JDBC CONNECTIVITY

Subject: JAVA PROGRAMMING
Contents: statement objects & Resultset

SAVIYA VARGHESE
Dept of BCA
2020-21

STATEMENT OBJECTS

- Once a connection is obtained we can interact with the database.
- The JDBC *Statement*,*CallableStatement*, and *PreparedStatement* interfaces define the methods and properties that enable you to send SQL or PL/SQL commands and receive data from your database.
- They also define methods that help bridge data type differences between Java and SQL data types used in a database.

- Statement

  Use this for general-purpose access to your database. Useful when you are using static SQL statements at runtime. The Statement interface cannot accept parameters.

- PreparedStatement

  Use this when you plan to use the SQL statements many times. The PreparedStatement interface accepts input parameters at runtime.

- CallableStatement

Use this when you want to access the database stored procedures.

The CallableStatement interface can also accept runtime input parameters.

Creating Statement Object

Before you can use a Statement object to execute a SQL statement, you need to create one using the Connection object's createStatement( ) method, as in the following example −

```
Statement stmt = null;
Try
 {
 stmt = conn.createStatement( );
 . . .
}
catch (SQLException e)
{ . . . }
 finally {
. . .
}
```

- Once you've created a Statement object, you can then use it to execute an SQL statement with one of its three execute methods.

- **boolean execute (String SQL)**:
- **int executeUpdate (String SQL)**:
- **ResultSet executeQuery (String SQL)**:

- **boolean execute (String SQL)**: Returns a boolean value of true if a ResultSet object can be retrieved; otherwise, it returns false. Use this method to execute SQL DDL statements or when you need to use truly dynamic SQL.

- **int executeUpdate (String SQL)**: Returns the number of rows affected by the execution of the SQL statement.
- Use this method to execute SQL statements for which you expect to get a number of rows affected - for example, an INSERT, UPDATE, or DELETE statement.

- **ResultSet executeQuery (String SQL)**: Returns a ResultSet object.
- Use this method when you expect to get a result set, as you would with a SELECT statement.

CLOSING STATEMENT OBJECT

A simple call to the close() method will do the job. If you close the Connection object first, it will close the Statement object as well. However, you should always explicitly close the Statement object to ensure proper cleanup.

```
Statement stmt = null;
 try
{
 stmt = conn.createStatement( );
 . . . }
catch (SQLException e)
 { . . . }
 finally
{
 stmt.close();
 }
```

THE PREPAREDSTATEMENT OBJECTS

- The *PreparedStatement* interface extends the Statement interface, which gives you added functionality with a couple of advantages over a generic Statement object.
- This statement gives you the flexibility of supplying arguments dynamically.
- Creating PreparedStatement Object

```
PreparedStatement pstmt = null;
try
{
String SQL = "Update Employees SET age = ? WHERE id =
   ?"; pstmt = conn.prepareStatement(SQL); . . .
}
catch (SQLException e)
{ . . . }
 finally { . . . }
```

- All parameters in JDBC are represented by the **?** symbol, which is known as the parameter marker. You must supply values for every parameter before executing the SQL statement.

- The **setXXX()** methods bind values to the parameters, where **XXX** represents the Java data type of the value you wish to bind to the input parameter. If you forget to supply the values, you will receive an SQLException.

- Each parameter marker is referred by its ordinal position. The first marker represents position 1, the next position 2, and so forth. This method differs from that of Java array indices, which starts at 0.

- All of the **Statement object's** methods for interacting with the database (a) execute(), (b) executeQuery(), and (c) executeUpdate() also work with the PreparedStatement object. However, the methods are modified to use SQL statements that can input the parameters.

CLOSING PREPAREDSTATEMENT OBJECT

- Just as you close a Statement object, for the same reason you should also close the PreparedStatement object.
- A simple call to the close() method will do the job.
- If you close the Connection object first, it will close the PreparedStatement object as well.
- However, you should always explicitly close the PreparedStatement object to ensure proper cleanup.

```
PreparedStatement pstmt = null;
Try
 {
 String SQL = "Update Employees SET age = ?
   WHERE id = ?";
 pstmt = conn.prepareStatement(SQL);
 . . . }
catch (SQLException e)
 { . . . }
 finally
 {
pstmt.close();
}
```

THE CALLABLESTATEMENT OBJECTS

- Just as a Connection object creates the Statement and PreparedStatement objects, it also creates the CallableStatement object, which would be used to execute a call to a database stored procedure.

CREATING CALLABLESTATEMENT OBJECT

- Suppose, you need to execute the following Oracle stored procedure –
- CREATE OR REPLACE PROCEDURE getEmpName (EMP_ID IN NUMBER, EMP_FIRST OUT VARCHAR) AS

```
BEGIN
 SELECT first INTO EMP_FIRST
 FROM Employees
  WHERE ID = EMP_ID;
END;
```

- Three types of parameters exist: IN, OUT, and INOUT.
- The PreparedStatement object only uses the IN parameter.
- The CallableStatement object can use all the three.
- IN:- A parameter whose value is unknown when the SQL statement is created.
- You bind values to IN parameters with the setXXX() methods.

- OUT:-

  A parameter whose value is supplied by the SQL statement it returns.

  You retrieve values from theOUT parameters with the getXXX() methods.

- INOUT:-

  A parameter that provides both input and output values.

  You bind variables with the setXXX() methods and retrieve values with the getXXX() methods.

- **Connection.prepareCall()** method to instantiate a **CallableStatement** object based on the preceding stored procedure –

```
CallableStatement cstmt = null;
 try
 {
 String SQL = "{call getEmpName (?, ?)}";
 cstmt = conn.prepareCall (SQL);
 . . .
}
catch (SQLException e)
 { . . . }
Finally
 { . . . }
```

- The String variable SQL, represents the stored procedure, with parameter placeholders.
- Using the CallableStatement objects is much like using the PreparedStatement objects. You must bind values to all the parameters before executing the statement, or you will receive an SQLException.
- If you have IN parameters, just follow the same rules and techniques that apply to a PreparedStatement object; use the setXXX() method that corresponds to the Java data type you are binding.

- When you use OUT and INOUT parameters you must employ an additional CallableStatement method, registerOutParameter().
- The registerOutParameter() method binds the JDBC data type, to the data type that the stored procedure is expected to return.
- Once you call your stored procedure, you retrieve the value from the OUT parameter with the appropriate getXXX() method. This method casts the retrieved value of SQL type to a Java data type.

CLOSING CALLABLESTATEMENT OBJECT

- Just as you close other Statement object, for the same reason you should also close the CallableStatement object.

- A simple call to the close() method will do the job. If you close the Connection object first, it will close the CallableStatement object as well. However, you should always explicitly close the CallableStatement object to ensure proper cleanup.

```
CallableStatement cstmt = null;
 try
 {
String SQL = "{call getEmpName (?, ?)}";
 cstmt = conn.prepareCall (SQL);
 . . . }
catch (SQLException e)
 { . . . }
Finally
 {
cstmt.close();
 }
```

JDBC - RESULT SETS

- The SQL statements that read data from a database query, return the data in a result set.
- The SELECT statement is the standard way to select rows from a database and view them in a result set. The *java.sql.ResultSet* interface represents the result set of a database query.
- A ResultSet object maintains a cursor that points to the current row in the result set. The term "result set" refers to the row and column data contained in a ResultSet object.

The methods of the ResultSet interface can be broken down into three categories –

- **Navigational methods:** Used to move the cursor around.
- **Get methods:** Used to view the data in the columns of the current row being pointed by the cursor.
- **Update methods:** Used to update the data in the columns of the current row. The updates can then be updated in the underlying database as well.

- JDBC provides the following connection methods to create statements with desired ResultSet –
- **createStatement(int RSType, int RSConcurrency);**
- **prepareStatement(String SQL, int RSType, int RSConcurrency);**
- **prepareCall(String sql, int RSType, int RSConcurrency);**

- The first argument indicates the type of a ResultSet object and the second argument is one of two ResultSet constants for specifying whether a result set is read-only or updatable.

TYPE OF RESULTSET

The possible RS Type are given below.

- ResultSet.TYPE_FORWARD_ONLY

The cursor can only move forward in the result set.

- ResultSet.TYPE_SCROLL_INSENSITIVE

The cursor can scroll forward and backward, and the result set is not sensitive to changes made by others to the database that occur after the result set was created.

- ResultSet.TYPE_SCROLL_SENSITIVE.

The cursor can scroll forward and backward, and the result set is sensitive to changes made by others to the database that occur after the result set was created.

- **If you do not specify any ResultSet type, you will automatically get one that is TYPE_FORWARD_ONLY**

CONCURRENCY OF RESULTSET

- The possible RSConcurrency are given below. If you do not specify any Concurrency type, you will automatically get one that is CONCUR_READ_ONLY.

- ResultSet.CONCUR_READ_ONLY

Creates a read-only result set.

- ResultSet.CONCUR_UPDATABLE

Creates an updateable result set.

```
try
{
 Statement stmt = conn.createStatement(
   ResultSet.TYPE_FORWARD_ONLY,
   ResultSet.CONCUR_READ_ONLY);
}
 catch(Exception ex)
{ .... }
Finally
 { .... }
```

NAVIGATING A RESULT SET

There are several methods in the ResultSet interface that involve moving the cursor, including –

- **public void beforeFirst() throws SQLException**Moves the cursor just before the first row.
- **public void afterLast() throws SQLException**Moves the cursor just after the last row.

public boolean first() throws SQLException

Moves the cursor to the first row.

public void last() throws SQLException

Moves the cursor to the last row.

public boolean absolute(int row) throws SQLException

Moves the cursor to the specified row.

public boolean relative(int row) throws SQLException

Moves the cursor the given number of rows forward or backward, from where it is currently pointing.

- **public boolean previous() throws SQLException**

  Moves the cursor to the previous row.

  This method returns false if the previous row is off the result set.

- **public boolean next() throws SQLException**

  Moves the cursor to the next row.

  This method returns false if there are no more rows in the result set.

- **public int getRow() throws SQLException**

  Returns the row number that the cursor is pointing to.

- **public void moveToInsertRow() throws SQLException**

  Moves the cursor to a special row in the result set that can be used to insert a new row into the database.

  The current cursor location is remembered.

- **public void moveToCurrentRow() throws SQLException**

  Moves the cursor back to the current row if the cursor is currently at the insert row; otherwise, this method does nothing