Day17: JDBC (Connection, PreparedStatement, ResultSet), with DAO Pattern

JDBC

JDBC stands for Java Database Connectivity. JDBC is a Java API to connect and execute the query with the database. JDBC API uses JDBC drivers to connect with the database.

We can use JDBC API to access tabular data stored in any relational database. By the help of JDBC API, we can save, update, delete and fetch data from the database.

The Java Database Connectivity (JDBC) API provides universal data access from the Java programming language. Using the JDBC API, you can access virtually any data source, from relational databases to spreadsheets and flat files. JDBC technology also provides a common base on which tools and alternate interfaces can be built.

There are 5 steps to connect any java application with the database using JDBC. These steps are as follows:

1) Register the driver class

forName()

Syntax of forName() method

1. **public static void** forName(String className)**throws** ClassNotFoundException

Note: Since JDBC 4.0, explicitly registering the driver is optional. We just need to put vender's Jar in the classpath, and then JDBC driver manager can detect and load the driver automatically.

Example to register the OracleDriver class

Here, Java program is loading oracle driver to esteblish database connection.

1. Class.forName("oracle.jdbc.driver.OracleDriver");

Example to register the Mysql driver class:

Class.forName("com.mysql.cj.jdbc.Driver");

---

2) Create the connection object

getConnection()

Syntax of getConnection() method

1) **public static** Connection getConnection(String url)**throws** SQLException

2) **public static** Connection getConnection(String url,String name,String password) **throws** SQLException

---

3) Create the Statement object

Syntax of createStatement() method

1. **public** Statement createStatement()**throws** SQLException

---

4) Execute the query

Syntax of executeQuery() method

1. **public** ResultSet executeQuery(String sql)**throws** SQLException

---

5) Close the connection object

Syntax of close() method

1. **public void** close()**throws** SQLException

Note: Since Java 7, JDBC has ability to use try-with-resources statement to automatically close resources of type Connection, ResultSet, and Statement.

It avoids explicit connection closing step.

Conneting Java with MySQL

Let's first create a table in the mysql database, but before creating table, we need to create database first.

```
create database databasename;

use databasename;

create table emp(id int,name varchar(40),age int);
```

```
import java.sql.*;
class MysqlCon{
    public static void main(String args[]){
        try{
            Class.forName("com.mysql.jdbc.Driver");
            Connection con=DriverManager.getConnection(
                    "jdbc:mysql://localhost:3306/databasename","username","password");
            Statement stmt=con.createStatement();
            ResultSet rs=stmt.executeQuery("select * from emp");
            while(rs.next())
                System.out.println(rs.getInt(1)+"  "+rs.getString(2)+"  "+rs.getString(3));
            con.close();
        }catch(Exception e){ System.out.println(e);}
    }
}
```

Connection

```
public interfaceConnection
extendsWrapper,AutoCloseable
```

A connection (session) with a specific database. SQL statements are executed and results are returned within the context of a connection.

A `Connection` object's database is able to provide information describing its tables, its supported SQL grammar, its stored procedures, the capabilities of this connection, and so on. This information is obtained with the `getMetaData` method.

ResultSet

A table of data representing a database result set, which is usually generated by executing a statement that queries the database.

A `ResultSet` object maintains a cursor pointing to its current row of data. Initially the cursor is positioned before the first row. The `next` method moves the cursor to the next row, and because it returns `false` when there are no more rows in the `ResultSet` object, it can be used in a `while` loop to iterate through the result set.

A default `ResultSet` object is not updatable and has a cursor that moves forward only. Thus, you can iterate through it only once and only from the first row to the last row. It is possible to produce `ResultSet` objects that are scrollable and/or updatable. The following code fragment, in which `con` is a valid `Connection` object, illustrates how to make a result set that is scrollable and insensitive to updates by others, and that is updatable. See `ResultSet` fields for other options.

```
        Statement stmt = con.createStatement();
        ResultSet rs = stmt.executeQuery("SELECT a, b FROM TABLE2");
```

The

```
ResultSet
```

interface provides

getter

methods (

```
getBoolean
```

,

```
getLong
```

, and so on) for retrieving column values from the current row. Values can be retrieved using either the index number of the column or the name of the column. In general, using the column index will be more efficient. Columns are numbered from 1. For maximum portability, result set columns within each row should be read in left-to-right order, and each column should be read only once.

For the getter methods, a JDBC driver attempts to convert the underlying data to the Java type specified in the getter method and returns a suitable Java value. The JDBC specification has a table showing the allowable mappings from SQL types to Java types that can be used by the `ResultSet` getter methods.

Column names used as input to getter methods are case insensitive. When a getter method is called with a column name and several columns have the same name, the value of the first matching column will be returned. The column name option is designed to be used when column names are used in the SQL query that generated the result set. For columns that are NOT explicitly named in the query, it is best to use column numbers. If column names are used, the programmer should take care to guarantee that they uniquely refer to the intended columns, which can be assured with the SQL *AS* clause.

Statement

The *Statement* interface accepts strings as SQL queries. Thus, **the code becomes less readable** when we concatenate SQL strings:

```
publicvoidinsert(PersonEntity personEntity) {
String query = "INSERT INTO persons(id, name) VALUES(" + personEntity.getId() + ", '"
      + personEntity.getName() + "')";

Statement statement = connection.createStatement();
    statement.executeUpdate(query);
}
```

it is vulnerable to SQL injection. The next examples illustrate this weakness.

In the first line, the update will set the column “*name*” on all the rows to “*hacker*“, as anything after “—” is interpreted as a comment in SQL and the conditions of the update statement will be ignored. In the second line, the insert will fail because the quote on the “*name*” column has not been escaped:

```
dao.update(newPersonEntity(1, "hacker' --"));
dao.insert(newPersonEntity(1, "O'Brien"))
```

Thirdly, **JDBC passes the query with inline values to the database**. Therefore, there's no query optimization, and most importantly, **the database engine must ensure all the checks**. Also, the query will not appear as the same to the database and **it will prevent cache usage**. Similarly, batch updates need to be executed separately:

```
public void insert(List<PersonEntity> personEntities) {
    for (PersonEntity personEntity: personEntities) {
        insert(personEntity);
    }
}
```

Fourthly, **the *Statement* interface is suitable for DDL queries like CREATE, ALTER, and DROP**:

```
public void createTables() {
    String query = "create table if not exists PERSONS (ID INT, NAME VARCHAR(45))";
    connection.createStatement().executeUpdate(query);
}
```

Finally, **the *Statement* interface can’t be used for storing and retrieving files and arrays**.

PreparedStatement

Firstly, the *PreparedStatement* extends the *Statement* interface. **It has methods to bind various object types**, including files and arrays. Hence, **the code becomes easy**

to understand:

```
public void insert(PersonEntity personEntity) {
    String query = "INSERT INTO persons(id, name) VALUES( ?, ?)";

    PreparedStatement preparedStatement = connection.prepareStatement(query);
    preparedStatement.setInt(1, personEntity.getId());
    preparedStatement.setString(2, personEntity.getName());
    preparedStatement.executeUpdate();
}
```

Secondly, **it protects against SQL injection**, by escaping the text for all the parameter values provided:

```
@Test
void whenInsertAPersonWithQuoteInText_thenItNeverThrowsAnException() {
    assertDoesNotThrow(() -> dao.insert(new PersonEntity(1, "O'Brien")));
}

@Test
void whenAHackerUpdateAPerson_thenItUpdatesTheTargetedPerson() throws SQLException {

    dao.insert(Arrays.asList(new PersonEntity(1, "john"), new PersonEntity(2, "skeet")));
    dao.update(new PersonEntity(1, "hacker' --"));

    List<PersonEntity> result = dao.getAll();
    assertEquals(Arrays.asList(
      new PersonEntity(1, "hacker' --"),
      new PersonEntity(2, "skeet")), result);
}
```

Thirdly, the ***PreparedStatement*** **uses pre-compilation**. As soon as the database gets a query, it will check the cache before pre-compiling the query. Consequently, **if it is not cached, the database engine will save it for the next usage.**

Moreover, **this feature speeds up the communication between the database and the JVM** through a non-SQL binary protocol. That is to say, there is less data in the packets, so the communication between the servers goes faster.

Fourthly, the ***PreparedStatement*** **provides a batch execution during a single database connection**. Let's see this in action:

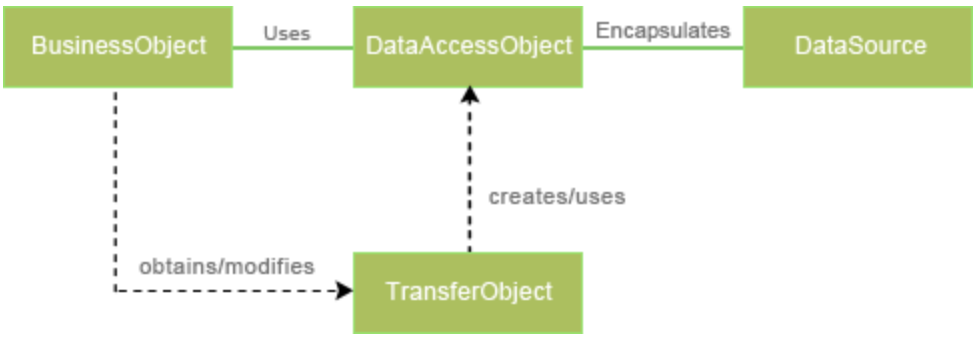
```
publicvoidinsert(List<PersonEntity> personEntities)throws SQLException {
String query = "INSERT INTO persons(id, name) VALUES( ?, ?)";
PreparedStatement preparedStatement = connection.prepareStatement(query);
for (PersonEntity personEntity: personEntities) {
        preparedStatement.setInt(1, personEntity.getId());
        preparedStatement.setString(2, personEntity.getName());
        preparedStatement.addBatch();
    }
    preparedStatement.executeBatch();
}
```

Next, the ***PreparedStatement*** **provides an easy way to store and retrieve files by using *BLOB* and *CLOB* data types**. In the same vein, it helps to store lists by converting *java.sql.Array* to a SQL Array.

Lastly, the *PreparedStatement* implements methods like *getMetadata()* that contain information about the returned result.

Data Access Object Pattern

Data Access Layer has proven good in separate business logic layer and persistent layer. The DAO design pattern completely hides the data access implementation from its clients. The interfaces given to client does not changes when the underlying data source mechanism changes. this is the capability which allows the DAO to adopt different access scheme without affecting to business logic or its clients. generally it acts as a adapter between its components and database. The DAO design pattern consists of some factory classes, DAO interfaces and some DAO classes to implement those interfaces.



The Data Access object is the primary object of this design pattern. This object abstract the data access implementations for the other object to enable transparently access to the database.

An example given below which illustrates the Data Access Design Pattern.

At first create table named student in MySql database and inset values into it as.

At first create table named student in MySql database and inset values into it as.

CREATE TABLE student (RollNo int(9) PRIMARY KEY NOT NULL, Name tinytext NOT NULL, Course varchar(25) NOT NULL, Address text);

ConnectionFactory.java

```
package roseindia.net;

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

public class ConnectionFactory {
  String driverClassName = "com.mysql.jdbc.Driver";
  String connectionUrl = "jdbc:mysql://localhost:3306/student";
  String dbUser = "root";
  String dbPwd = "root";

  private static ConnectionFactory connectionFactory = null;

  private ConnectionFactory() {
    try {
      Class.forName(driverClassName);
    } catch (ClassNotFoundException e) {
      e.printStackTrace();
    }
  }

  public Connection getConnection() throws SQLException {
    Connection conn = null;
    conn = DriverManager.getConnection(connectionUrl, dbUser, dbPwd);
    return conn;
  }

  public static ConnectionFactory getInstance() {
    if (connectionFactory == null) {
      connectionFactory = new ConnectionFactory();
    }
    return connectionFactory;
  }
}
```

StudentBean.java

```
package roseindia.net;

import java.io.Serializable;

public class StudentBean implements Serializable {
  int rollNo;
  String name;
  String course;
  String address;

  public StudentBean() {

  }

  public StudentBean(int roll, String name, String course, String address) {
    this.rollNo = roll;
    this.name = name;
    this.course = course;
    this.address = address;
  }

  public int getRollNo() {
    return rollNo;
  }

  public void setRollNo(int rollNo) {
    this.rollNo = rollNo;
  }

  public String getName() {
    return name;
  }

  public void setName(String name) {
    this.name = name;
  }

  public String getCourse() {
    return course;
  }

  public void setCourse(String course) {
    this.course = course;
  }

  public String getAddress() {
    return address;
  }
```

```
  public void setAddress(String address) {
    this.address = address;
  }

}
```

StudentJDBCDAO.java

```
package roseindia.net;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;

public class StudentJDBCDAO {
  Connection connection = null;
  PreparedStatement ptmt = null;
  ResultSet resultSet = null;

  public StudentJDBCDAO() {

  }

  private Connection getConnection() throws SQLException {
    Connection conn;
    conn = ConnectionFactory.getInstance().getConnection();
    return conn;
  }

  public void add(StudentBean studentBean) {
    try {
      String queryString = "INSERT INTO student(RollNo, Name, Course, Address) VALUES
(?,?,?,?)";
      connection = getConnection();
      ptmt = connection.prepareStatement(queryString);
      ptmt.setInt(1, studentBean.getRollNo());
      ptmt.setString(2, studentBean.getName());
      ptmt.setString(3, studentBean.getCourse());
      ptmt.setString(4, studentBean.getAddress());
      ptmt.executeUpdate();
      System.out.println("Data Added Successfully");
    } catch (SQLException e) {
      e.printStackTrace();
    } finally {
      try {
        if (ptmt != null)
          ptmt.close();
        if (connection != null)
          connection.close();
      } catch (SQLException e) {
```

```
        e.printStackTrace();
      } catch (Exception e) {
        e.printStackTrace();
      }

    }

  }

  public void update(StudentBean studentBean) {

    try {
      String queryString = "UPDATE student SET Name=? WHERE RollNo=?";
      connection = getConnection();
      ptmt = connection.prepareStatement(queryString);
      ptmt.setString(1, studentBean.getName());
      ptmt.setInt(2, studentBean.getRollNo());
      ptmt.executeUpdate();
      System.out.println("Table Updated Successfully");
    } catch (SQLException e) {
      e.printStackTrace();
    } finally {
      try {
        if (ptmt != null)
          ptmt.close();
        if (connection != null)
          connection.close();
      }

      catch (SQLException e) {
        e.printStackTrace();
      } catch (Exception e) {
        e.printStackTrace();

      }
    }
  }

  public void delete(int rollNo) {

    try {
      String queryString = "DELETE FROM student WHERE RollNo=?";
      connection = getConnection();
      ptmt = connection.prepareStatement(queryString);
      ptmt.setInt(1, rollNo);
      ptmt.executeUpdate();
      System.out.println("Data deleted Successfully");
    } catch (SQLException e) {
      e.printStackTrace();
    } finally {
      try {
        if (ptmt != null)
          ptmt.close();
        if (connection != null)
```

```
          connection.close();
      } catch (SQLException e) {
        e.printStackTrace();
      } catch (Exception e) {
        e.printStackTrace();
      }

    }

  }

  public void findAll() {
    try {
      String queryString = "SELECT * FROM student";
      connection = getConnection();
      ptmt = connection.prepareStatement(queryString);
      resultSet = ptmt.executeQuery();
      while (resultSet.next()) {
        System.out.println("Roll No " + resultSet.getInt("RollNo")
            + ", Name " + resultSet.getString("Name") + ", Course "
            + resultSet.getString("Course") + ", Address "
            + resultSet.getString("Address"));
      }
    } catch (SQLException e) {
      e.printStackTrace();
    } finally {
      try {
        if (resultSet != null)
          resultSet.close();
        if (ptmt != null)
          ptmt.close();
        if (connection != null)
          connection.close();
      } catch (SQLException e) {
        e.printStackTrace();
      } catch (Exception e) {
        e.printStackTrace();
      }

    }
  }
}
```

MainClaz.java

```
package roseindia.net;

public class MainClaz {
  public static void main(String[] args) {
    StudentJDBCDAO student = new StudentJDBCDAO();
    StudentBean alok = new StudentBean();
```

```
    alok.setName("Alok");
    alok.setRollNo(8);
    alok.setCourse("MBA");
    alok.setAddress("Ranchi");
    StudentBean tinkoo = new StudentBean();
    tinkoo.setName("Arvind");
    tinkoo.setRollNo(6);
    // Adding Data
    student.add(alok);
    // Deleting Data
    student.delete(7);
    // Updating Data
    student.update(tinkoo);
    // Displaying Data
    student.findAll();
  }
}
```

When you run this application it will display message as shown below:

```
Data Added Successfully
Data deleted Successfully
Table Updated Successfully
Roll No 1, Name Java, Course MCA, Address Motihari
Roll No 2, Name Ravi, Course BCA, Address Patna
Roll No 3, Name Mansukh, Course `M.Sc`, Address Katihar
Roll No 4, Name Raman, Course B.Tech, Address Betiah
Roll No 5, Name Kanhaiya, Course M.Tech, Address Delhi
Roll No 6, Name Arvind, Course MBA, Address Alligarh
Roll No 8, Name Alok, Course MBA, Address Ranchi
```

References:

https://docs.oracle.com/javase/7/docs/api/java/sql/ResultSet.html
https://www.javatpoint.com/java-jdbc

https://www.baeldung.com/java-statement-preparedstatement

https://www.roseindia.net/tutorial/java/jdbc/dataaccessobjectdesignpattern.html

https://www.oracle.com/java/technologies/data-access-object.html