9 - JDBC

< Previous | Next >

§9.1 Fundamentals

- A JDBC driver implementation must provide implementation classes for the following interfaces: java.sql.Driver, java.sql.Connection, java.sql.Statement and java.sql.ResultSet.
- Provided by the JDBC API are: java.sql.DriverManager, java.sql.SQLException and java.sql.Date.
- Objects of the following classes/interfaces are independent of the implementation of a JDBC driver: javax.sql.RowSet, java.sql.Date, java.sql.Time and java.sql.SQLException.
- Applications no longer need to explicitly load JDBC drivers using Class.forName(). JDBC 4.0 Drivers must include the file META-INF/services/java.sql.Driver. This file contains the name of the JDBC drivers implementation of java.sql.Driver. Therefore, no java code in necessary to load the driver classes.
- When the method getConnection is called `DriverManager.getConnection()` , the DriverManager will attempt to locate a suitable driver from amongst those loaded at initialization and those loaded explicitly using the same classloader as the current applet or application.
- The method signatures of the DriverManager.getConnection() method:
  - `public static Connection getConnection(String url)`
  - `public static Connection getConnection(String url, String user, String password)`
  - `public static Connection getConnection(String url, java.util.Properties info)`

- You can set the user and password in a Properties object via the `setProperty()` method. The correct property names are "user" and "password". jdbc.driver is also an invalid property name but it is not an error to use it.
- DriverManager.registerDriver is a valid method but it takes java.sql.Driver instance and not a String. This method is used by the Driver class to register itself with the DriverManager. It need not be called by the application programmer. In JDBC 4.0, if you have the jar file that implements the Driver in the classpath, the Driver is automatically registered.
- There is no setClientInfo or connect method in Connection.
- When a connection is created, it is in auto-commit mode. i.e. auto-commit is enabled. This means that each individual SQL statement is treated as a transaction and is automatically committed right after it is completed.
- When auto-commit has been disabled in the given code (by calling c.setAutoCommit(false)), you have to explicitly commit the transaction to commit the changes to the database. The regular way to do this is to **call con.commit()**. Notice that commit method does not take any arguments.
- **There is no commit method in Statement**.
- Another way is to utilize the side effect of changing the auto-commit mode of the connection. If the setAutoCommit method is called during a transaction and the auto-commit mode is changed, the transaction is committed. If setAutoCommit is called and the auto-commit mode is not changed, the call is a no-op.
- The way to allow two or more statements to be grouped into a transaction is to disable the auto-commit mode. Since it is enabled by default, you have to explicitly disable it after creating a connection by calling con.setAutoCommit(false);
- Calling connection.rollback() without passing in a savepoint, will cause the whole transaction to rollback.
- If the connection.close() method is called and there is an active transaction, the results are implementation-defined. Therefore, calling this method does not guarantee that the active transaction will be committed.
- As per Section 6.2 of JDBC 4.1 Specification: A JDBC API implementation must support Entry Level SQL92 plus the SQL command Drop Table. Entry Level SQL92 represents a "floor" for the level of SQL that a JDBC API implementation must support. Access to features based on SQL99 or SQL:2003 should be provided in a way that is compatible with the relevant part of the SQL99 or SQL:2003 specification.

- A java.sql.Connection object implements a session with the database and provides methods to commit and rollback transactions.
- A java.sql.Statement object provides various execute methods to execute queries on the database.
- All queries run on the database.
- All queries executed on the database have to run in native SQL, whether fired through Statement, PreparedStatement, or from within a stored procedure.

# ResultSet:

- A java.sql.ResultSet object provides a cursor to fetch data from the database.
- The numbering of columns in a ResultSet starts with 1. The value of a field with type INT, can still be retrieved using getString(). Note that if a field is of type VARCHAR and if you try to retrieve the value using say getInt() or getDouble(), it may throw an exception at runtime if the value cannot be parsed into an Integer or Double.
- getString retrieves the value of the designated column in the current row of this ResultSet object as a String. If the value is SQL NULL, the value returned is null.
- rs.next() returns true only if there is a row left to be processed in the ResultSet. It moves the cursur in front of the next row and returns true. If there is no row left, it returns false. Thus, when you use rs.next() in a while loop, you are basically iterating through the rows returned by the query one by one.
- Statement has a setMaxRows method that limits the total number of rows returned by the ResultSet. If the query returns more rows than the limit, the extra rows are silently (i.e. without any exception) are ignored.
- java.sql.ResultSetMetaData gives you the information about the result of executing a query. You can retrieve this object by calling getMetaData() on ResultSet. ResultSetMetaData contains several methods that tell you about the ResultSet. Some important methods are: getColumnCount(), getColumnName(int col), getColumnLabel(int col), and getColumnType(int col). Remember that the column index starts from 1.
- If a Statement object from which a ResultSet was retrieved is closed, its current ResultSet object, if one exists, is also closed. If the ResultSet object that you are iterating is closed, you will get an exception saying: `Exception in thread "main" java.sql.SQLException: ResultSet not open`.
- JDBC 2.0 allows you to use ResultSet object to update an existing row and even insert new row in the database. For both the cases, the ResultSet must be updatable, which can be achieved by passing `ResultSet.CONCUR_UPDATABLE` while creating a Statement object:
  `stmt = con.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,  ResultSet.CONCUR_UPDATABLE);`
  or
  `stmt = con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,  ResultSet.CONCUR_UPDATABLE);`
- To update an existing row:

  1. First, go to the row you want to update. You can either iterate through a ResultSet to reach a particular row or just call `rs.absolute(int rowNumber)`.
  2. Now update the columns of the ResultSet with required values using `rs.updateXXX(columnNumber, value)` or `rs.updateXXX(columnName, value)` methods.
  3. Call `rs.updateRow();` If you call `rs.refreshRow()` without calling `updateRow()`, the updates will be lost.

- To insert a new Row:
  1. Call `rs.moveToInsertRow();` first. You can't insert a row without calling this method first.
  2. Use `rs.updateXXX` methods to update all column values. You must set values for all the columns.
  3. Call `rs.insertRow();`
  4. Call `rs.moveToCurrentRow();` to go back to the row where you were before calling moveToInsertRow.

- **IMPORTANT**: The exam will test you on implications of calling various methods out of sequence. For example, what happens when you call insertRow without first calling moveToInsertRow? (An SQLException will be thrown.) or what happens when you call refreshRow without first calling updateRow? (No exception but updates will be lost.).

# CallableStatement vs. PreparedStatement:

PreparedStatement

- A PreparedStatement is used for SQL statements that are executed multiple times with different values. For example, if you want to insert several values into a table, one after another, it is a lot easier with PreparedStatement:

```
ps = c.prepareStatement("INSERT INTO STUDENT VALUES (?, ?)");
//This is created only once
//Once created, the PreparedStatement is compiled automatically.
ps.setInt(1, 111);
ps.setString(2, "Bob");
ps.executeUpdate();

//Now change the parameter values and execute again.
ps.setInt(1, 112);
ps.setString(2, "Cathy");
ps.executeUpdate();
```

- PreparedStatement offers better performance when the same query is to be run multiple times with different parameter values.
- PreparedStatement has specific methods for additional SQL column type such as setBlob(int parameterIndex, Blob x) and setClob(int parameterIndex, Clob x).

### CallableStatement

- A CallableStatement is meant for executing a stored procedure, which has already been created in the database. For example:

```
//computeMatrixForSales is a stored procedure that has already been created in the database.
callableStatement = connection.prepareCall("{call computeMatrixForSales(?)}");
callableStatement.setInt(1, 1000);
callableStatement.executeUpdate();
```

- One advantage of CallableStatement is that it allows IN/OUT parameters.
- A CallableStatement is the only way for a JDBC program to execute stored procedures in the database if the procedure has in and out parameters.

< Previous | Next >