# Lesson: 1.3

# JDBC Statements

## Objectives:

In this lesson, you will exercise to:

- ✓ Understand about JDBC Statement.
- ✓ Creating and using Statement Object.
- ✓ Use PreparedStatement Object.
- ✓ Use CallableStatement Object.

## Understand JDBC Statement

- A Statement object is used to send SQL statements to a database.
- There are actually three kinds of Statement objects, all of which act as containers for executing SQL statements on a given connection: *Statement, PreparedStatement*, which inherits from *Statement*, and *CallableStatement*, which inherits from *PreparedStatement*.
- 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.
- The Statement interface provides basic methods for executing statements and retrieving results.
- They are specialized for sending particular types of SQL statements; a *Statement* object is used to execute a simple SQL statement with no parameters.

## Creating Statement Objects

- Once a connection to a particular database is established, that connection can be used to send SQL statements.
- Statement objects are created by *Connection* object, using the *createStatement* ().

**Sample code Fragment**

```
Connection conn = dataSource.getConnection (user, passwd);
Statement stmt = conn.createStatement ();
```

- Each Connection object can create multiple Statement objects that may be used concurrently by the program.
- Additional constructors may be used to set the type and concurrency or the type, concurrency, and holdability of any result sets produced by a *Statement*.
- Can use the Statement object to returns result sets that are scrollable, that are insensitive to changes made while the ResultSet object is open, that can be updated, and that do not close the ResultSet objects when commit is called.
- The method used to execute a Statement object depends on the type of SQL statement being executed.
- The Statement interface provides three different methods for executing SQL statements: *executeQuery*, *executeUpdate*, and *execute*. The correct method to use is determined by what the SQL statement produces.
- If the Statement object represents an SQL query returning a ResultSet object, the method *executeQuery* should be used.

- If the SQL is known to be a DDL statement or a DML statement returning an update count, the method *executeUpdate* should be used.
- If the type of the SQL statement is not known, the method *execute* should be used.

**Core Note**

Be careful to always put spaces in the SQL string at the right places!

## ✍ Task 9 : Explore the API

9.1 Identify the method which returns a new default *Statement* object.

9.2 Identify the method which returns a new Statement object that will generate ResultSet objects with the given type, concurrency, and holdability.

9.3 Identify the method which returns either the row count for INSERT, UPDATE, or DELETE statements, or 0 for SQL statements that return nothing.

9.4 Identify the methods which returns the current result as a *ResultSet* object or null if the result is an update count or there are no more results

9.5 Method which retrieves the result set type for *ResultSet* objects generated by this Statement object.

## ✍ Fill in Queries:

1. Create a scrollable, insensitive, updatable *ResultSet* that stays open after the method *commit* is called.

2. Complete the method declaration: **public <*return type*> executeUpdate( *Parameter1, Parameter2* ) throws <*Exception*>**

3. Write a *Statement* object that returns a ResultSet object.

4. Complete the method declaration: **public <*return type*> getMoreResults (*Parameter1* ) throws <*Exception*>**

5. Method which returns the first *SQLWarning* object or null if there are no warnings.

**Task 9 Workspace**

## ✍ Code 1 : Test the Coding Skills

**Code the Program**

// CreatePassengerTable.java

**package jdbcdemo;**

/* Following is the example which makes use of *Statement* object to execute DDL command that creates a *Passenger* table */.

//**Import the appropriate classes and interfaces.**

public class **CreatePassengerTable**

{ public static void main (_____)

{ try

{

Class.forName ( "_____" ); //**Type 4 Driver**

String url = "_____";

Connection con = DriverManager._____( url, "db2admin", "db2admin" );

Statement statement = _____();

**Code the Program…….Continued**

```
String createPassengerTable = //DDL command to create Passenger Table.

statement._____( _____ );
 //Clean-up environment.
_____;
_____;
 } catch( _____ se ) { e.printStackTrace(); }
 }
 } // End of Class.
```

## Using PreparedStatement Object

- The *PreparedStatement* interface extends *Statement*, adding the ability to set values for parameter markers contained within the statement.
- PreparedStatement objects represent SQL statements that can be prepared, or precompiled, for execution once and then executed multiple times.
- Parameter markers, represented by **"?"** in the SQL string, are used to specify input values to the statement that may vary at runtime.
- Because PreparedStatement objects are precompiled, their execution can be faster than that of *Statement* objects.
- As with Statement objects, you create *PreparedStatement* objects with a Connection method.

### Sample code Fragment

```
PreparedStatement updateSales = con.prepareStatement ("UPDATE COFFEES SET SALES = ? WHERE COF_NAME = ?");
updateSales.setInt(1, 75);
updateSales.setString(2, "Colombian");
updateSales.executeUpdate();
```

- User need to supply values to be used in place of the question mark placeholders, if there are any, before user can execute a PreparedStatement object.
- User does this by calling one of the *setXXX* methods defined in the class PreparedStatement.
- Once a parameter has been set with a value, it will retain that value until it is reset to another value or the method *clearParameters* is called.
- The setter methods for setting IN parameter values must specify types that are compatible with the defined SQL type of the input parameter.

## ✍ Task 10 : Explore the API

10.1 How to set value for IN clause in a PreparedStatement in JDBC while executing a query.
10.2 Identify the method which returns description of a ResultSet object's columns or null if the driver cannot return a ResultSetMetaData object.
10.3 Identify the method that adds a set of parameters to this *PreparedStatement* object's batch of commands.
10.4 Identify the method which sets the value of the designated parameter using the given object.
10.5 Identify the method which returns *ParameterMetaData* object that contains information about the number, types and properties of this PreparedStatement object's parameters.

## ✍ Fill in Queries:

1. Complete the method declaration: **<*Specifier*> void setObject(*Parameter1, Parameter2*) throws <*Exception*>**

2. Write the statement which creates a *PreparedStatement* object such that each time it is executed, it will produce a *ResultSet* object that is scrollable and updatable.

3. Write the method which returns either (1) the row count for INSERT, UPDATE, or DELETE statements or (2) 0 for SQL statements that return nothing.

4. Method which submits a batch of commands to the database for execution and if all commands execute successfully, returns an array of update counts.

5. Complete the method declaration: **public <*return type*> getResultSet() throws <*Exception*>**

**Core Approach**
At the point of creation of a *PreparedStatement* object the SQL code is sent to the DB and compiled. Subsequent executions may therefore be more efficient than normal statements

**Task 10 Workspace**

## ✍ Code 2 : Test the Coding Skills

**Code the Program**

```
// GetPassengerDetails.java
package jdbcdemo;

/* Following is the example which makes use of PreparedStatement object to execute
   DML command that retrieves the Passenger details */.

//Import the appropriate classes and interfaces.



public class GetPassengerDetails
{  static private final String driver ="_____"; //Type 1 Driver

   static private final String connection = "_____";

 public static void main(String args[]) {

 Connection con = _____;

 PreparedStatement pst = _____;

 ResultSet rs = _____;

 try {

        //Load the respective driver. _____;

        con = _____ //Get the Connection Object.
```

**Code the Program…….Continued**

```
String sql = "_____ " + "in(_____)";
pst = con._____(_____);
pst._____(____, _____);
pst._____(____, _____);
pst._____(____, _____);
//Returning the ResultSet Object.
rs = pst._____();
System.out.println("PassengerID\tFirstName");
while (rs._____()) { //Code for displaying the Passenger details.



}
 //Clean-up environment.
_____;
_____;
} catch( _____ se ) { e.printStackTrace(); }
}
} // End of Class.
```

## ✍ Code 3 : Test the Coding Skills

- The code fragment that follows demonstrates using a for loop to set values for parameters in the *PreparedStatement* object *updateBookings*.
- The array salesForWeek holds the weekly sales amounts. These sales amounts correspond to the Airlines names listed in the array airlines, so that the first amount in salesForWeek (175) applies to the first Airline name in airlines (" Kingfisher "), the second amount in salesForWeek (150) applies to the second Airline name in airlines (" AirIndia "), and so on.
- The code fragment demonstrates updating the *Num_of_seats* column for all the Airlines in the table *Reservation.*

**Code the Program**

```
PreparedStatement _____________________________;

String updateString = "update Reservation" + "set Num_of_seats = ? where Flight_id
                                                                    like ? ";

updateSales = con._____________________________(_____________________);

int [] _____________________ = {175, 150, 60, 15, 90};

String [] airlines = {"Kingfisher", "AirIndia", "Spicejet", "Thai", "AirLufthansa"};

int len = airlines._____________________;

for(int i = 0; i < ______; i++)
{
updateSales._____________(1, salesForWeek[i]);

updateSales._____________________(2, airlines [i]);

updateSales._____________________();
}
```

## Using CallableStatement Object

- A *CallableStatement* object provides a way to call stored procedures in a standard way for all RDBMSs.
- A stored procedure is stored in a database; the *call* to the stored procedure is what a CallableStatement object contains.
- This call is written in an escape syntax that may take one of two forms: one form with a result parameter, and the other without one.
- A result parameter, a kind of OUT parameter, is the return value for the stored procedure.
- Both forms may have a variable number of parameters used for input (IN parameters), output (OUT parameters), or both (INOUT parameters).
- A question mark (?) serves as a placeholder for a parameter.

**Core Note**

If the procedure returns output parameters, then the index of the input parameters must account for the number of output parameters.

- Calling a stored procedure from any Java program requires a five-step process:
  - ➦ Formulate a callable statement.
  - ➦ Create a *CallableStatement* object.
  - ➦ Register any *OUT* parameters.
  - ➦ Set any *IN* parameters.
  - ➦ Execute the callable statement.
- One can obtain a *CallableStatement* from a Connection with the *prepareCall* method, as below.

**Sample code Fragment**

```
String procedure = "{ ? = call procedure_name ( ?, ? ) }";

CallableStatement statement = connection.prepareCall(procedure);
```

- If the stored procedure returns *OUT* parameters, the JDBC type of each OUT parameter must be registered before the CallableStatement object can be executed.

### ✍ Task 11 : Explore the API

11.1 Define the syntax of method used for registering the JDBC type of each output parameter.

11.2 Identify the method which retrieves the value of a JDBC CHAR, VARCHAR, or LONGVARCHAR parameter as a *String* in the Java programming language.

11.3 Identify the method which sets the designated parameter to the given *Reader* object, which is the given number of characters long.

11.4 Method which retrieves the value of the designated JDBC BINARY or VARBINARY parameter as an array of byte values in the Java programming language.

11.5 Define the method which sets the value of the designated parameter with the given object.

## Registering the Output Parameter Types

You must register the JDBC type of each output parameter, using registerOutParameter, as follows,

**statement.registerOutParameter(*n*, *type*);**

where *n* corresponds to the ordered output parameter (using 1-based indexing), and *type* corresponds to a constant defined in the java.sql.Types class (Types.FLOAT, Types.DATE, etc.).

**Core Note**

A CallableStatement can return one *ResultSet* object or multiple *ResultSet* objects. Multiple ResultSet objects are handled using operations inherited from *Statement*.

## ✍ Fill in Queries:

1. Define the four general forms of calling a stored procedure.

2. Complete the method declaration: **public <*return type*> registerOutParameter (*Parameter1, Parameter2, Parameter3*) throws <*Exception*>**

3. Write the code to call the stored procedure SHOW_SUPPLIERS using the connection *con*.

4. Complete the method declaration: **public <*return type*> setCharacterStream (*Parameter1, Parameter2, Parameter3*) throws <*Exception*>**

5. Write the code to create *CallableStatement* objects that can produce ResultSet objects that are scrollable and updatable.

**Task 11 Workspace**

## ✍ Code 4 : Test the Coding Skills

### Code the Program

```
// GetFlightDetailsUsingCallable.java
package jdbcdemo;

/* Example which makes use of CallableStatement object to execute the stored procedure getFlightName that retrieves the Flight name based on the Flight id */.

//Import the appropriate classes and interfaces.



public class GetFlightDetailsUsingCallable {
  // Oracle driver name and database URL
  static final String ORACLE_DRIVER = "_____";

  static final String DB_URL = "_____";


  //  Database credentials
  static final String USER = "username";
  static final String PASS = "password";

  public static void main(String[] args) {
  Connection conn = null;
  CallableStatement stmt = null;
  try{
    //Register Oracle driver
    Class.forName("_____");

    //Open a connection
    conn = DriverManager._____(_____,_____,_____);

    //Execute a query
    System.out.println("Creating statement...");
    String sql = "{call getFlightName (?, ?)}";
    stmt = conn._____(_____);   //CallableStatement object.

    //Bind IN parameter first, then bind OUT parameter
    int flightID = 1002;
    stmt._____(___ , flightID); // This would set ID as 1002

    // Because second parameter is OUT so register it
    stmt._____(2, _____);
```

## Code the Program…..Continued

```
//Run the stored procedure.
System.out.println("Executing stored procedure..." );
stmt._____();

//Retrieve employee name with getXXX method
String flightName = stmt._____(____);
System.out.println("Flight Name with ID:" + flightID + " is " + flightName);

//Clean-up environment.

  _____;

  _____;

 } catch( _____ se ) { e._____(); }
}

} // End of Class.
```

## Unguided Practice

//Program to display the Flight price based on the Flight ID, using *CallableStatement* Object and **Type 4 driver**.