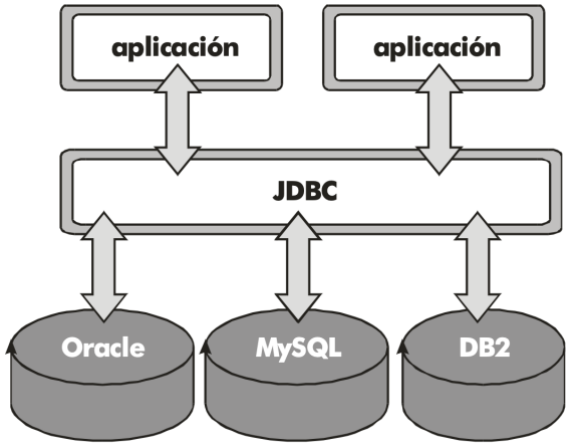
UT 9: BBDD

1. Introducción

Originalmente *Sun Microsystems* pretende desarrollar una sola API (*application programming interfaces*, interfaz de programación de aplicaciones) para el acceso a bases de datos. A esta interfaz se conoce como JDBC (*java data base connect*).

1.1. Estructura JDBC

En el diagrama siguiente se puede apreciar que las aplicaciones solo se tengan que comunicar con la interfaz JDBC. Esta es el encargada de comunicarse con los sistemas de base de datos.



2. Conexión

Para conseguir conectar una base de datos con una aplicación, nuestra aplicación requiere la URL de la base de datos y las propiedades que establezca nuestro controlador JDBC. Las clases necesarias para usar JDBC están en el paquete “java.sql”.

La cadena de conexión o URL tendría el siguiente aspecto:

jdbc:sgbd://servidor/basedatos:puerto

Por ejemplo en MySQL

jdbc:mysql://localhost/prueba:3306

La conexión se realiza mediante un objeto de la clase **java.sql.Connection**. La construcción típica implica indicar la URL de la base de datos, el usuario y la contraseña. Ejemplo (MySQL):

```
Connection con=DriverManager.getConnection(
"jdbc:mysql://localhost/almacen:3306","root","mimono");
```

El método estático **getConnection** de la clase **DriverManager** es el encargado de realizar la conexión. Al crearla pueden ocurrir excepciones **SQLException** que habrá

que capturar. Los fallos ocurren porque la URL está mal, la base de datos no está ejecutándose, el usuario no es el correcto, etc.

La conexión se cierra con el método **close** de la clase **Connection .**

3. Clases importantes

- **Statement**: Forma más simple de ejecutar un SQL. No guarda nada en caché y la forma de crear sentencias SQL se realiza mediante concatenación de *String*. Métodos:
  - boolean st.execute(sql): Se usa con CREATE o DROP.
  - int st.executeUpdate(sql): Se usa con INSERT, UPDATE o DELETE.
  - ResulSet st.executeQuery(sql): Se usa con SELECT.
- **ResultSet**: clase encargada de almacenar los resultados de una consulta SELECT creada con el método *executeQuery* de la clase *Statement*.
- **PreparedStatement**: Usa una caché, por tanto, para las consultas que se repiten tiene mejor rendimiento que con *Statement*. Dispone de método *setXXX*() para sustituir valores en las *querys*. Mismos métodos que el anterior.
- **CallableStatement**: Extiende de *PreparedStatement* y se utiliza para ejecutar *procedures* o funciones de BBDD. Distingue entre:
  - **Creación del Calleable**
  - **Ejecución**.

4. Driver

Para que una aplicación pueda funcionar, primero hay que incluir el driver. El proceso es el de siempre, se debe buscar el driver en el repositorio de Maven y luego incluirlo en el proyecto. Por ejemplo, para mysql sería el siguiente:

```
<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.11</version>
    <scope>test</scope>
  </dependency>
  <!-- https://mvnrepository.com/artifact/mysql/mysql-connector-java -->
  <dependency>
      <groupId>mysql</groupId>
      <artifactId>mysql-connector-java</artifactId>
      <version>8.0.29</version>
  </dependency>
</dependencies>
```

5. Ejemplos

Para todos los ejemplos se utilizan versiones superiores a Java 7, es decir, se utiliza *try-with-resources*, por tanto, no hay que cerrar los flujos de forma manual (siempre que se implemente bien el try).

5.1. Query Básica

```
public static void ejemploQuery() {
    List<Employee> result = new ArrayList<>();

    final String SQL_SELECT = "Select * from EMPLOYEE";

    // Desde la versión 6 no hay que registrar el driver
    // Se hace porque mysql ha cambiado de driver standar debido a que el "normal" tiene vulnerabilidades
    // Con Oracle por ejemplo no haría falta
    try {
        Class.forName("com.mysql.cj.jdbc.Driver");
    } catch (ClassNotFoundException e1) {
        e1.printStackTrace();
    }


    //La cadena de conexión de mySql tiene primero la ip del servidor, el puerto y el nombre de la base de datos.

    try (Connection conn = DriverManager.getConnection("jdbc:mysql://127.0.0.1:3306/test", "root", "password");
         PreparedStatement preparedStatement = conn.prepareStatement(SQL_SELECT)) {

            ResultSet resultSet = preparedStatement.executeQuery();

        while (resultSet.next()) {
            long id = resultSet.getLong("ID");
            String name = resultSet.getString("NAME");
            BigDecimal salary = resultSet.getBigDecimal("SALARY");
            Timestamp createdDate = resultSet.getTimestamp("CREATED_DATE");

            Employee obj = new Employee();
            obj.setId(id);
            obj.setName(name);
            obj.setSalary(salary);
            // Timestamp -> LocalDateTime
            obj.setCreatedDate(createdDate.toLocalDateTime());
            result.add(obj);

        }
        result.forEach(x -> System.out.println(x));

    } catch (SQLException e) {
        System.err.format("SQL State: %s\n%s", e.getSQLState(), e.getMessage());
    }
```

5.2. Insert con prepared statement

```
public static void ejemploInsert() {

    final String INSERT = "INSERT INTO EMPLOYEE (NAME, SALARY, CREATED_DATE) VALUES (?,?,?)";

        // Cadena de conexión con postgreSQL
    try (Connection conn = DriverManager.getConnection("jdbc:postgresql://127.0.0.1:5432/test", "postgres", "password");
         PreparedStatement preparedStatement = conn.prepareStatement(INSERT)) {

        preparedStatement.setString(1, "mkyong");
        preparedStatement.setBigDecimal(2, new BigDecimal(799.88));
        preparedStatement.setTimestamp(3, Timestamp.valueOf(LocalDateTime.now()));

        int row = preparedStatement.executeUpdate();

        // rows affected
        System.out.println(row); //1

    } catch (SQLException e) {
        System.err.format("SQL State: %s\n%s", e.getSQLState(), e.getMessage());
    }

}
```

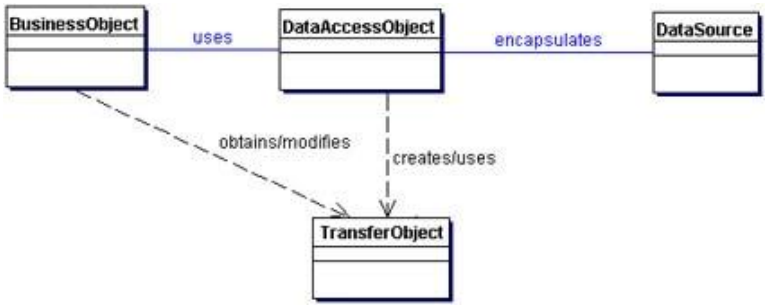
5.3. Ejemplo Procedure

```
public static void ejemploProcedure(String nombre, String pass) {
    final String CREATE_PROCEDURE = "CREATE OR REPLACE PROCEDURE get_employee_by_id( "
            + " p_id IN EMPLOYEE.ID%TYPE, "
            + " o_name OUT EMPLOYEE.NAME%TYPE, "
            + " o_salary OUT EMPLOYEE.SALARY%TYPE, "
            + " o_date OUT EMPLOYEE.CREATED_DATE%TYPE) "
            + " AS "
            + " BEGIN "
            + "     SELECT NAME, SALARY, CREATED_DATE INTO o_name, o_salary, o_date from EMPLOYEE WHERE ID = p_id; "
            + " END;";

    String runSP = "{ call get_employee_by_id(?,?,?,?) }";

    // Ejemplo de cadena de conexión con Oracle.
    try (Connection conn = DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:orcl", "system", "Password123");
         Statement statement = conn.createStatement();
         CallableStatement callableStatement = conn.prepareCall(runSP)) {

        // create or replace stored procedure
        statement.execute(CREATE_PROCEDURE);

        callableStatement.setInt(1, 3);

        callableStatement.registerOutParameter(2, java.sql.Types.VARCHAR);
        callableStatement.registerOutParameter(3, Types.DECIMAL);
        callableStatement.registerOutParameter(4, java.sql.Types.DATE);

        // run it
        callableStatement.executeUpdate();

        // java.sql.SQLException: operation not allowed: Ordinal binding and Named binding cannot be combined!
        /*String name = callableStatement.getString("NAME");
        BigDecimal salary = callableStatement.getBigDecimal("SALARY");
        Timestamp createdDate = callableStatement.getTimestamp("CREATED_DATE");*/

        String name = callableStatement.getString(2);
        BigDecimal salary = callableStatement.getBigDecimal(3);
        Timestamp createdDate = callableStatement.getTimestamp(4);

        System.out.println("name: " + name);
        System.out.println("salary: " + salary);
        System.out.println("createdDate: " + createdDate);

    } catch (SQLException e) {
        System.err.format("SQL State: %s\n%s", e.getSQLState(), e.getMessage());
        e.printStackTrace();
    }
}
```

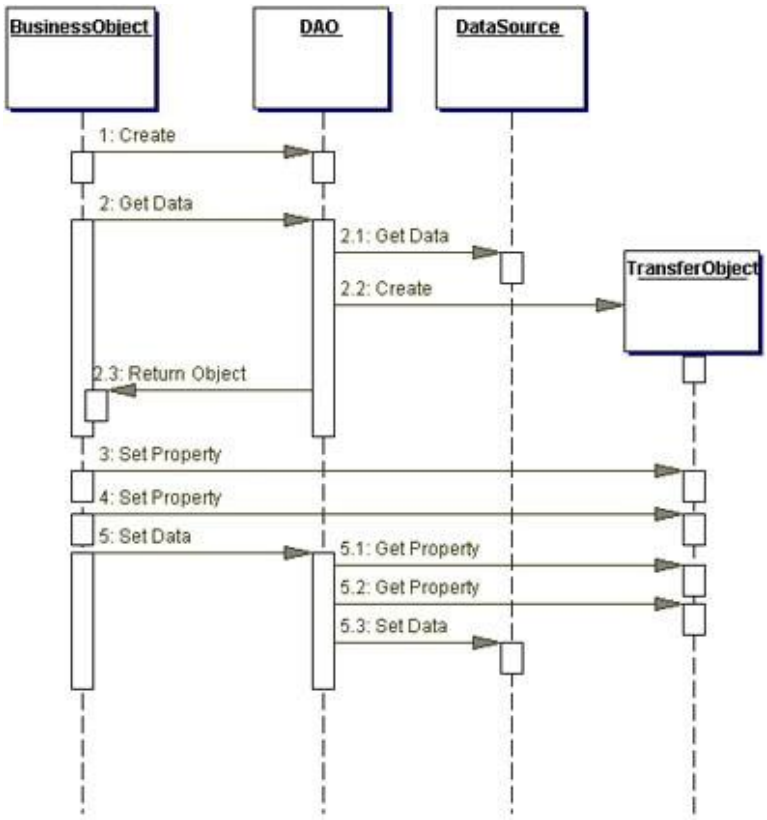
6. Factory DAO

https://www.oracle.com/java/technologies/dataaccessobject.html

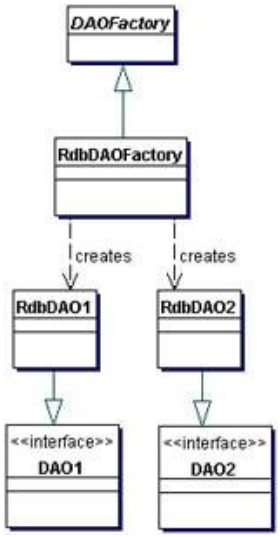
El patrón **DAO** (Objeto de Acceso a Datos) permite localizar los datos y acceder a ellos, separando la lógica de negocio de la aplicación de los datos.



Para poder visualizar cómo se ejecutaría se dispone del siguiente diagrama de flujo:

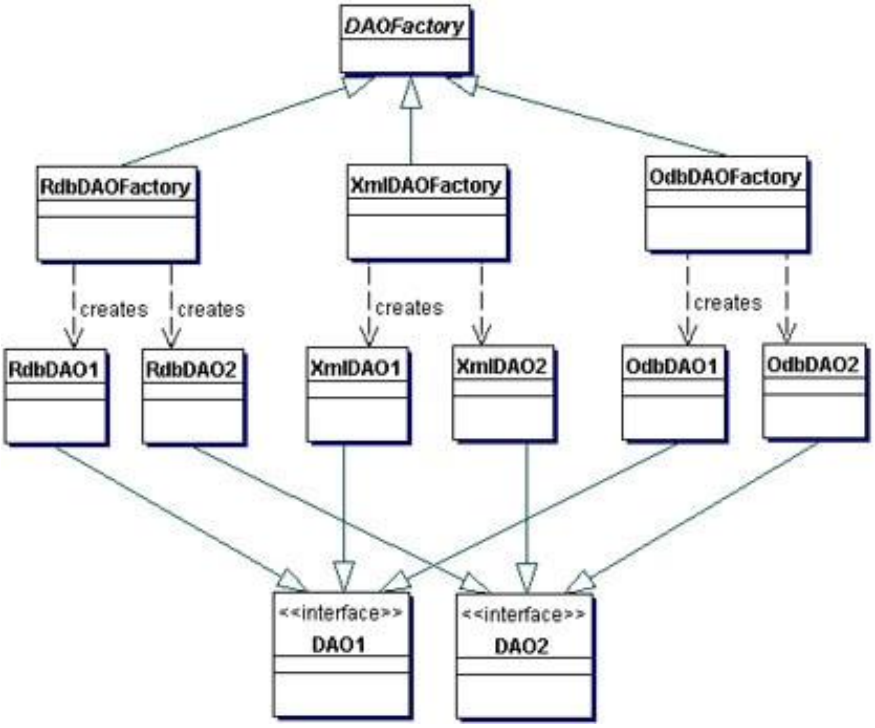


El patrón **Factory**, permite crear objetos específicos para distintas situaciones, es decir, es un generador de objetos. Por ejemplo, con BBDD, podríamos tener especificaciones diferentes por cada SGBD que se necesitase.



En este caso, el Factory nos permitiría obtener un DAO distinto para cada tabla. Por ejemplo, RdbDAO1 y 2 se podrían corresponder con clases **RdbCustomerDAO**, **RdbAccountDAO**.

¿Y qué pasa si se dispone de distintas fuentes de datos? Se podría ampliar el diagrama con el fin de disponer de factorías por cada SGBD o fuente de datos distinta.



Esto se podría aplicar a distintos SGBD. Por ejemplo, el segundo nivel de clases (las que heredan de *DAOFactory*) podrían ser factorías de *Oracle*, *MySql*, etc.