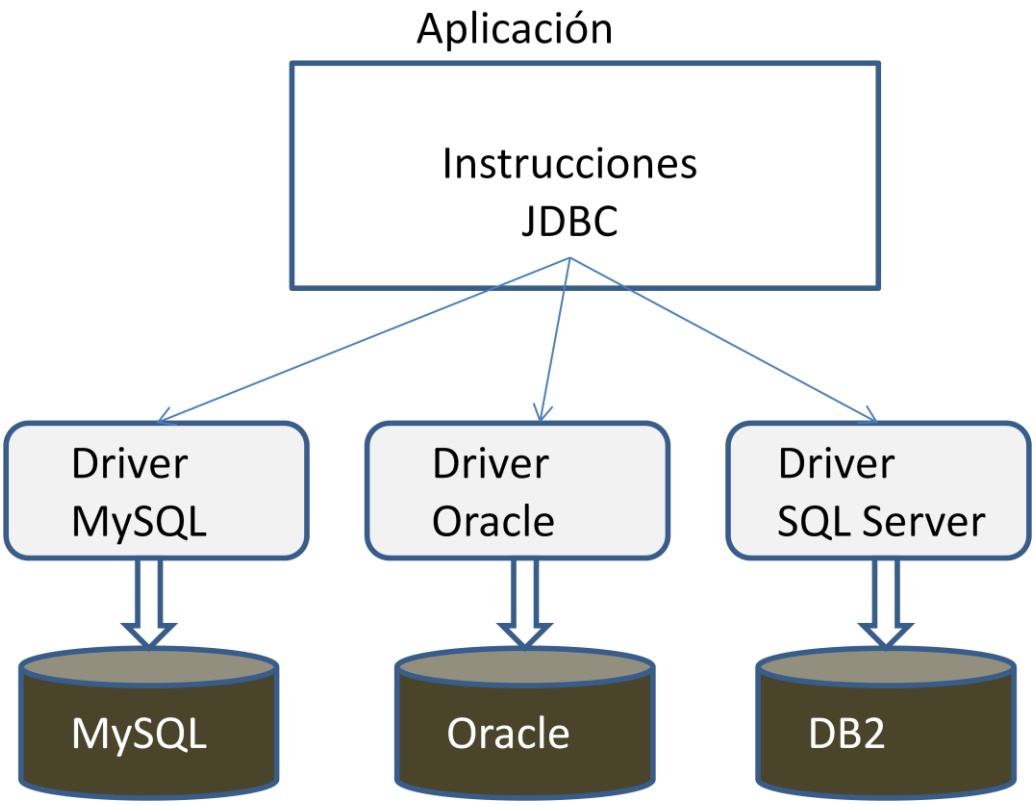
Acceso a datos desde una aplicación Java

Java proporciona un API, conocido como JDBC (Java DataBase Conectivity), que permite acceder a bases de datos relacionales desde una aplicación Java, ya sea de escritorio o Web.

Las clases e interfaces que componen JDBC forman parte del Java Standar Edition y se encuentran definidas dentro del paquete java.sql. Seguidamente vamos a explicar como utilizar este API para realizar las cuatro operaciones sobre una base de datos que se llevan habitualmente a cabo desde cualquier aplicación. Estas operaciones son conocidas como operaciones CRUD (creación, recuperación, actualización y eliminación de datos) y se llevan a cabo enviando instrucciones SQL a la base de datos desde la aplicación

El driver JDBC

JDBC se basa en el uso de un ***driver* o intermediario**, que permite a las aplicaciones abstraerse del tipo de base de datos con el que se va a trabajar. La aplicación utiliza los objetos JDBC para enviar instrucciones SQL a la base de datos y manipular los resultados, dejando al *driver* los detalles de conexión y comunicación. De este modo, el mismo **código podría ser utilizado con cualquier tipo de base de datos**, simplemente cargando el *driver* correspondiente.



Físicamente, un *driver* JDBC es un conjunto de clases Java que se distribuyen en un archivo .jar. Para cada tipo de base de datos existe un driver. Existen muchas fuentes desde las que podemos obtener un driver, una de ellas es el repositorio de Maven

Si accedemos a la siguiente dirección https://mvnrepository.com/, y en la caja de texto de búsqueda escribimos, por ejemplo, "MySQL JDBC", aparecerá una lista de diferentes librerías para trabajar con MySQL. La primera de ellas nos dará acceso a las librerías del *driver*.

Pulsando sobre el primer enlace, veremos las distintas versiones de este *driver* que han ido apareciendo. Si elegimos las de la versión 5, que son las más ampliamente probadas, nos aparecerá una página en la que podremos proceder a la descarga del *driver* .

Una vez descargado el driver, lo guardamos en cualquier directorio de nuestro disco para su posterior utilización en una aplicación

Principales clases e interfaces de JDBC

Las principales clases e interfaces del API JDBC, utilizadas en una aplicación Java para el acceso y manipulación de una base de datos, son:

- **DriverManager**. Esta clase proporciona un método estático para poder obtener conexiones contra la base de datos.
- **Connection**. Representa una conexión contra la base de datos. La obtención de una conexión es un paso previo para poder operar contra la misma.
- **Statement**. A través de este objeto podemos enviar consultas SQL a la base de datos.
- **PreparedStatement**. Es una versión alternativa de *Statement,* con la que podemos podemos precompilar consultas SQL antes de enviarlas a la BD.
- **ResultSet**. Cuando una consulta devuelve resultados (caso de las instrucciones *Select*), la manipulación de los mismos se realiza a través de un objeto *ResultSet.*

Proceso para acceder a datos desde una aplicación

Independientemente de la operación a realizar con la base de datos, si utilizamos JDBC para acceder a la base de datos habrá que seguir los siguientes pasos:

1. Establecimiento de una conexión con la base de datos
2. Envío de la instrucción SQL
3. Manipulación de resultados (si procede)
4. Cierre de la conexión

Establecimiento de una conexión

Para realizar cualquier operación con una base de datos es necesario establecer una conexión con la misma. Las conexiones con la base de datos se representan con un objeto de la interfaz Connection. Para obtener este objeto habrá que realizar dos tareas:

- **Carga del driver**. Lo primero que haremos será cargar el driver JDBC en memoria para poder realizar las operaciones a través del mismo. Esta operación se realiza una única vez durante la vida de la aplicación y para ello utilizaremos el método estático *forName()* de la clase Class, al cual le pasaremos como parámetro el nombre de la clase del driver.

  Por ejemplo, para cargar el driver de MySQL sería:

```
try {
            Class.forName("com.mysql.jdbc.Driver");
} catch (ClassNotFoundException e) {

            e.printStackTrace();
}
```

Fíjate como la llamada al método forName nos obliga a capturar la excepción ClassNotFoundException

- **Establecimiento de la conexión**. Una vez que el driver está cargado en memoria, podemos obtener un objeto Connection que represente la conexión con la base de datos utilizando el método estático *getConnection()* de DriverManager. A este método le pasaremos tres parámetros, la cadena de conexión con la base de datos, el usuario y la contraseña de acceso a la base de datos. Por ejemplo, para conectar con la base de datos "agenda" que se encuentra en la misma máquina realizaremos:

```
try {
   String url="jdbc:mysql://localhost:3306/agenda";
   Connection cn=
      DriverManager.getConnection(url,"root","root");
} catch (SQLException e) {

            e.printStackTrace();
}
```

En este ejemplo hemos considerado que el usuario y contraseña de MySQL es "root" y "root". No resulta muy elegante ni seguro poner usuarios y contraseñas en código, se puede definir estos datos en algún fichero y que nuestro programa los lea y los guarde en alguna variable. Por otro lado, vemos como cualquier llamada a métodos del API JDBC implicará capturar la excepción SQLException

Ejecución de instrucciones SQL

Una vez que disponemos del objeto Connection, podemos llamar al método createStatement() de ésta interfaz para obtener un objeto Statement. Un objeto Statement permite enviar instrucciones SQL a la base de datos a través de su método execute(). En el siguiente ejemplo, insertamos un nuevo registro en una hipotética tabla "contactos" existente en la base de datos "agenda":

```
try {
   String url="jdbc:mysql://localhost:3306/agenda";
   Connection cn=
      DriverManager.getConnection(url,"root","root");
   Statement st=cn.createStatement();
   String sql="insert into contactos(nombre,email,edad)";
   sql+=" values('prueba1','prueba1@gmail.com',22)";
   st.execute(sql);
} catch (SQLException e) {

           e.printStackTrace();
}
```

El método *execute()* lo utilizamos para envío de instrucciones que no devuelven resultados (insert, update y detele), para instrucciones de tipo select que devuelven resultados emplearemos el método *executeQuery()*.

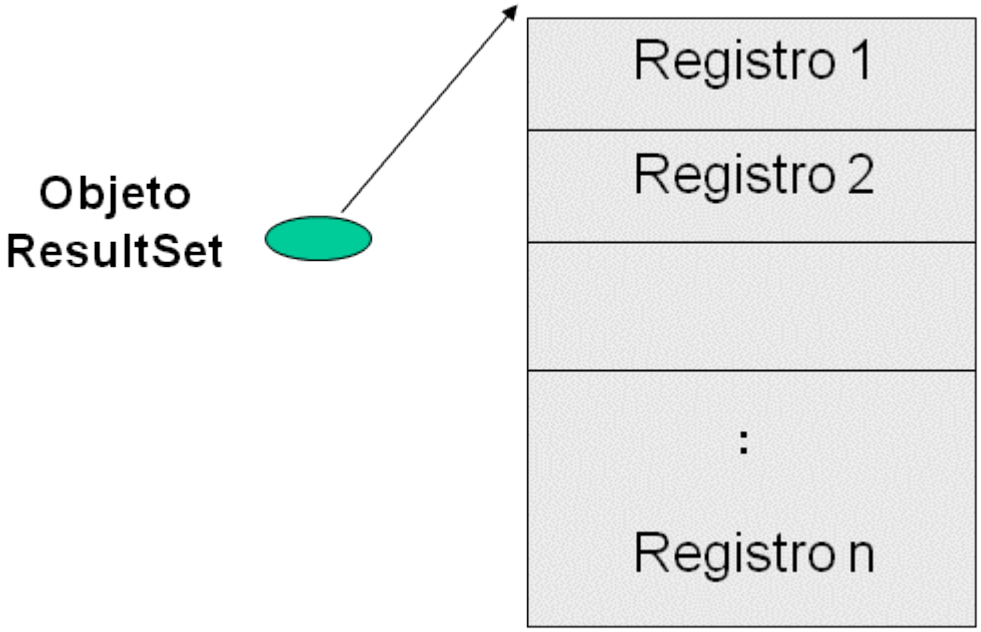
Manipulación de resultados

En el caso de que enviemos una instrucción de tipo **select** a la base de datos, podremos acceder a los resultados devueltos por la consulta a través de un objeto ResultSet. ResultSet es la interfaz que implementa el objeto devuelto por el método *executeQuery()* de Statement y proporciona una serie de métodos para acceder a los resultados de la consulta:

```
ResultSet rs=st.executeQuery("select * from contactos");
```

El objeto ResultSet es una especie de puntero de solo lectura y solo avance, de modo que para poder acceder a cada registro habrá que ir desplazándolo hacia adelante y solo podemos recuperar la información de la fila o registro apuntado, nunca modificarlo.

Inicialmente, el objeto ResultSet se encuentra posicionado delante de la primera fila a la que hace referencia el select:



Para realizar el desplazamiento por los registros, la interfaz ResultSet proporciona el método *next().* La llamada a este método desplaza el cursor al siguiente registro del conjunto, devolviendo como resultado un boolean que indica si la nueva posición apuntada se corresponde con un registro (*true*), o si el cursor se ha salido del conjunto (*false*).

Utilizando este método y una instrucción *while*, es posible recorrer todos los registros desde el primero hasta el último:

```
while(rs.next())
{
        //instrucciones
}
```

Además, la interfaz ResultSet dispone de una familia de métodos getXxx() para recuperar los datos de una determinada columna, siendo Xxx el nombre de cualquiera de los tipos básicos Java más Date, String y Object, debiéndose utilizar aquel método que se corresponda con el tipo almacenado en el campo. A estos métodos les pasaremos como parámetro el nombre de la columna cuyo valor queramos recuperar

En el siguiente ejemplo, mostramos por pantalla todos los nombres de los contactos existentes en la tabla de contactos

```
try {
   String url="jdbc:mysql://localhost:3306/agenda";
   Connection cn=
      DriverManager.getConnection(url,"root","root");
   Statement st=cn.createStatement();
   String sql="Select * from contactos";
   ResultSet rs=st.executeQuery(sql);
```

```
      while(rs.next()){
         System.out.println(rs.getString("nombre"));
      }
   } catch (SQLException e) {

            e.printStackTrace();
   }
```

Cierre de la conexión

Un objeto Connection consume muchos recursos de la máquina, así que para evitar problemas de rendimiento se todas las conexiones se deben cerrar cuando ya no van a ser utilizadas. Una conexión se cierra a través del método *close()* de Connection. En el ejemplo anterior, debemos de cerrar la conexión después de recorrer el ResultSet y mostrar los contactos:

```
   try {
      String url="jdbc:mysql://localhost:3306/agenda";
      Connection cn=
         DriverManager.getConnection(url,"root","root");
      Statement st=cn.createStatement();
      String sql="Select * from contactos";
      ResultSet rs=st.executeQuery(sql);
      while(rs.next()){
         System.out.println(rs.getString("nombre"));
      }
      cn.close();
   } catch (SQLException e) {

            e.printStackTrace();
   }
```

Sin embargo, si dejamos así el código anterior, es posible que la conexión se quede sin cerrar. Imaginemos que se produce una excepción al crear el ResultSet, en ese caso, el programa saltaría al catch y la conexión quedaría abierta. Una forma de solucionar este problema es poner la instrucción de cierre de la conexión en un bloque finally, aunque haría el código muy engorroso. Lo mejor es utilizar try con recursos, que cierra de forma implicita la conexión al salir del bloque try:

```
      String url="jdbc:mysql://localhost:3306/agenda";
   try(Connection cn=
         DriverManager.getConnection(url,"root","root");){
      Statement st=cn.createStatement();
      String sql="Select * from contactos";
      ResultSet rs=st.executeQuery(sql);
      while(rs.next()){
         System.out.println(rs.getString("nombre"));
      }
   } catch (SQLException e) {

            e.printStackTrace();
   }
```

Como vemos, en este caso no necesitamos llamar explícitamente a *close()*, basta con crear el objeto autocerrable (en nuestro caso Connection) entre los paréntesis del try. Al finalizar el bloque try, bien de forma natural o bien porque se ha producido una excepción, la conexión **se cerrará automáticamente**.