://localhost/ejemplo","root","");

     String sql="SELECT * FROM departamentos";
     String sql="UPDATE departamentos SET dnombre=LOWER(dnombre)" ;
     Statement sentencia = conexion.createStatement();
     boolean valor = sentencia.execute(sql);

     if (valor){
        ResultSet rs=sentencia.getResultSet();
        while (rs.next()) System.out.printf("%d, %s, %s, %n",
rs.getInt(1), rs.getString(2), rs.getString(3));
        rs.close();
```

```
      }else{
         int f=sentencia.getUpdateCount();
         System.out.printf("Filas afectadas: %d %n",f);
      }
      sentencia.close();
      conexion.close();
   }
}
```

Otro ejemplo podría ser actualizar el salario:

```
import java.sql.*;

public class E5ModificarSalario {

   public static void main(String[] args) {
      try {
         Class.forName("com.mysql.jdbc.Driver"); //Cargar el driver
         //Establecer conexión con la base de datos
         Connection conexion =
DriverManager.getConnection("jdbc:mysql://localhost/ejemplo","root","");
         //Datos a insertar
         String dep = "15";
         String subida="100";
         //construir la orden SQL
         String sql=String.format("UPDATE empleados SET salario = salario +
%s WHERE dept_no=%s",subida,dep);
         System.out.println(sql);

         Statement sentencia = conexion.createStatement();
         int filas=sentencia.executeUpdate(sql);
         System.out.printf("Empleados modificados: %d %n", filas);

         sentencia.close();
         conexion.close();
      }catch (ClassNotFoundException cn){
         cn.printStackTrace();
      }catch (SQLException e){
         e.printStackTrace();
      }
   }
}
```

Y por último vamos a crear una vista llamada totales que contiene por cada departamento el número de departamento,, el nombre, el número de empleados que tiene y el salario media:

```
import java.sql.*;

public class E6CrearVista {

   public static void main(String[] args) {
      try {
         Class.forName("com.mysql.jdbc.Driver"); //Cargar el driver
         //Establecer conexión con la base de datos
         Connection conexion =
DriverManager.getConnection("jdbc:mysql://localhost/ejemplo","root","");
```

```
        //Create View
        StringBuilder sql=new StringBuilder();
        sql.append("CREATE OR REPLACE VIEW totales ");
        sql.append("(dep, dnombre, nemp, media) AS ");
        sql.append("SELECT d.dept_no, dnombre, COUNT(emp_no), AVG(salario)
");
        sql.append("FROM departamentos d LEFT JOIN empleados e " );
        sql.append("ON e.dept_no = d.dept_no ");
        sql.append("GROUP BY d.dept_no, dnombre ");
        System.out.println(sql);

        Statement sentencia = conexion.createStatement();
        int filas=sentencia.executeUpdate(sql.toString());
        System.out.printf("Resultados de la ejecución: %d %n", filas);

        sentencia.close();
        conexion.close();
      }catch (ClassNotFoundException cn){
        cn.printStackTrace();
      }catch (SQLException e){
        e.printStackTrace();
      }
    }
}
```

Ejercicios:

1. Crea un programa en java para el mantenimiento de una base de datos en Db4o de nombre EMPLEDEP.YAP.

La aplicación mediante un menú debe ser capaz de insertar registros, eliminar y actualizarlos.

El contenido de la base de datos son objetos que tienen EMPLEADOS Y DEPARTAMENTOS.

2. Crea una base de datos en MySQL WorkBench con tres campos, introducir cinco registros y crear una aplicación java mediante JDBC que recorra todos sus registros visualizando su contenido.

3. Crea un programa java que introduzca tres registros en la tabla empleados. Los datos a insertar son (emp_no, apellido, profesión, director, salario, comisión y dept_no). Antes de insertar se deben realizar las siguientes comprobaciones:

- Que el departamento exista en la tabla departamentos, si no existe no se inserta
- que el número de empleado no exista, si existe no se inserta
- que el salario sea mayor que 0, en caso contrario no se inserta
- que el director exista en la tabla empleados, si no existe no se inserta (dir es el número de empleado de su director)
- El apellido y el oficio no pueden ser nulos.
- La fecha de alta del empleado es la fecha actual

Cuando se inserte la fila visualizar mensaje y si no se inserta visualizar el motivo (departamento inexistente, numero de empleado duplicado, director inexistente, etc.

4. Crea una Base de datos en MySQL de nombre UNIDAD2, con las tablas siguientes:

PRODUCTOS:

- ID numérico, Clave Primaria
- DESCRIPCION varchar(50), no nulo
- STOCKACTUAL numérico
- STOCKMINIMO numérico
- PVP numérico

CLIENTES:

- ID numérico, clave primaria
- NOMBRE varchar(50), no nulo
- DIRECCIÓN varchar(50)
- POBLACIÓN varchar(50)
- TELEF varchar(20)
- NIF varchar(10)

VENTAS:

- IDVENTA numérico, clave primaria
- FECHA VENTA no nulo
- IDCLIENTE numérico, Cclave ajena a CLIENTES
- IDPRODUCTO numérico, clave ajena a PRODUCTOS
- CANTIDAD numérico
- *Un cliente puede tener muchas ventas*

Una vez creadas, haz un programa Java que llene las tablas PRODUCTOS y CLIENTES (los datos a insertar se definen e el propio programa).

Una vez rellenas, visualiza los datos insertados y el número de filas que se han insertado en cada tabla.

Se pueden crear las tablas y métodos que se crean convenientes.

5.A partir de las tablas creadas en el ejercicio anterior, realizar un programa Java para insertar ventas en la tabla VENTAS.

Realizar las siguientes comprobaciones antes de insertar la venta en la tabla:

- El identificador de venta no debe existir en la tabla VENTAS
- El identificador de ciente debe existir en la tabla CLIENTES
- El identificador de producto debe existir en la tabla PRODUCTOSS
- La cantidad debe ser mayor que 0

La fecha de venta es la fecha actual.

Una vez insertada la fila en la tabla visualizar un mensaje indicándolo. Si no se ha podido insertar, visualizar el motivo (no existe cliente, no existe producto, cantidad < o igual a 0, …)

6.Realizar un listado de las ventas de un cliente, introducido por teclado. El programa deberá mostrar:

Ventas del cliente: *nombre de cliente*
Venta: *idVenta*
Producto: *descripción* – PVP: *pvp*
Cantidad: *cantidad*
Importe: *cantidad*pvp*
Venta: *idVenta*
Producto: *descripción* – PVP: *pvp*
Cantidad: *cantidad*
Importe: *cantidad*pvp*
…….

Número total de ventas: *ntotalventas*
Importe total: *totalimporte*

7.Repetir los ejercicios 4, 5 y 6 para la base de datos SQLite

8.Repetir los ejercicios 4, 5 y 6 para la base de datos DB4O