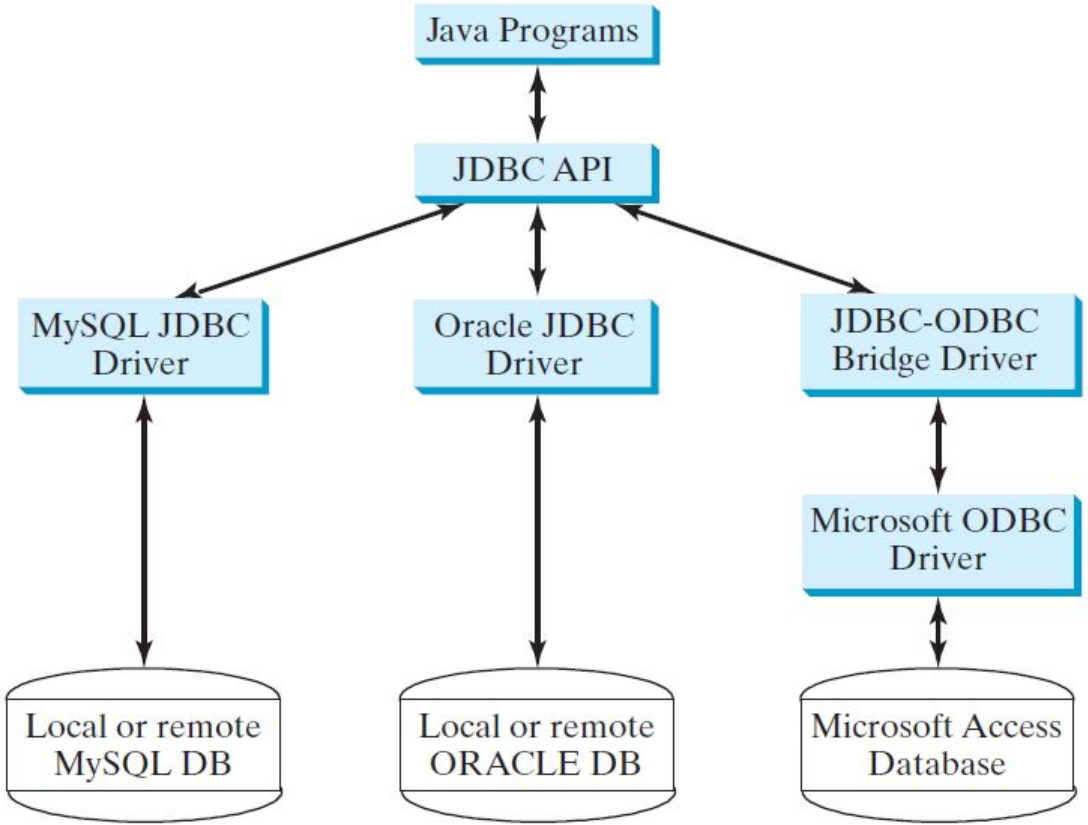
Capítulo 16
JDBC
Acceso a datos

¿Por qué Java para la programación de bases de datos?

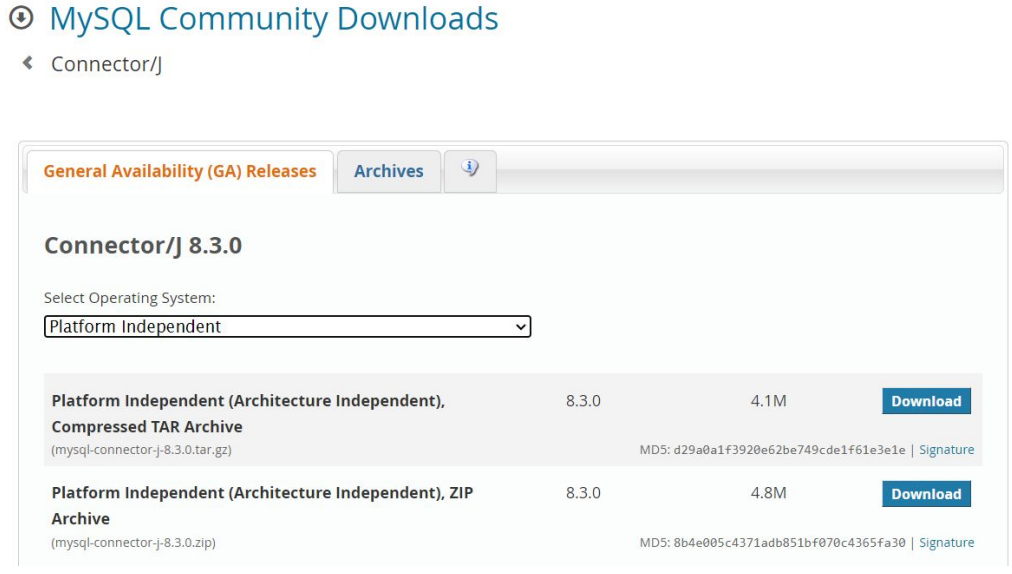
- Primero, Java es independiente de la plataforma. Puede desarrollar aplicaciones de base de datos independientes de la plataforma utilizando SQL y Java para cualquier sistema de base de datos relacional.
- En segundo lugar, el soporte para acceder a los sistemas de bases de datos desde Java está integrado en la API de Java, por lo que puede crear aplicaciones de base de datos utilizando todo el código Java con una interfaz común.
- En tercer lugar, Java se enseña en casi todas las universidades como primer lenguaje de programación o como segundo lenguaje de programación.

La arquitectura JDBC



Descarga del driver conector de MySQL

Para descargar el driver JDBC deberemos de ir a la siguiente URL: http://dev.mysql.com/downloads/connector/j/, que nos llevará a la página oficial de MySQL para adquirirlo.
Seleccionaremos la opción ”Platform Independent” para facilitarnos las cosas y posteriormente elegiremos una de las dos opciones de descarga que nos ofrecen.



Añadir el conector jdbc de MySQL a Eclipse

Podemos incluir el conector en nuestro proyecto de dos manera:

- Desde las propiedades del proyecto. Pulsamos sobre el proyecto con click derecho, Properties. y seleccionamos Java Build Path.
- Desde la opción Build Path del proyecto. Pulsamos sobre el proyecto con click derecho, Build Path → Configure Build Path.

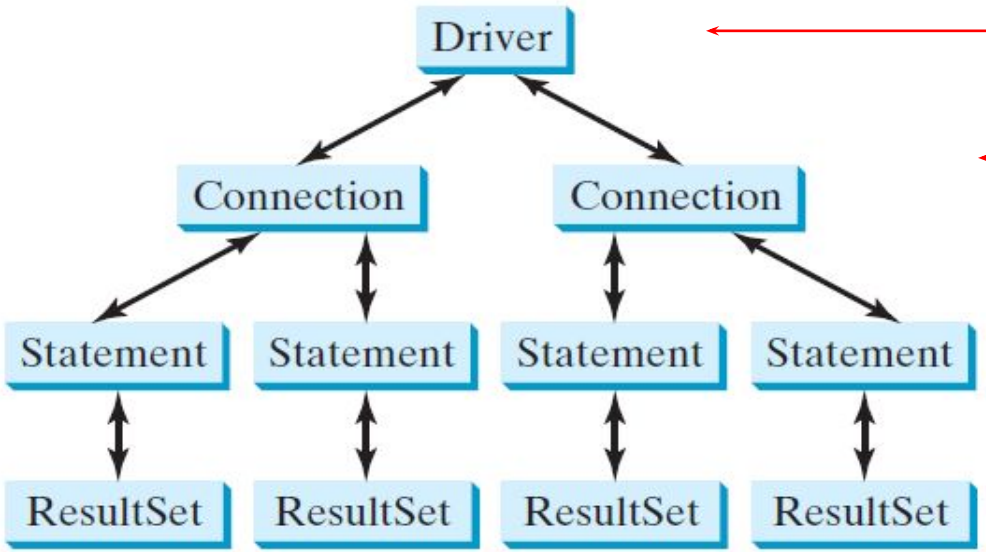
Ambos caminos nos llevarán a la ventana Java Build Path de Eclipse, donde deberemos seleccionar la opción Add External JARs para añadir el archivo ”.jar” de nuestro conector. Escogeremos el archivo mysql-connector-java-version conector-bin.jar y pulsaremos aceptar.

Con esto ya tendremos añadido a nuestro proyecto el conector JDBC de MySQL.

La API JDBC

La API de JDBC permite ejecutar sentencias sql en un SGBD desde una aplicación Java, que funcionará como un cliente que accede a los servicios del servidor de base de datos.

La API se encuentra en el paquete java.sql y está compuesto de muchas clases, las más relevantes son:



Cargando drivers.

Estableciendo conexiones.

Creación y ejecución de sentencias.

Procesando ResultSet.

Principales clases de la API JDBC

DriverManager: permite manipular los distintos drivers. Con cada driver se puede acceder a un SGBD distinto.

Conection: crea una conexión entre la la aplicación y la base de datos.
Statement: representa una sentencia sql que ejecutará el servidor de base de datos.

PreparedStatement: también representa una sentencia sql, que permite parametrizar fácilmente valores en la consulta, como por ejemplo la edad de un alumno o su fecha de nacimiento en una condición.

ResultSet: representa una tabla con el resultado que genera el SGBD tras ejecutar la sentencia de consulta de información (SELECT).

Developing JDBC Programs

- Cargando drivers.
- Estableciendo conexiones.
- Creación y ejecución de sentencias.
- Procesando Resultados

Declaración para cargar un driver:

```
Class.forName("JDBCDriverClass");
```

El driver es un Clase. Por ejemplo:

Database	Driver Class	Source
MySQL	com.mysql.cj.jdbc.Driver	Website

MySQL driver class is in mysqljdbc.jar

Para usar el controlador de MySQL debe agregar mysqljdbc.jar y classes12.jar en la ruta de clase usando el siguiente comando DOS en Windows:

```
classpath=%classpath%;c:\book\mysqljdbc.jar;c:\book\classes12.jar
```

Developing JDBC Programs

Cargando drivers.

Estableciendo conexiones.

Creación y ejecución de sentencias.

Procesando Resultados

Conexión connection = DriverManager.getConnection(databaseURL);

Database	URL Pattern
MySQL	jdbc:mysql://hostname/dbname

Ejemplo:
Para MySQL:

```
Connection connection = DriverManager.getConnection
  ("jdbc:mysql://localhost/test", "usuario", "pasword");
```

Hay usar ODBC como fuente de datos.

El método getConnection, en el caso en el que se produzca un error al crear la conexión, lanzará alguna de las siguientes excepciones:

- SQLException: si ocurre algún error en el acceso a la BD o la url es null.
- SQLTimeoutException: cuando el tiempo que ha transcurrido sin llegar a conectar a la BD es excesivo.

Creación clase Java para usar el conector

```
import java . sql . Connection ; import java . sql . DriverManager ; import java . sql . SQLException ; import java . sql . Statement ;

public class ConnectionJDBC {
        private static String db_ = " Nombre Base de Datos " ;
        private static String login_ = " Usuario de la Base de Datos " ;
        private static String password_ = " Contrasena " ;
        private static String url_ = " jdbc : mysql ://ip_servidor / " + db_ ;
        private static Connection connection_ ;
        private static Statement st_ = null ;

        public ConnectionJDBC () {
                try {
                        Class.forName ( " com . mysql . jdbc . Driver " ) ;
                        connection_ = DriverManager.getConnection ( url_ , login_ , password_ ) ;
                        if ( connection_ != null ) {
                                st_ = connection_.createStatement () ;
                                System.out.println ( " Conexion a base de datos " + db_ + " correcta . " ) ;
                        }
                        else
                                System . out . println ( " Conexion fallida . " ) ;
                } catch ( SQLException e ) { e . printStackTrace () ;}
                catch ( ClassNotFoundException e ) { e . printStackTrace () ;}
                catch ( Exception e ) { e . printStackTrace () ;}
        }
}
```

Developing JDBC Programs

Cargando drivers.

Estableciendo conexiones.

Creación y ejecución de sentencias.

Procesando Resultados

Creación de declaración:

```
Statement statement = connection.createStatement();
```

Declaración de ejecución (for update, delete, insert):

```
statement.executeUpdate
  ("create table Temp (col1 char(5), col2 char(5))");
```

Ejecución de sentencias (for select):

```
// Selecciona las columnas de la tabla de alumnos.
ResultSet resultSet = statement.executeQuery
  ("select firstName, mi, lastName from Student where lastName "
    + " = 'Smith'");
```

Developing JDBC Programs

Cargando drivers.

Estableciendo conexiones.

Creación y ejecución de sentencias.

Procesando Resultados

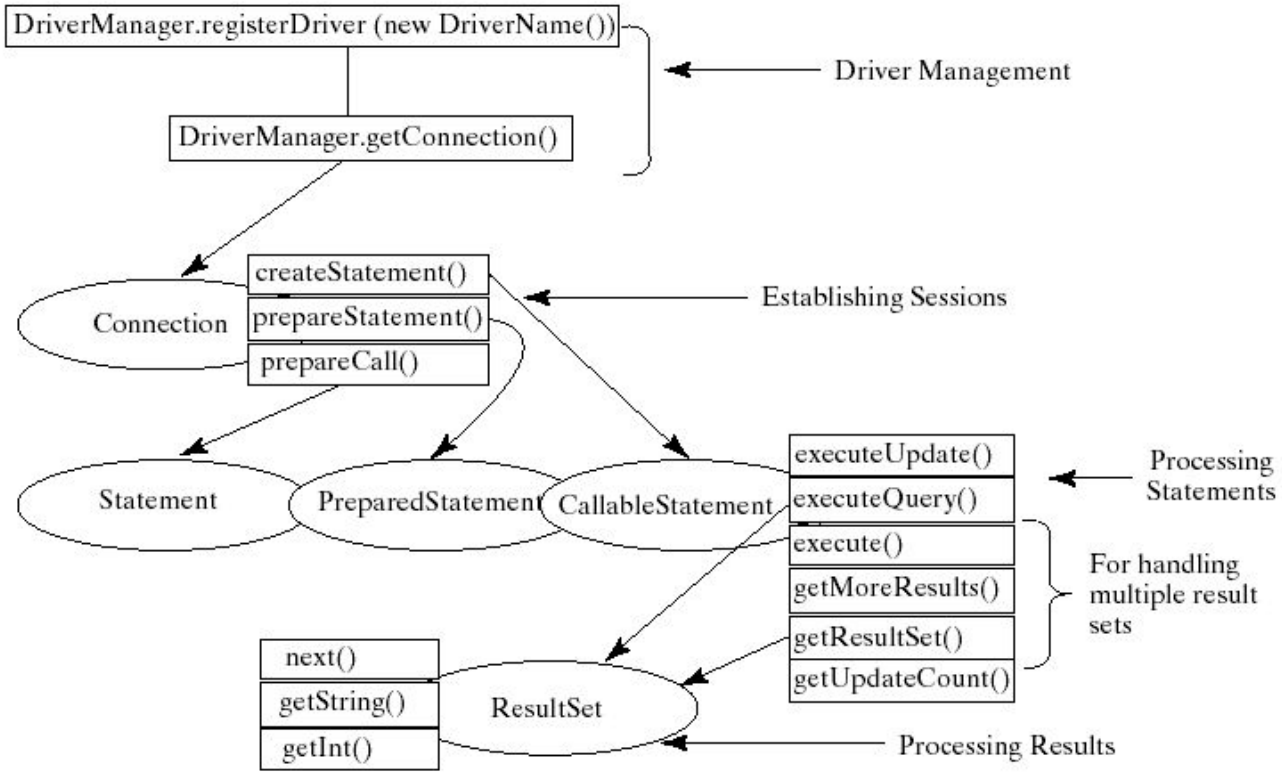
Ejecutar sentencia (for select):

```
// Selecciona las columnas de la tabla de alumnos.

ResultSet resultSet = stmt.executeQuery
  ("select firstName, mi, lastName from Student where lastName "
    + " = 'Smith'");
```

Procesar ResultSet (for select):

```
// Iterar a través del resultado y mostrar los nombres de los estudiantes

while (resultSet.next())
  System.out.println(resultSet.getString(1) + " " + resultSet.getString(2)
    + ". " + resultSet.getString(3));
```

Processing Statements Diagram



Métodos execute, executeQuery y executeUpdate

Los métodos para ejecutar sentencias SQL son execute, executeQuery y executeUpdate, cada uno de los cuales acepta una cadena que contiene una sentencia SQL como argumento. Esta cadena se pasa a la base de datos para su ejecución.

El método execute se debe utilizar si la ejecución produce múltiples conjuntos de resultados, múltiples recuentos de actualizaciones o una combinación de conjuntos de resultados y recuentos de actualizaciones.

Ejecución de consultas SELECT

El método executeQuery ejecuta una consulta y devuelve el resultado de esta mediante un objeto de tipo ResultSet. Debe utilizarse si la ejecución produce un conjunto de resultados único, como la instrucción de una Select de SQL.

Prototipo:

ResultSet executeQuery(String sql)

Métodos execute, executeQuery y executeUpdate

Ejemplo: Consultar la tabla Alumnos:

```
String sql = "SELECT * FROM Alumnos"
Statement sentencia =  con.createStatement();
ResultSet rs = sentencia.executeQuery(sql);
```

Ejecución de sentencias INSERT, UPDATE o DELETE

El método executeUpdate ejecuta la sentencia sql correspondiente y devuelve el número de filas que han sido afectadas por la sentencia, es decir, el número de registros que se han insertado, actualizado o eliminado. Aunque su nombre sea executeUpdate, sirve también para insertar o eliminar registros en una tabla.

El método executeUpdate se debe utilizar si la instrucción da como resultado un único recuento de actualizaciones o un recuento sin actualizaciones, como una instrucción SQL INSERT, DELETE, UPDATE o DDL.

Prototipo:

int executeUpdate(String sql)

Métodos execute, executeQuery y executeUpdate

Ejemplo: Incrementar la nota media en un punto a todos los alumnos del curso 1B:

```
try{
	Connection con =
		DriverManager.getConnection ("jdbc:mysql://localhost/Instituto", "root", "root");
	Statement sentencia =  con.createStatement();
	String sql = "UPDATE Alumnos SET media=media+1 " + "WHERE curso = '1B'";
	sentencia.executeUpdate(sql);
	con.close();
	System.out.println("Se ha modificado la nota media");
}catch (SQLException ex){
	System.out.println("Ha ocurrido algún error");
}
```

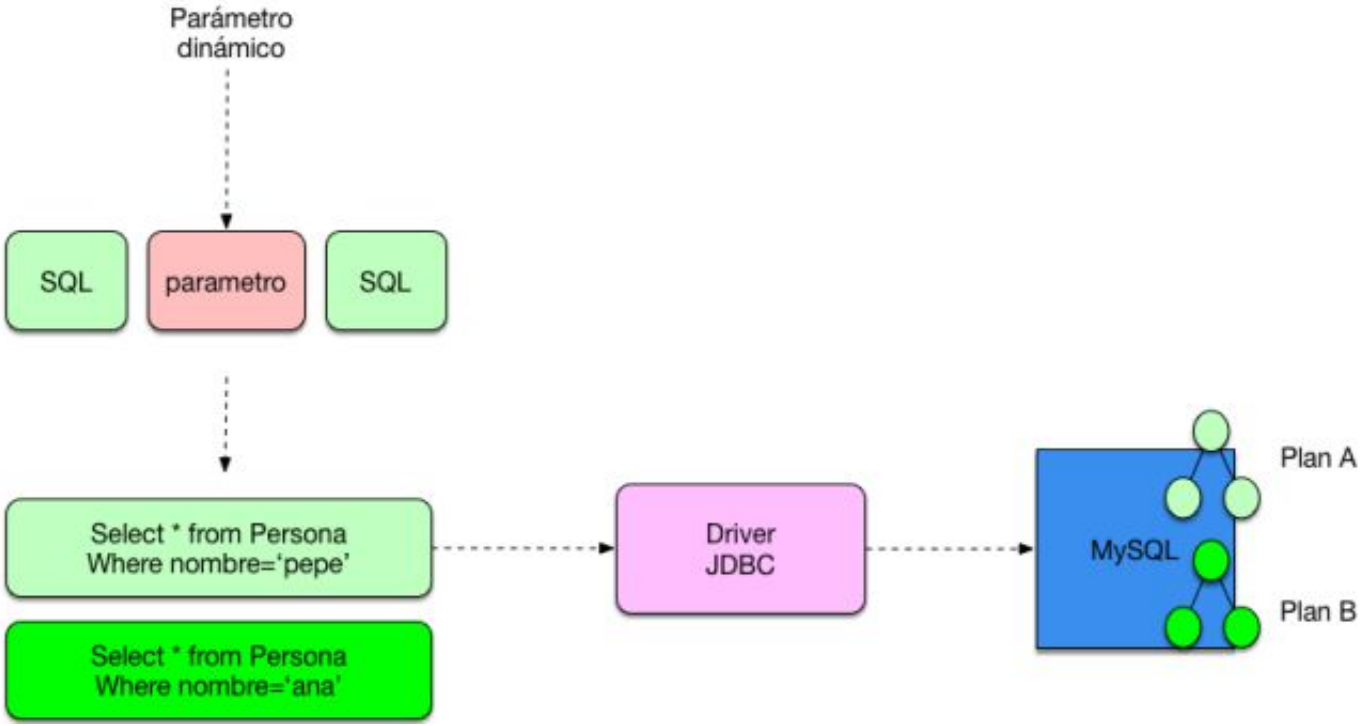
Tratamiento de declaraciones Statement

Una vez que se establece una conexión a una base de datos particular, se puede usar para enviar declaraciones SQL desde su programa a la base de datos. JDBC proporciona las interfaces Statement o PreparedStatement para facilitar el envío de sentencias a una base de datos para la ejecución y recibir los resultados de la ejecución de la base de datos.

La **Interfaz Statement** se encarga de definir una consulta SQL, de forma dinámica (con un parámetro), a ejecutar contra el motor de la base de datos. Ejemplo:

```
Connection conexion =
DriverManager.getConnection("jdbc:mysql://localhost/prueba", "root", "root");
Statement sentencia = conexion.createStatement();
String nombre="pepe";
String consulta = "select * from Persona where nombre = '"+ nombre +"'";
ResultSet rs=sentencia.executeQuery(consulta);
```

Tratamiento de declaraciones Statement



A dos consultas diferentes se crean dos planes de ejecución diferentes aunque ambas consultas sean realmente muy similares y únicamente entre en juego el valor del parámetro que las pasamos.

Recuperando datos de la base de datos

Cuando se ejecuta una consulta (SELECT) obtenemos los resultados encapsulados en un objeto de tipo *ResultSet*.

La clase **ResultSet** tiene una forma de trabajar muy parecida a un iterator. Dispone de un cursor que apunta en cada momento a una única fila (fila activa). Sólo se puede acceder a los datos de la fila activa, para trabajar con todos los datos del objeto ResultSet tendremos que ir moviendo el cursor de fila en fila.

La forma de pasar de una fila a la siguiente se realiza con el método:

boolean next(): mueve el cursor a la siguiente fila. Devuelve verdadero o falso si ha sido posible realizar el movimiento.

El booleano que devuelve next() es fundamental para conocer cuándo hemos terminado de procesar todas las filas de un ResultSet. Es habitual utilizar rs.next() como la condición de un bucle while.

Recuperando datos de la base de datos

Disponemos de los siguientes métodos para extraer los datos de la fila activa:

- *String getString(String nombreCampo)*: devuelve el valor del campo como una cadena.
- *int getInt(String nombreCampo)*: devuelve el valor del campo como un entero.
- *Double getDouble(String nombreCampo)*: devuelve el valor del campo como un real.
- *Date getDate(String nombreCampo)*: devuelve el valor del campo como una fecha.

Todos los métodos para extraer los datos de una tabla están sobrecargados para que en lugar de especificar el nombre del campo, se pueda indicar su posición en la consulta. Hay que tener en cuenta que ResultSet numera los campos de una consulta comenzando en 1.

Simple JDBC Example

```
import java.sql.*;
public class SimpleJdbc {
  public static void main(String[] args)
      throws SQLException, ClassNotFoundException {
    // Load the JDBC driver
    Class.forName("com.mysql.cj.jdbc.Driver");
    System.out.println("Driver loaded");

    // Establish a connection
    Connection connection =
            DriverManager.getConnection("jdbc:mysql://localhost/test", "root", "root");
    System.out.println("Database connected");

    // Create a statement
    Statement statement = connection.createStatement();

    // Execute a statement
    ResultSet resultSet = statement.executeQuery
      ("select firstName, mi, lastName from Student where lastName " + " = 'Smith'");

    // Iterate through the result and print the student names
    while (resultSet.next())
      System.out.println(resultSet.getString(1) + "\t" +
            resultSet.getString(2) + "\t" + resultSet.getString(3));

    // Close the connection
    connection.close();
  }
}
```

Tipos de ResultSet

El objeto ResultSet por defecto es de solo lectura, se pueden extraer los datos pero no modificarlos, y su cursor siempre avanza hacia delante:

- *String getString(String nombreCampo)*: devuelve el valor del campo como una cadena.
- *int getInt(String nombreCampo)*: devuelve el valor del campo como un entero.
- *Double getDouble(String nombreCampo)*: devuelve el valor del campo como un real.
- *Date getDate(String nombreCampo)*: devuelve el valor del campo como una fecha.

Todos los métodos para extraer los datos de una tabla están sobrecargados para que en lugar de especificar el nombre del campo, se pueda indicar su posición en la consulta. Hay que tener en cuenta que ResultSet numera los campos de una consulta comenzando en 1.

EJEMPLO: Mostrar el nombre y fecha nacimiento de todos los alumnos de un curso introducido por teclado.

```
// Load the JDBC driver
Class.forName("com.mysql.cj.jdbc.Driver");
System.out.println("Driver loaded");

// Establish a connection
Connection connection =
      DriverManager.getConnection("jdbc:mysql://localhost/EjInstituto", "root", "root");
System.out.println("Database connected");

//Solicitamos el curso al usuario
System.out.println("Escriba un curso: ");
var sc = new Scanner(System.in);
String curso= sc.nextLine();

// Create y Execute a statement
Statement statement = connection.createStatement();
String sql = "SELECT * FROM Alumnos WHERE curso = '" + curso + "'";
ResultSet rs = statement.executeQuery(sql);

// Iterate through the result and print the student
System.out.println("La lista de alumnos del curso " + curso + " es:");
while (rs.next())
       System.out.println("Alumno: " +rs.getString(2) + "\tF.nac: " + rs.getDate(3));

// Close the connection y terminal
connection.close();
sc.close();
```

Declaración parametrizada PreparedStatement

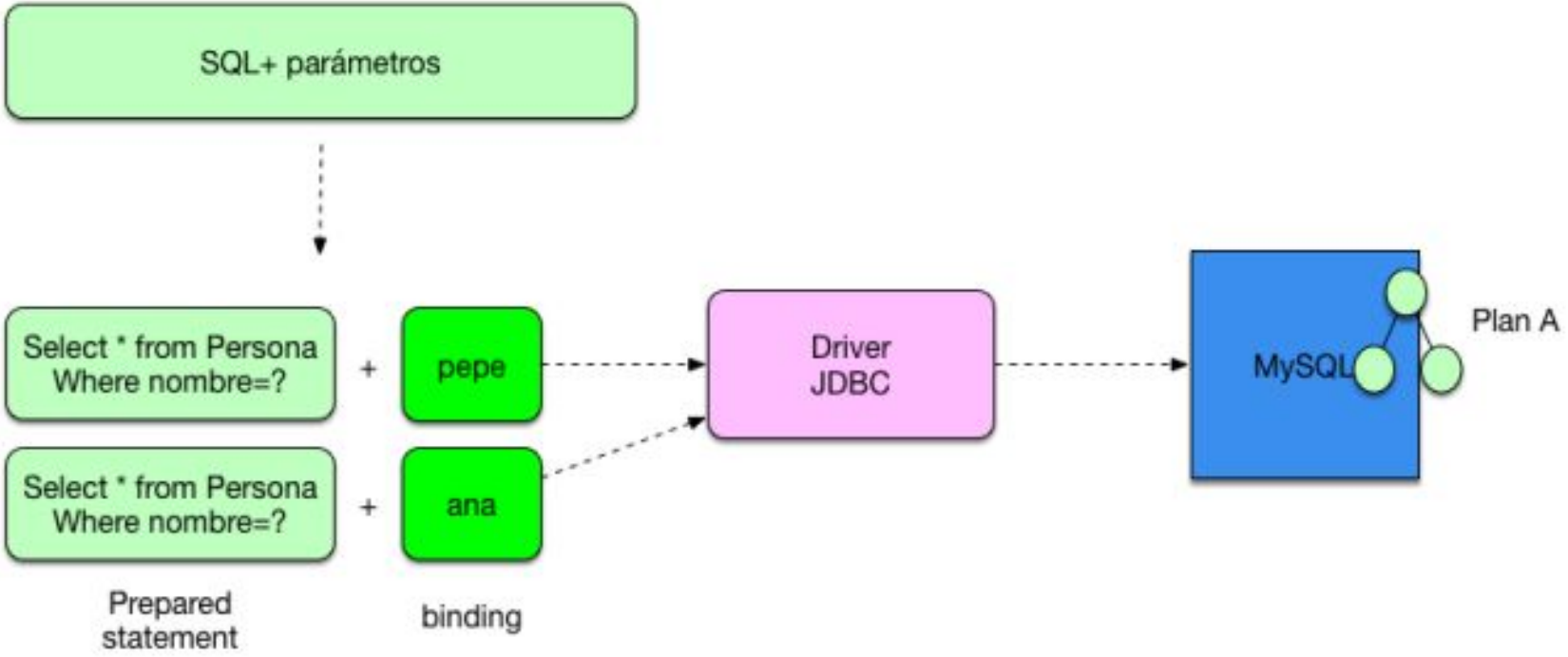
La **interfaz PreparedStatement** está diseñada para ejecutar sentencias de SQL dinámico y procedimientos almacenados de SQL con parámetros IN.
Permite adaptar y reutilizar la misma consulta varias veces.
Estas instrucciones SQL y procedimientos almacenados están precompilados para un uso eficiente cuando se ejecutan repetidamente.

Ejemplo:

```
String consulta = "select * from Persona where nombre = ? ";
Connection conexion=
      DriverManager.getConnection("jdbc:mysql://localhost/prueba", "root", "root");
PreparedStatement sentencia= conexion.prepareStatement(consulta);
sentencia.setString(1, "pepe");
ResultSet rs = sentencia.executeQuery();
```

Nota:Normalmente el uso de JDBC consultas parametrizadas mejora el rendimiento entre un 20 y un 30 % a nivel de base de datos.

Declaración parametrizada PreparedStatement



También podemos ejecutar sentencias de SQL que modifiquen la BD. Ejemplo:

```
Statement pstmt = connection.prepareStatement
    ("insert into Student (firstName, mi, lastName) + values (?, ?, ?)");
```

Declaración parametrizada PreparedStatement

Para asignar los parámetros disponemos de los siguientes métodos:

- void setString(int indiceParametro, String valor)
- void setInt(int indiceParametro, int valor)
- void setDoble(int indiceParametro, double valor)
- void setBoolean(int indiceParametro, boolean valor)
- void setDate(int indiceParametro, Date valor)
- void setNull(int indiceParametro, int tipoSQL) pone a nulo el parámetro indicado, que se trata como si fuera del tipo tipoSQL y que será uno de los tipos definidos en java.sql.types: Integer, boolean, varchar, decimal.

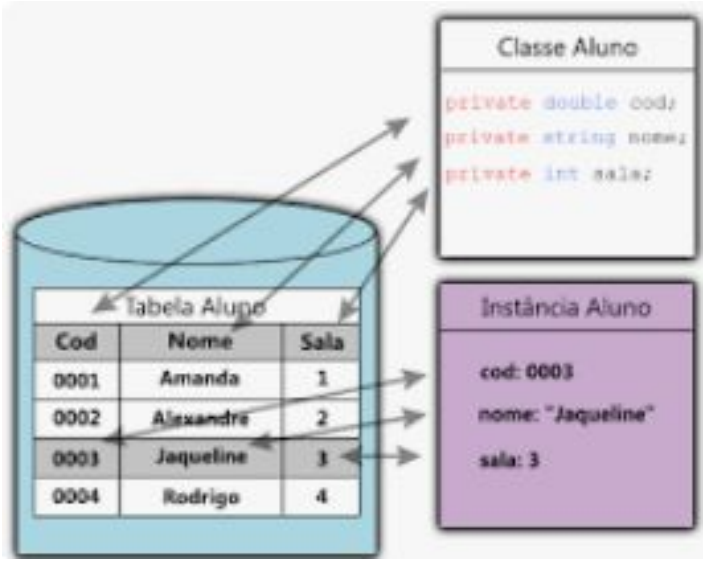
EJEMPLO: Mostrar todos los alumnos de un curso concreto cuya nota media sea mayor que una nota de corte.

```
// Load the JDBC driver y Establish a connection
. . .

// Formateamos la consulta
String sql = "SELECT nombre, media FROM Alumnos WHERE curso = ? AND media > ?";
PreparedStatement sentencia = connection.prepareStatement(sql);

//Solicitamos el curso y la nota de corte al usuario
var sc = new Scanner(System.in);
System.out.println("Escriba un curso: ");
String curso= sc.nextLine();
System.out.println("Escriba la nota de corte: ");
Double notaCorte= sc.nextDouble();

//asignamos los parámetros y ejecutamos la sentencia
sentencia.setString(1, curso); // el primer interrogante corresponde al curso
sentencia.setDouble(2, notaCorte); // el segundo interrogante es la nota de corte
ResultSet rs = sentencia.executeQuery();

// Iterate through the result and print
System.out.println("La lista de alumnos resultante es:");
while (rs.next())
  System.out.println("Nombre: " +rs.getString("nombre") + "\tMedia: " + rs.getDouble("media"));

. . .
```

Operaciones CRUD

CRUD son las siglas en inglés de crear, leer actualizar y borrar, y se usa para referirse a las operaciones básicas que se realizan sobre una base de datos. Hasta ahora hemos trabajado con datos aislados como simples variables, ahora manejaremos clases y objetos a las que incorporaremos las operaciones CRUD, que se encargarán de gestionar el objeto en la base de datos.

Existe una técnica llamada ORM, mapeo objeto-relacional, que consiste en mapear o vincular cada atributo de una clase con un campo de la base de datos. Es decir, convertimos los datos como objeto en un registro de una tabla relacional y viceversa.



Métodos de acceso a datos

A cada clase de nuestra aplicación se le añadirán los métodos:

- ***create()***: inserta los datos del objeto en la base de datos mediante una sentencia INSERT.
- ***read()***: mediante una sentencia SELECT, rescata los datos de la BD y los carga en un objeto.
- ***update()***: actualiza los datos de un objeto(que se habrán modificado) guardándolos en la base de datos, mediante la sentencia UPDATE.
  El flujo de update suele ser desde el objeto hacia la base de datos, cuando sabemos que un objeto en la BD y se ha modificado, guardamos las actualizaciones en la BD. Sin embargo, una vez modificado un objeto podemos querer desechar estos cambios y que vuelva a cargarse con los datos existentes en la BD, *refresh()*.
- ***delete()***: elimina los datos del objeto actual en la base de datos.

DAO Objetos de acceso a datos

Dado que cualquier operación CRUD en una BD no pertenece realmente a la clase de datos que definimos (por ejemplo Alumno), añadir dichos métodos a la clase es una distorsión de la abstracción que se ha ce de la realidad. Por ello, si necesitamos que en nuestra aplicación una clase se almacene en una BD, debemos diseñar una segunda clase, denominada NomClassDAO (por ejemplo AlumnoDAO) con la única misión de realizar todas las operaciones de acceso a la BD, tanto la propia conexión como las operaciones CRUD requeridas.