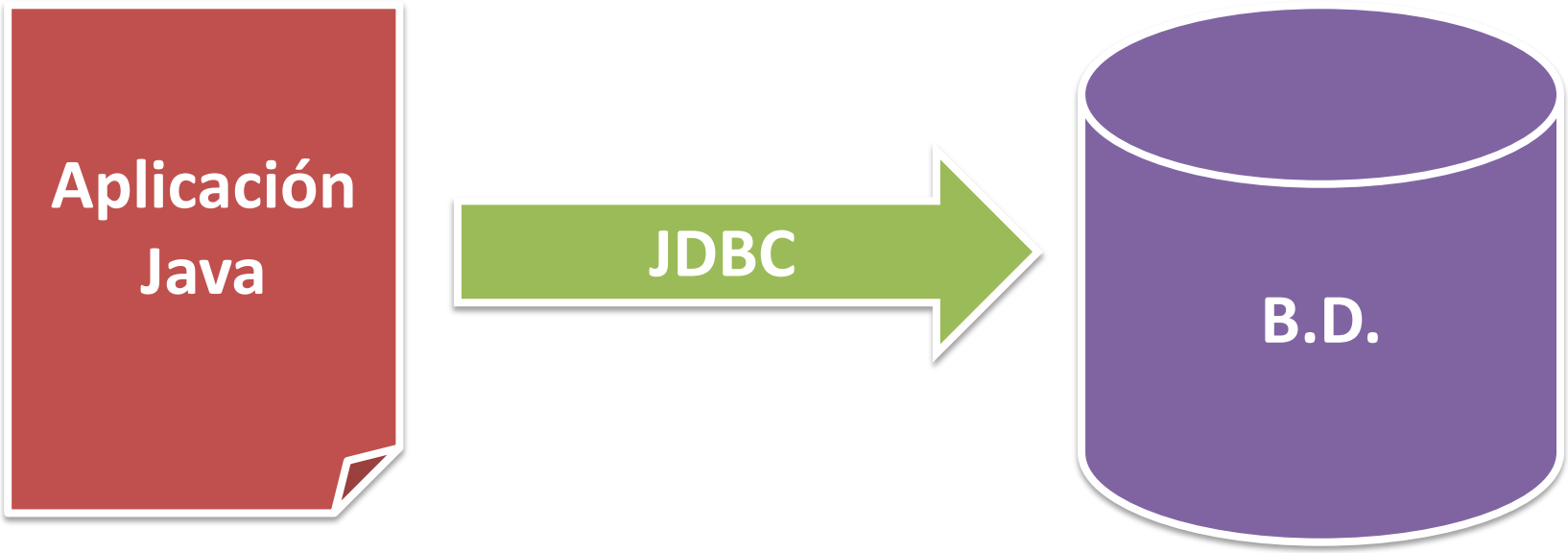
Desarrollo de Aplicaciones Multiplataforma

PROGRAMACIÓN

JDBC (Java Database Connectivity) es un conjunto de clases e interfaces escritos en Java que ofrecen una API completa para la programación de bases de datos de diferentes proveedores (Microsoft SQL Server, Oracle, MySQL, Interbase, Microsoft Access, IBM DB2, PostgreSQL, etc...) usando instrucciones SQL.



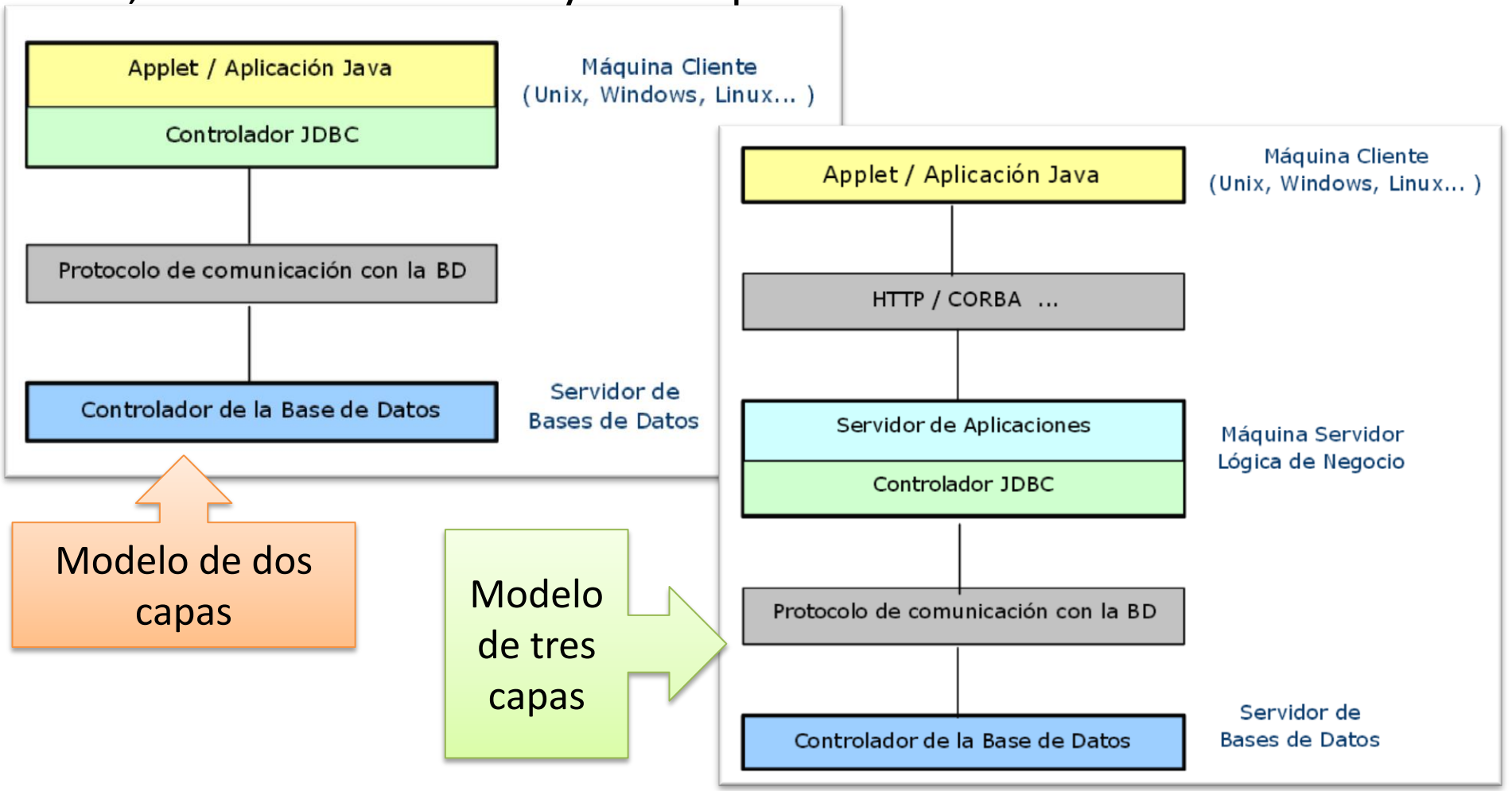
En las tecnologías Java podemos encontrarnos dos normas de conexión a una B.D. SQL.

- **ODBC** (**O**pen **D**ata**b**ase **C**ontivity) define una API que pueden utilizar las aplicaciones para abrir una conexión con una BD.

- **JDBC** (**J**ava **D**ata**b**ase **C**ontivity)define una API que pueden utilizar los programas para conectarse a los sistemas de B.D. relacionales

El paquete actual de JDK incluye JDBC y el puente JDBC-ODBC. 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

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



Java suministra tres componentes JDBC como parte del JDK:

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

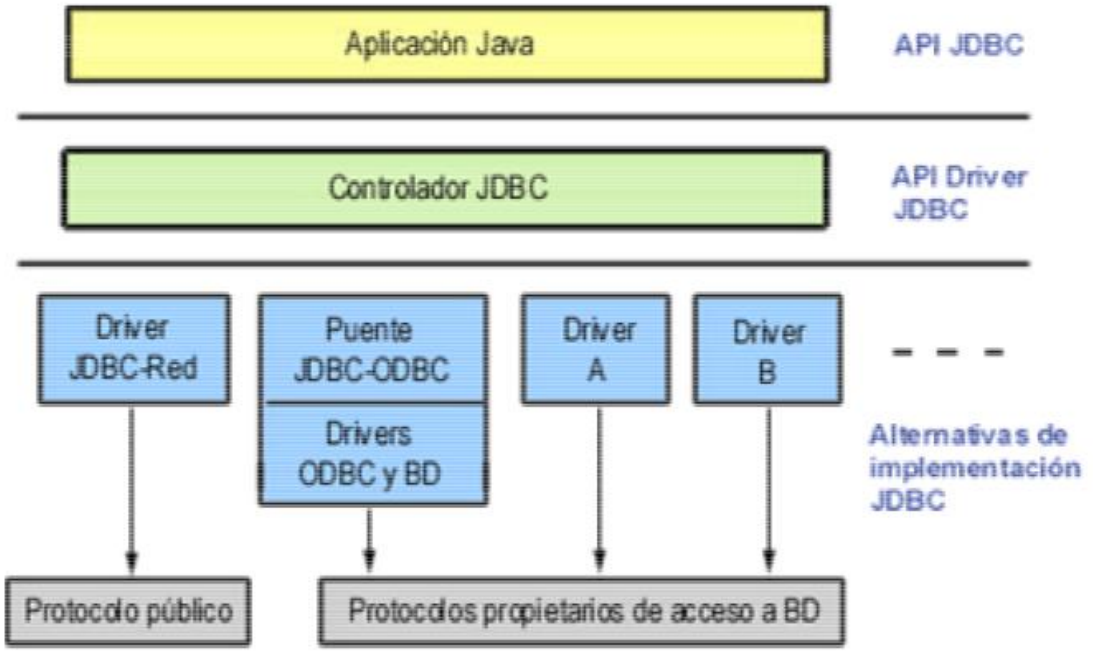
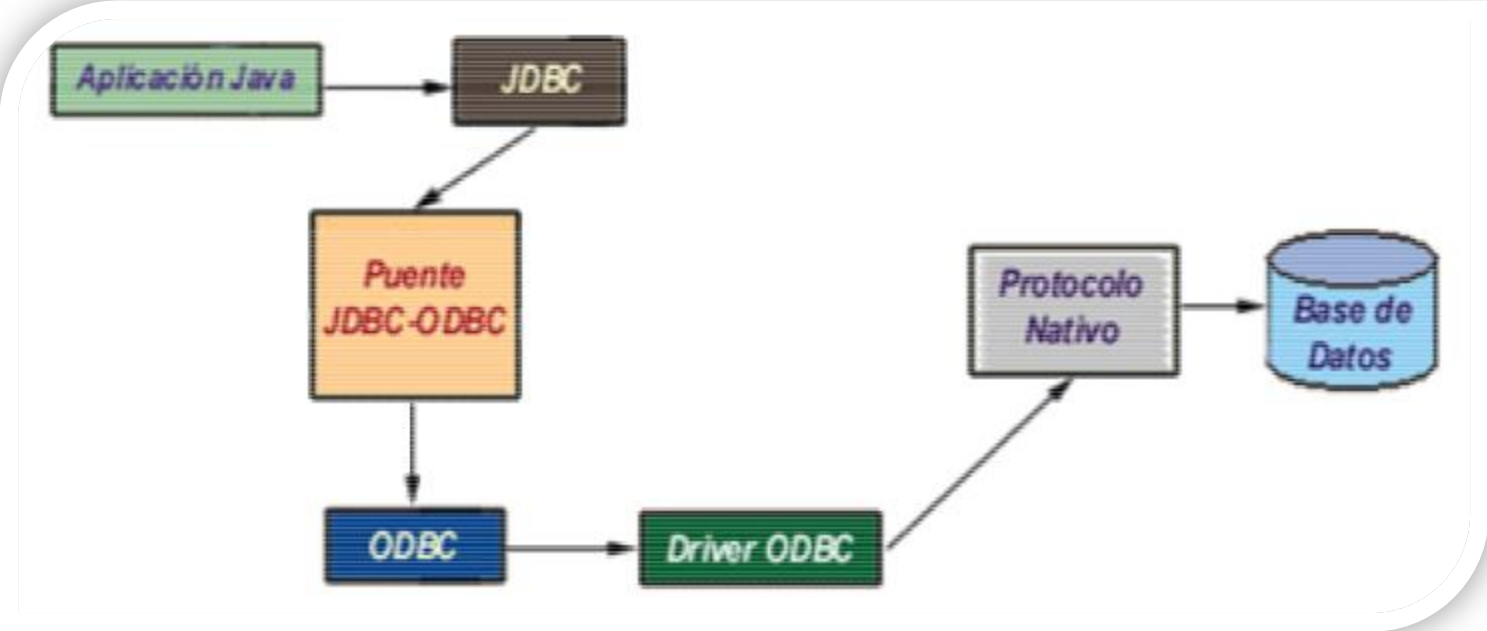


Figura 12.1: Modelo de capas para la conexión a bases de datos

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

- **Puente JDBC-ODBC**

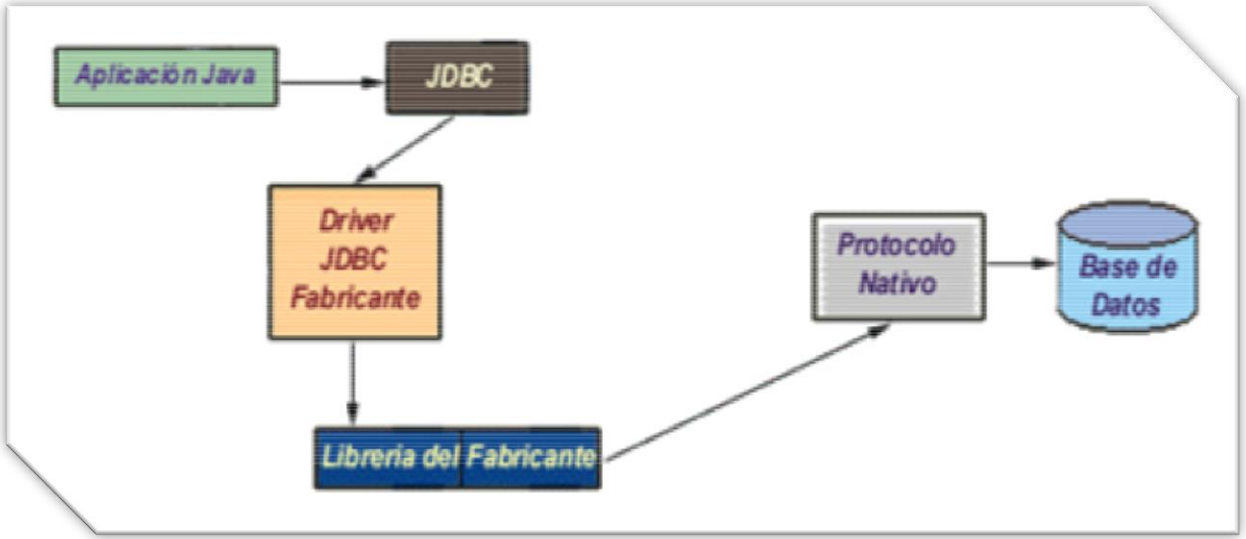
Este driver convierte todas las llamadas JDBC a llamadas ODBC y realiza la conversión correspondiente de los resultados.



- **Java parcialmente Nativo (Java/Binario)**

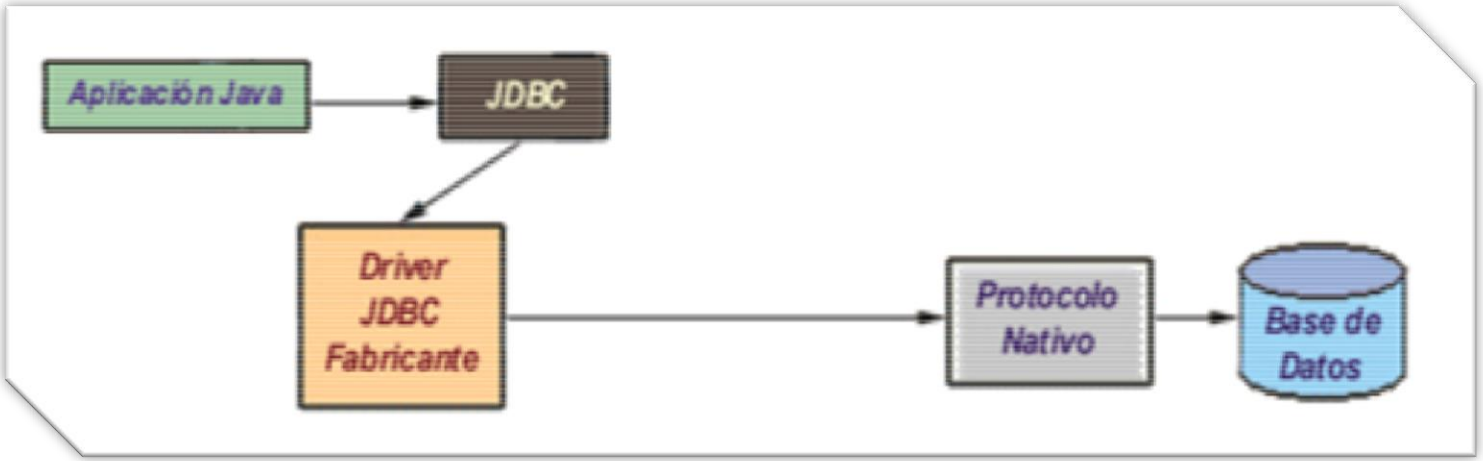
Este tipo de driver convierte llamadas JDBC en llamadas del API cliente para Oracle, Sybase, Informix, DB2 y otros DBMS.

Este driver se salta la capa ODBC y habla directamente con la librería nativa del fabricante del sistema DBMS. Este driver es un driver 100% Java pero aún así necesita la existencia de un código binario (la libre ría DBMS) en la máquina del cliente, con las limitaciones y problemas que esto implica.



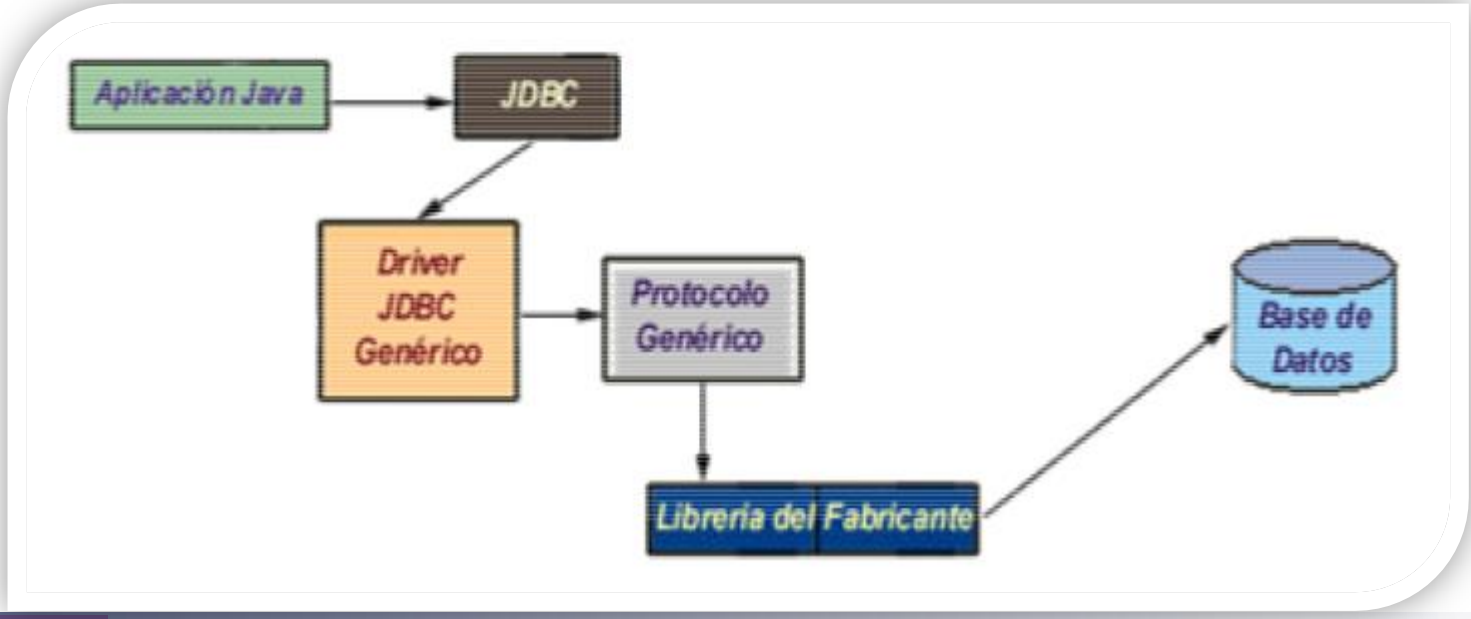
- **Java nativo JDBC-Net**

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.



- **Java nativo JDBC-Independiente**

Esta es la opción más flexible. 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.



El trabajo con JDBC comienza con la clase **DriverManager** que es la encargada de establecer las conexiones con los orígenes de datos a través de los drivers JDBC.

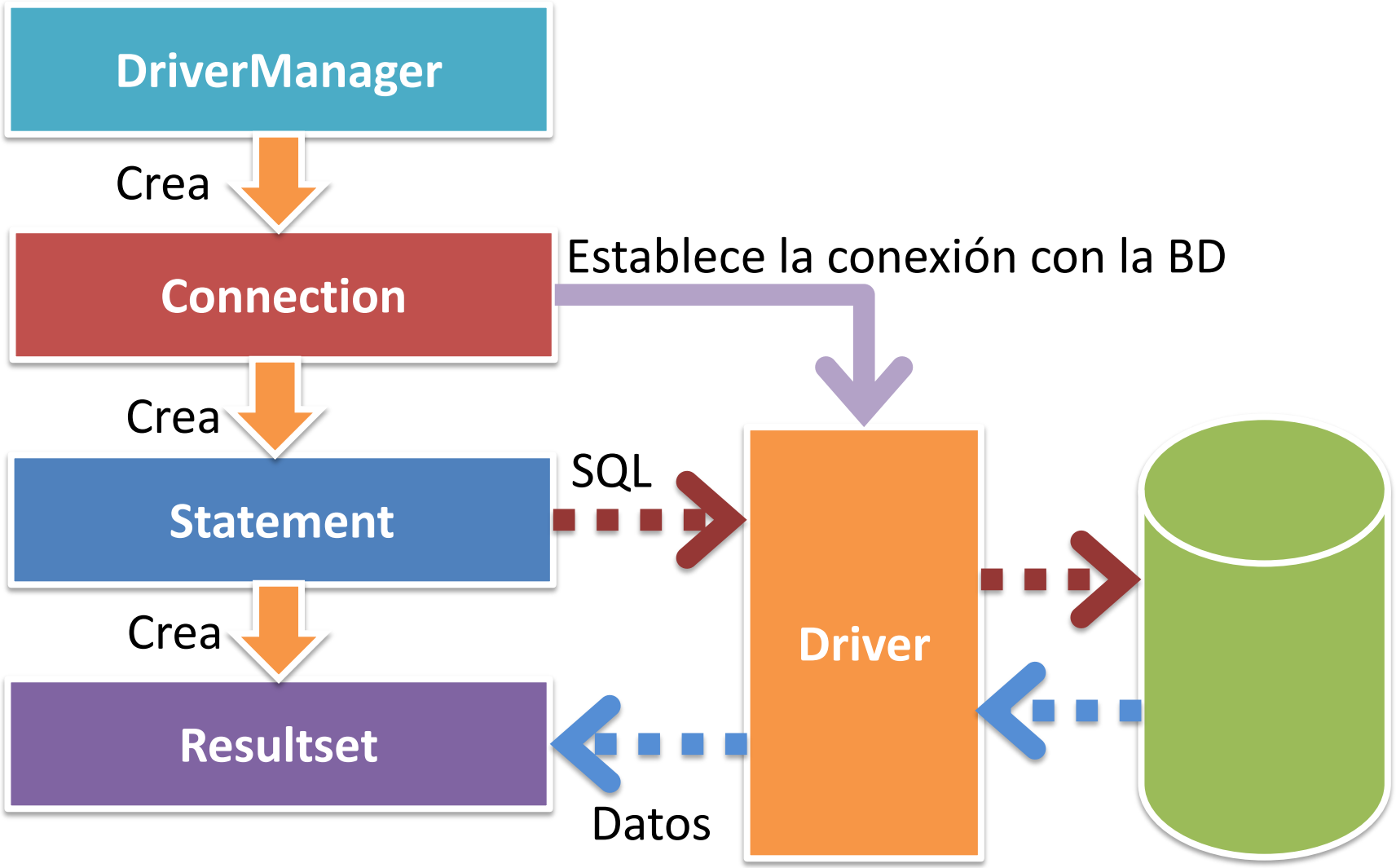
El funcionamiento de un programa con JDBC requiere los siguientes pasos.

1º Importar las clases necesarias	**6º Ejecutar una consulta con el objeto Statement**
2º Cargar el driver JDBC	**7º Recuperar datos con Resultset**
3º Identificar el Origen de Datos	**8º Liberar el objeto Resulset**
4º Crear un objeto Connection	**9º Liberar el objeto Statement**
5º Crear un objeto Statement	**10º Liberar el objeto Connection**

Como resumen de lo anterior, JDBC realiza varias funciones:

- **Conecta** con la base de datos; la BD puede ser local o remota
- **Envía** las sentencias SQL
- **Procesa** los resultados obtenidos

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


DriverManager
Crea
Connection
Establece la conexión con la BD
Crea
Statement
SQL
Crea
Resultset
Driver
Datos

Conexión	• DriverManager • Connection • DatabaseMetadata
Comandos	• Statement • CallableStatement • PreparedStatement
Resultados	• ResultSet • ResulsetMetadata

Lo primero que tenemos que hacer es establecer una conexión con el controlador de base de datos que queremos utilizar. Esto implica dos pasos:

- Cargar el driver que se quiere utilizar
- Realizar la conexión.

Cargar los Drivers

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

Class.forName("jdbc.DriverX");
```

Esta es la clase que nos permite establecer las conexiones

```
Connection conexion = DriverManager.getConnection( Url ) ;
```

Por defecto, las transacciones sobre la BD se realizan inmediatamente, no se pueden deshacer los cambios (auto commit)

conexion.setAutoCommit(true)	Activa el auto commit
conexion.setAutoCommit(false)	Desactiva el auto commit
conexion.commit()	Graba los cambios
conexion.roolback()	Deshace los cambios
conexion.getCommit()	Obtiene el auto commit

Permite crear el comando SQL sobre la BD

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

Ejecución de los comandos:

- **int executeUpdate(sql)**. Ejecuta un comando y devuelve el nº de registros afectados (update, insert, delete)

```
String sql = "insert ….);
int num = stmt.executeUpdate(sql);
```

- **int executeQuery(sql)**. Devuelve un conjunto de resultados (Resulset)

```
String sql = "insert ….);
int num = stmt.executeUpdate(sql);
```

- Tiene el mismo comportamiento que un cursor.
- Define los métodos que permiten acceder al cursor generado como resultado de un Select.
- Inicialmente el puntero está posicionado en la primera fila. Para moverse entre registros se emplean los métodos nextXXX() (rs.next())
- Para obtener un campo de un registro se invoca el método getXXX (rs.getInt()). Existen métodos get para distintos tipos de datos.

ResultSet rs= smtpexecuteQuery(sql) ;

```
ResultSet rs = stmt.executeQuery("SELECT dni, edad, sueldo FROM emple");
while (rs.next()) {
        String dni = rs.getString(“dni");
        int edad =rs.getInt(“edad");
        float sueldo = rs.getFloat(“sueldo");
}
```

Es una especialización de Statement que permite definir sentencias parametrizadas. Agiliza la ejecución al realizar una precompilación en la creación. Los parámetros se especifican por posición utilizando los métodos setXXX.

```
PreparedStatement pstmt= conexion.prepareStatement( sql) ;
```

Ejemplo:

```
String sql="SELECT dni,nombre FROM usuarios WHERE grupo=?";
PreparedStatement pstmt= conexion.prepareStatement( sql) ;
pstmt.setString(1,"DAM");
Resulset rs = pstmt.executeQuery();
```

clearParameters() borra los parámetros.

Es el modo estandar de llamar a procedimientos almacenados. Los procedimientos almacenados son un conjunto de sentencias sql junto con un lenguaje procedimental específico del SGBD. Se pueden invocar por su nombre para llevar a cabo alguna tarea en la B.D.

Pueden definirse con parámetros de entrada (IN), salida (OUT), de entrada/salida (INOUT) o sin ningún parámetro. También pueden devolver un valor, en este se trataría de una función.

Las técnicas para desarrollar procedimientos almacenados dependen del SGBD.

Ejemplo de procedimiento almacenado en MYSQL

```
DELIMITER  //
CREATE PROCEDURE aumenta.precio (cod int, p float)
      BEGIN
            UPDATE articulos SET precio = precio + precio * p
                  WHERE codigo=cod;
            COMMIT;
      END;
//
```

Llamada desde Java

```
String sql =“{call aumenta.precio (?,?)}”;
CallableStatement procAumentaPrecio= conexión.prepareCall(sql);
procAumentaPrecio.setInt(1,20);              // codigo 20
procAumentaPrecio.setFLoat(2,0.15);          // incremento 15%
procAumentaPrecio. executeUpdate();
procAumentaPrecio.close();
```

Hay cuatro formas de llamar a un procedimiento almacenado:

- **{ call procedimiento }**
- **{ ? = call función }**
- **{ call procedimiento(?,?,…) }**
- **{ ? = call función (?,?,…) }**

El usuario debe tener derecho para ejecutar procedimientos

GRANT SELECT ON mysql.proc TO usuario

También se debe incluir en la conexión el parámetro **noAccessToProcedureBodies** con el valor **true**

jdbc:mysql://ip/bd?noAccessToProcedureBodies =true

En ocasiones necesitamos hacer aplicaciones en Java que trabajen contra una base de datos desconocida, de la que no sabemos qué tablas tiene, ni qué columnas cada tabla.

En java, las clases **DataBaseMetaData** y **ResultSetMetaData** permiten, respectivamente, analizar la estructura de una base de datos (qué tablas tiene, que columnas cada tabla, de qué tipos, etc) o de un ResultSet de una consulta, para averiguar cuántas columnas tiene dicho ResulSet, de qué columnas de base de datos proceden, de qué tipo son, etc.

Los métodos de **ResultSetMetaData** nos permiten determinar las características de un objeto ResultSet. Así, podemos determinar:

- El número de columnas.
- Información sobre una columna, tal como el tipo de datos, la longitud, la precisión y la posibilidad de contener nulos.
- La indicación de si una columna es de solo lectura, etc.

```
ResultSetMetaData rsmd = rs.getMetadata();
```

Algunos de los métodos que puede utilizar sobre un objeto ResultSetMetaData son: getTableName(int colum), getColumnCount(), getColumnName(int colum), getColumnTypeName(int colum), getPrecision(int colum)

La clase **DataBaseMetaData** nos permite acceder a catálogos, esquemas, tablas, tipos de tablas, columnas de las tablas, etc. de una base de datos. Una vez establecida la conexión, podemos obtener la instancia correspondiente de dicha clase con el método **getDataBaseMetaData()** de la Connection.

```
Connection conexion = …
DataBaseMetaData dbmd = conexion.getMetaData();
```

Algunos de los métodos que podemos utilizar son:

```
ResultSet rs = dbmd.getTables(null, null, "%", null);
ResultSet rs = dbmdmetaDatos.getColumns(catalogo, null, tabla, null);
```

Podemos consultar las tablas del catálogo con **getTables()**:

```
ResultSet rs = dbmd.getTables(null, null, "%", null);
```

donde los cuatro parámetros que hemos pasado son:

- **Catálogo** de la base de datos. Al poner *null*, estamos preguntando por el catálogo actual.
- **Esquema** de la base de datos. Al poner *null*, es el actual.
- **Patrón** para las tablas en las que tenemos interés (% es el comidín, ej us%).
- El cuarto parámetro es un array de *String*, en el que pondríamos qué **tipos de tablas** queremos (normales, vistas, etc). Al poner *null*, nos devolverá todos los tipos de tablas.

Podemos consultar las columnas de una tabla con **getColumns()**:

```
ResultSet rs = dbmd.getTables(null, null, "%", null);
```

donde los cuatro parámetros que hemos pasado son:

- El nombre del **catálogo** al que pertenece la tabla.
- El nombre del **esquema**, *null* para el esquema actual.
- El nombre de la **tabla**. Nuevamente podríamos poner comodines al estilo *SQL* .
- El nombre de las **columnas** buscadas, usando comodines. *null* nos devuelve todas las columnas.