¿Qué es JDBC?

JDBC significa Java Database Connectivity, es decir, Conectividad a Base de Datos con Java.

Se trata de un conjunto de clases, interfaces y procedimientos definidos que permiten a una aplicación implementada en Java conectarse a cualquier base de datos SQL.

En el paquete java.sql existen una serie de interfaces que podremos utilizar con este objetivo.

¿Qué es un driver JDBC?

Como hemos indicado en el apartado anterior, lo que java ofrece es un conjunto de interfaces, pero no están implementadas. ¿Por qué? Porque la implementación va a depender de la base de datos a la que nos queramos conectar. No será igual la implementación para trabajar con Oracle, con MySQL, con MariaDB, con MS-SQLServer, con PostgreSQL, etc.

Un driver JDBC es precisamente el conjunto de clases para una base de datos concreta que implementa todas las interfaces necesarias para trabajar con JDBC.

De este modo, existen drivers para cada tipo de BBDD.

¿Dónde consigo un driver JDBC para trabajar con mi BBDD?

Normalmente los drivers se pueden descargar de la web oficial corporativa de la base de datos con la que queremos trabajar. El driver, al ser un conjunto de clases empaquetadas, será un fichero .jar.

Haciendo una búsqueda en Google lo encontramos rápido. Siempre escribiendo algo así: "Oracle jdbc driver" o "MySQL jdbc driver", etc.

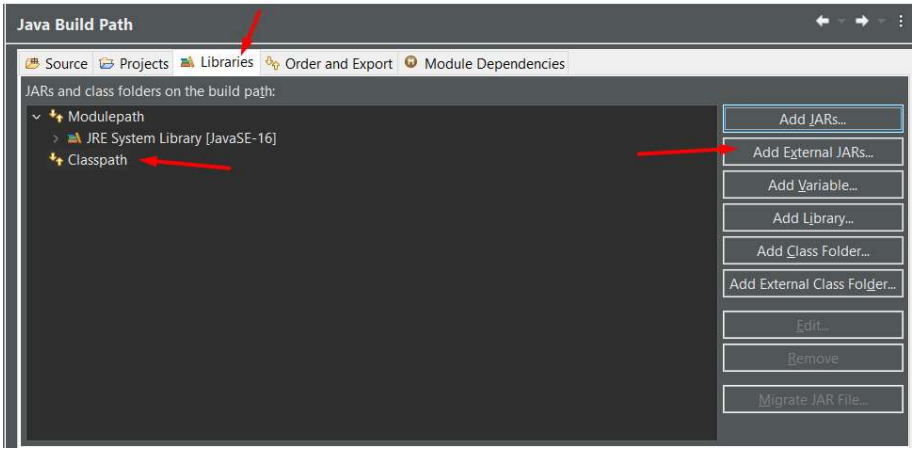
Para las BBDD principales:

- https://dev.mysql.com/downloads/connector/j/
- https://www.oracle.com/es/database/technologies/appdev/jdbc-downloads.html
- https://mariadb.com/kb/en/about-mariadb-connector-j/
- https://docs.microsoft.com/es-es/sql/connect/jdbc/download-microsoft-jdbc-driver-for-sql-server?view=sql-server-ver15

¿Cómo añado ese driver (.jar) a mi proyecto para utilizarlo?

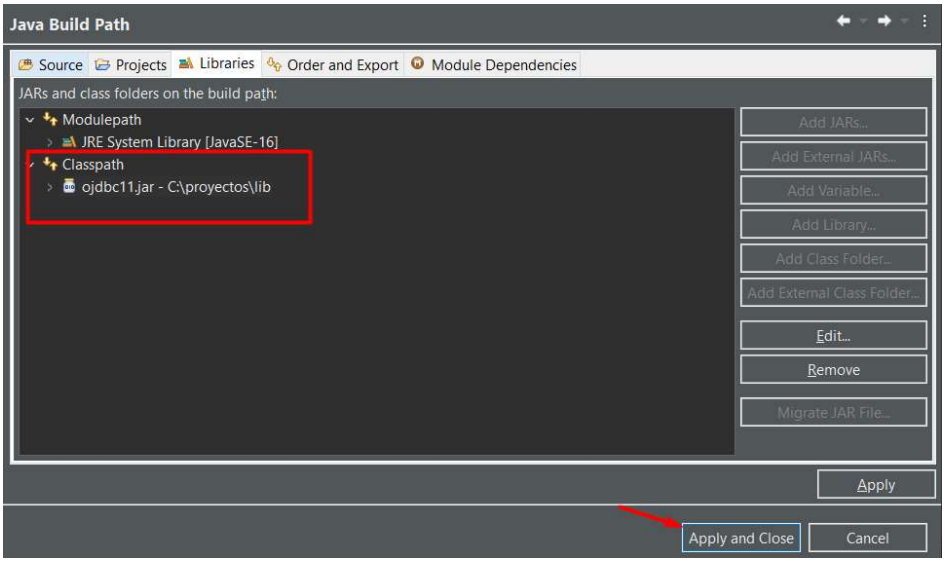
Lo añadimos como si fuera una librería más en Eclipse. Esto se hace del siguiente modo:

1. Sobre nuestro proyecto, hacemos click con el botón dcho → Build path → Configure build path…
2. A continuación, seleccionamos la pestaña "Libraries", luego seleccionamos "Classpath" y hacemos click en "Add external jar"



3. En la ventana que se abre, buscamos el .jar del driver que nos hemos descargado. Lo ideal es que lo hayamos guardado en alguna carpeta "estable" donde vaya a permanecer sin riesgo de ser movido o borrado. Es decir, evitar dejarlo en la carpeta de descargas, escritorio, etc.

4. Al seleccionar el fichero .jar, nos aparecerá después colgando de "Classpath". Podemos aceptar cerrar el build path y ya habríamos terminado



NOTA: Ten en cuenta que si cargas tu código desde otro ordenador donde el driver no esté ubicado en el mismo lugar, tendrás que volver a corregir el build path. Este tipo de problemas se evita con una gestión de dependencias de librerías externas a través de frameworks como Maven, que podréis ver en Entornos de Desarrollo o, en su defecto, en segundo curso.

¿Cuál es el procedimiento para realizar cualquier operación sobre la BBDD?

Da igual si vamos a consultar, insertar, actualizar o borrar. Desde Java, siempre tendremos que seguir estos pasos:

1. Abrir una Connection
2. Crear un Statement que nos permite ejecutar sentencias SQL
3. Ejecutar la sentencia SQL (Select, Insert, Update, Delete)
4. Leer los resultados si los hubiera (con un ResultSet)
5. Cerrar Statement (y ResultSet si lo hubiera)
6. Cerrar Connection

A continuación, vamos a ir desgranando un poco esto con ejemplos concretos…

¿Cómo abrimos una conexión?

Para abrir una nueva conexión vamos a necesitar todos estos datos. Si no los conoces, no podrás continuar:

- En qué IP o host está ubicada la BBDD
- Por qué puerto nos podemos conectar a la BBDD
- Cuál es el usuario y la contraseña
- Cuál es el nombre de la BBDD o del SID o Service Name

Además, necesitamos dos cosas más que dependerán del driver que estamos utilizando. Normalmente tendremos que consultar la documentación de dicho driver para conocerlas. Son estas dos:

- Cómo se llama la clase del driver que vamos a usar:
  - Oracle: oracle.jdbc.driver.OracleDriver
  - MySQL: com.mysql.jdbc.Driver
  - MariaDB: org.mariadb.jdbc.Driver
- Cuál es el formato de la URL JDBC de conexión:
  - Oracle:
    - Formato: jdbc:oracle:thin:@//[HOST:PUERTO]/[SID_SERVICE_NAME]
    - Ejemplo: jdbc:oracle:thin:@//localhost:1521/XE
  - MySQL:
    - Formato: jdbc:mysql://[HOST:PUERTO]/[DATABASE]
    - Ejemplo: jdbc:mysql://localhost:3306/mibasededatos
  - MariaDB:
    - Formato: jdbc:mariadb://[HOST:PUERTO]/[DATABASE]
    - Ejemplo: jdbc:mariadb://localhost:3306/mibasededatos

Con todo lo anterior… ya estamos listos!!! Así se abre una conexión:

```java
// 1. Prepararmos url de conexión y el resto de datos
String urlConexion = "jdbc:mysql://localhost:3306/ejemplo";
String claseDriver = "org.mariadb.jdbc.Driver";
String usuario = "usuario";
String password = "1234";

// 2. Cargamos la clase del driver. Aquí habrá una excepción que controlar
Class.forName(claseDriver);

// 3. Obtenemos nueva conexión. Aquí habrá otra excepción que controlar
Connection conn = DriverManager.getConnection(urlConexion, usuario, password);
```

¿Cómo consulto datos sobre una tabla?

Una vez que nos hemos conectado a la base de datos, ahora podemos lanzar sentencias SQL sobre ella. Veamos los pasos que tenemos que seguir:

1. Obtener la conexión (ya lo hemos visto en el apartado anterior)
2. Obtener un Statement a partir de la conexión.
3. Ejecutar la query SQL utilizando el Statement. Esto nos devolverá un ResultSet
4. Recorrer el ResultSet utilizando el método next(). Tratar los resultados.
5. Cerrar el ResultSet, el Statement y la conexión.
   (Al cerrar Statement, se cierra automáticamente el ResultSet también)

Lo más sencillo es utilizar un try-with-resources para abrir la conexión y el Statment, así nos aseguramos que siempre se cierren. Ejemplo:

```java
String sql = "SELECT * FROM PERSONAS";
try(Connection conn = getNewConnection();
    Statement stmt = conn.createStatement()){

    ResultSet rs = stmt.executeQuery(sql);

    while(rs.next()) {
        String nombre = rs.getString("NOMBRE");
        String dni = rs.getString("DNI");
        System.out.println(nombre + " - " + dni);
    }
}
catch (SQLException e) {
    System.err.println("Error accediendo a BBDD");
}
```

Desgranemos un poco el código anterior:

- createStatement() es un método de la clase Connection que nos devuelve un Statement. Con este objeto podremos ejecutar código SQL sobre la base de datos. ¿Cómo? Con estos métodos:
  - executeQuery() → Recibe un String con la SQL y devuelve un ResultSet
  - executeUpdate() → Recibe un String con la SQL y devuelve un Integer con el número de registros actualizados. Se usa para lanzar UPDATEs o DELETEs
  - execute() → Recibe un String con la SQL. Se usa para lanzar INSERTs, UPDATEs y DELETEs

- El objeto ResultSet es como si fuera una lista. Dispone de un método llamado next() que sirve para avanzar por la lista (como si fuera un iterador). Cuando el método next() devuelve true significa que aún no has llegado al final y quedan registros por recorrer. Cuando devuelve false significa que no hay más. Para obtener el detalle de cada registro, tenemos estos métodos:
  - getString() → Recibe el nombre de la columna que queremos consultar y devuelve un String con el valor contenido en dicha columna
  - getInt() → Igual, pero devuelve un Integer
  - getBoolean() → Igual, pero devuelve un Boolean
  - etc. → Usaremos el método apropiado según el tipo de dato que tenga nuestra columna en la tabla de la BBDD

¿Cómo inserto/actualizo/borro datos de una tabla?

Del mismo modo que se consulta. Únicamente, en lugar de utilizar el método executeQuery(), utilizamos el método execute() o executeUpdate() como hemos visto antes.

¿Y si no puedo construir la SQL concatenando cadenas?

Por ahora estamos construyendo las sentencias SQL concatenando cadenas. Es decir, al SELECT se sumamos lo que queremos en el WHERE. O a un INSERT le sumamos los valores que queremos insertar en cada columna. Esto es un poco tedioso. Además, en muchos casos, no podremos construir así la SQL. Por ejemplo, cuando lo que vayamos a insertar o escribir sea una fecha… ¿con qué formato debemos escribirlo?

Para estas situaciones, lo correcto es utilizar, en lugar de un Statement, un PreparedStatement.

Éste funciona muy parecido al Statement. Se diferencia en dos cosas:

- Para crearlo es necesario ya indicar la sentencia SQL.
- La sentencia puede llevar parámetros que se indican con una interrogación (?) y a los que luego se les puede dar valores.

Veamos un ejemplo:

```
String sql = "SELECT * FROM PERSONAS WHERE NOMBRE = ?";

try(Connection conn = getNewConnection();
    PreparedStatement stmt = conn.prepareStatement(sql)){

    // Indicamos que valor debe tomar cada parámetro ? del SQL
    stmt.setString(1, "BLAS");

    // Ejecutamos el statement igual que antes
    ResultSet rs = stmt.executeQuery();

    while(rs.next()) {
        String nombre = rs.getString("NOMBRE");
        String dni = rs.getString("DNI");
        System.out.println(nombre + " - " + dni);
    }
}
catch (SQLException e) {
    System.err.println("Error accediendo a BBDD");
}
```

Fíjate en:

- En el SQL indicamos "?" donde queramos pasar un parámetro. Podemos poner todas las que queramos. Observa que ya no ponemos comillas simples si se trata de una cadena, eso es algo que se hará luego automáticamente.
- Para darle valor a los parámetros, utilizamos los métodos set. Hay uno por cada tipo de dato: setString(), setBoolean(), setInt(), setBigDecimal(), etc.
- Los métodos set reciben:
  - El número del parámetro al que le queremos dar un valor. Empiezan desde 1
  - El valor que le queremos dar a ese parámetro
- Cuando usamos los métodos executeQuery(), execute(), executeUpdate()… ya no es necesario que indicamos la sentencia SQL.

¿Cómo trabajo con fechas (LocalDate)?

Para trabajar con fechas lo haríamos igual que con cualquier otro tipo de datos (String, Integer, etc.).

- La clase ResultSet tiene un método getDate() para cuando necesitemos obtener una fecha de la consulta que hemos realizado. Esto nos devuelve un objeto java.**sql**.Date. ¿Cómo convertimos a un LocalDate? Con el método toLocalDate()

```
DateTimeFormatter formato = DateTimeFormatter.ofPattern("dd/MM/yyyy");
while (rs.next()) {
    Date date = rs.getDate("FECHA_TEST");
    LocalDate fecha = date.toLocalDate();
    System.out.println(formato.format(fecha));
}
```

- La clase PreparedStatement tiene un método setDate() para cuando necesitamos introducir una fecha en un SQL. Tenemos que pasarle un objeto java.**sql**.Date. ¿Cómo convertimos desde un LocalDate? Con el método valueOf()

```
stmt = conn.prepareStatement("INSERT INTO TEST VALUES (?)");
LocalDate fecha = LocalDate.now();
stmt.setDate(1, Date.valueOf(fecha));
```

¿Cómo trabajo con fechas y horas (LocalDateTime)?

Para trabajar con horas lo haríamos igual que con fechas. Pero los métodos son setTimestamp() y getTimestamp():

- La clase ResultSet tiene un método getTimestamp() para cuando necesitemos obtener una fecha-hora de la consulta que hemos realizado. Esto nos devuelve un objeto java.**sql**.Timestamp. ¿Cómo convertimos a un LocalDateTme? Con el método toLocalDate()

```
DateTimeFormatter formato = DateTimeFormatter.ofPattern("dd/MM/yyyy HH:mm");
while (rs.next()) {
    Timestamp time = rs.getTimestamp("FECHA_HORA_TEST");
    LocalDateTime fechaHora = time.toLocalDateTime();
    System.out.println(formato.format(fechaHora));

}
```

- La clase PreparedStatement tiene un método setTimestamp() para cuando necesitamos introducir una fecha-hora en un SQL. Tenemos que pasarle un objeto java.**sql**.Timestamp. ¿Cómo convertimos desde un LocalDateTime? Con el método valueOf()

```
LocalDateTime fechaHora = LocalDateTime.now();
stmt = conn.prepareStatement("INSERT INTO TEST VALUES (?)");
stmt.setTimestamp(1, Timestamp.valueOf(fechaHora));
```

¿Cómo controlamos la transacción en BBDD para hacer commit o rollback?

El autocommit es una propiedad de la BBDD que significa que cada vez que hacemos una operación (insert, delete, update) automáticamente se hace un commit justo después (sin que nosotros hagamos nada). Es decir, cada vez que lanzamos la operación, si no hay errores, queda confirmada.

Por defecto, con JDBC, el autocommit está siempre activo. Por eso nunca hemos tenido que hacer commit hasta ahora.

Pero… ¿y si quiero que no se haga este commit automáticamente porque necesito hacer varias operaciones y que el commit se haga sólo al final si todo ha ido bien? → Podemos desactivar el autocommit con el método setAutoCommit():

```
conn.setAutoCommit(false);
```

Si lo hacemos, sí tendremos que hacer commit() cuando terminemos, porque no se va a hacer de manera automática. También podremos hacer rollback() si salta alguna excepción. Por ejemplo:

```
String sql = "...";
try(Connection conn = abrirConexion();
        PreparedStatement stmt = conn.prepareStatement(sql)){
    conn.setAutoCommit(false);
    try {
        // Aquí hacemos las operaciones que queramos
        // ...
        // Cuando terminemos hacemos commit
        conn.commit();
    }
    catch(SQLException e) {
        // Si hay algún error hacemos rollback
        conn.rollback();
        throw e; // relanzamos la excepción para capturarla más abajo
    }
}
catch(SQLException e) {
    e.printStackTrace();
    // Nueva excepción o lo que queramos hacer
    throw new ErrorException("...", e);
}
```