

Chapter 7:

JDBC Programming

JDBC

JDBC stands for Java Database Connectivity, which is a standard Java API for database-independent connectivity between the Java programming language and a wide range of databases.

The JDBC library includes APIs for each of the following tasks

- Making a connection to a database.
- Creating SQL or MySQL statements.
- Executing SQL or MySQL queries in the database.
- Viewing & Modifying the resulting records.

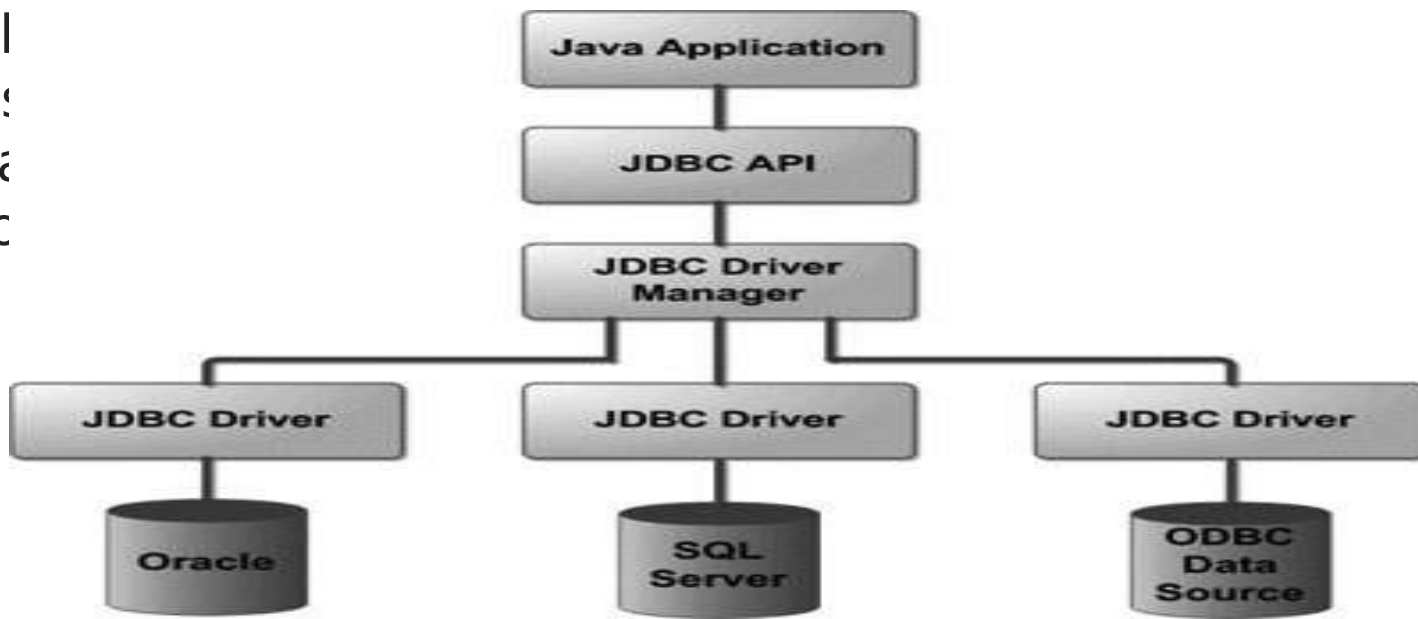
JDBC Architecture

JDBC Architecture consists of two layers –

- JDBC API: This provides the application-to-JDBC Manager connection.
- JDBC Driver API: This supports the JDBC Manager-to-Driver Connection.

The JDBC API uses a driver manager and database-specific drivers to provide transparent connectivity to heterogeneous databases.

TI
us
ca
tc



river is
ager is
nected

JDBC Components

Common JDBC Components

The JDBC API provides the following interfaces and classes –

1. **DriverManager:** This class manages a list of database drivers. Matches connection requests from the java application with the proper database driver using communication sub protocol. The first driver that recognizes a certain subprotocol under JDBC will be used to establish a database Connection.
2. **Driver:** This interface handles the communications with the database server. You will interact directly with Driver objects very rarely. Instead, you use DriverManager objects, which manages objects of this type. It also abstracts the details associated with working with Driver objects.

JDBC Components

3. **Connection:** This interface with all methods for contacting a database. The connection object represents communication context, i.e., all communication with database is through connection object only.
4. **Statement:** You use objects created from this interface to submit the SQL statements to the database. Some derived interfaces accept parameters in addition to executing stored procedures.
5. **ResultSet:** These objects hold data retrieved from a database after you execute an SQL query using Statement objects. It acts as an iterator to allow you to move through its data.
6. **SQLException:** This class handles any errors that

Steps for Creating JDBC Application

There are following six steps involved in building a JDBC application –

- Import the packages: Requires that you include the packages containing the JDBC classes needed for database programming. Most often, using `import java.sql.*` will suffice.
- Register the JDBC driver: Requires that you initialize a driver so you can open a communication channel with the database.
- Open a connection: Requires using the `DriverManager.getConnection()` method to create a `Connection` object, which represents a physical connection with the database.
- Execute a query: Requires using an object of type `Statement` for building and submitting an SQL statement to the database.
- Extract data from result set: Requires that you use the

JDBC Connection Establishment Steps

- Import JDBC Packages: Add import statements to your Java program to import required classes in your Java code.
- Register JDBC Driver: This step causes the JVM to load the desired driver implementation into memory so it can fulfill your JDBC requests.
- Database URL Formulation: This is to create a properly formatted address that points to the database to which you wish to connect.
- Create Connection Object: Finally, code a call to the DriverManager object's getConnection() method to establish actual database connection.

JDBC Connection Establishment Steps

1. Import JDBC Packages: Add import statements to your Java program to import required classes in your Java code.

```
import java.sql.*; // for standard JDBC programs  
import java.math.* ; // for BigDecimal and BigInteger support
```

2. Register JDBC Driver: This step causes the JVM to load the desired driver implementation into memory so it can fulfill your JDBC requests.

Approach I - Class.forName()

```
try {  
    Class.forName("oracle.jdbc.driver.OracleDriver");  
}  
catch(ClassNotFoundException ex) {  
    System.out.println("Error: unable to load driver class!");  
    System.exit(1);  
}
```

JDBC Connection Establishment Steps

Approach II - DriverManager.registerDriver()

```
try {  
    Driver myDriver = new oracle.jdbc.driver.OracleDriver();  
    DriverManager.registerDriver( myDriver );  
}  
catch(ClassNotFoundException ex) {  
    System.out.println("Error: unable to load driver class!");  
    System.exit(1);  
}
```

3. Database URL Formulation: This is to create a properly formatted address that points to the database to which you wish to connect.

JDBC Connection Establishment Steps

Following table lists down the popular JDBC driver names and

RDBMS	JDBC driver name	URL format
MySQL	com.mysql.jdbc.Driver	jdbc:mysql:// hostname/databaseName
ORACLE	oracle.jdbc.driver.OracleDriver	jdbc:oracle:thin:@ hostname:portNumber:databaseName
DB2	COM.ibm.db2.jdbc.net.DB2Driver	jdbc:db2: hostname:portNumber/databaseName
Sybase	com.sybase.jdbc.SybDriver	jdbc:sybase:Tds: hostname: portNumber/databaseName

4. Create Connection Object: Finally, code a call to the DriverManager object's getConnection() method to establish actual database connection.

JDBC Connection Establishment Steps

overloaded DriverManager.getConnection() methods –

- getConnection(String url)
 - getConnection(String url, Properties prop)
 - getConnection(String url, String user, String password)
- a. Using a Database URL with a username and password
String URL = "jdbc:oracle:thin:@amrood:1521:EMP";
String USER = "username";
String PASS = "password"
Connection conn = DriverManager.getConnection(URL,
USER, PASS);
- b. Using Only a Database URL
String URL =
"jdbc:oracle:thin:username/password@amrood:1521:EMP";
Connection conn = DriverManager.getConnection(URL);

JDBC Connection Establishment Steps

- c. Using a Database URL and a Properties Object
String URL = "jdbc:oracle:thin:@amrood:1521:EMP";
Properties info = new Properties();
info.put("user", "username");
info.put("password", "password");
Connection conn = DriverManager.getConnection(URL,
info);

Creating JDBC Statements

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

Interfaces	Recommended Use
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 and Closing The PreparedStatement Objects

The PreparedStatement statement gives you the flexibility of supplying arguments dynamically.

```
PreparedStatement pstmt = null;
```

```
try {
```

```
    String SQL = "Update Employees SET age = ? WHERE id = ?";
```

```
    pstmt = conn.prepareStatement(SQL);
```

```
    ...
```

```
}
```

```
catch (SQLException e) {
```

```
    ...
```

```
}
```

```
finally {
```

```
    pstmt.close();
```

```
}
```

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.

Inorder to set the parameter use the The setXXX() methods of the PreparedStatement bind values to the parameters, where XXX represents the Java data type of the value you wish to bind to the input parameter.

Void setXXX(int parameterIndex, xxx value)

Indexing starts from 1

Creating and Closing Statement Object

Statement object is created by using the Connection object's `createStatement( )` method and closed by using the `close` method

```
Statement stmt = null;
```

```
try {
```

```
    stmt = conn.createStatement( );
```

```
    . . .
```

```
}
```

```
catch (SQLException e) {
```

```
    . . .
```

```
}
```

```
finally {
```

```
    stmt.close()
```

```
}
```

Creating and Closing The CallableStatement Objects

The Connection.prepareCall() method is used 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 {
```

```
    cstmt.close();
```

```
}
```

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

Execute an SQL statement

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

1. `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.
2. `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.
3. `ResultSet executeQuery (String SQL)`: Returns a
ResultSet object. Use this method when you expect

ResultSet

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 –

1. Navigational methods: Used to move the cursor around.
2. Get methods: Used to view the data in the columns of the current row being pointed by the cursor.
3. 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.

The cursor is movable based on the properties of the `ResultSet`. These properties are designated when the corresponding `Statement` that generates the `ResultSet` is created

ResultSet

JDBC provides the following connection methods to create statements with desired ResultSet –

- `createStatement(String sql, int RSType, int RSConcurrency);`
- `prepareStatement(String sql, int RSType, int RSConcurrency);`
- `prepareCall(String sql, int RSType, int RSConcurrency);`

RSType indicates the type of a ResultSet object and RSConcurrency is one of two ResultSet constants for specifying whether a result set is read-only or updatable.

Type	Description
<u>ResultSet Type</u> ResultSet.TYPE_FORWARD_ONLY	The cursor can only move forward in the result set.(Default)
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

ResultSet

Concurrency of ResultSet

Concurrency	Description
ResultSet.CONCUR_READ_ONLY	Creates a read-only result set. This is the default
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 {  
    ....  
}
```

ResultSet

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

`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.

ResultSet

`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 does²²

Viewing a ResultSet

The ResultSet interface contains dozens of methods for getting the data of the current row.

There is a get method for each of the possible data types, and each get method has two versions –

One that takes in a column name.

One that takes in a column index.

For example, if the column you are interested in viewing contains an int, you need to use one of the getInt() methods of ResultSet –

public int getInt(String columnName) throws SQLException

Returns the int in the current row in the column named columnName.

public int getInt(int columnIndex) throws SQLException

Returns the int in the current row in the specified column index. The column index starts at 1, meaning the first column of a row is 1, the second column of a row is 2, and so on.

Updating a ResultSet

The ResultSet interface contains a collection of update methods for updating the data of a result set.

As with the get methods, there are two update methods for each data type –

One that takes in a column name.

One that takes in a column index.

For example, to update a String column of the current row of a result set, you would use one of the following updateString() methods –

**public void updateString(int columnIndex, String s)
throws SQLException**

Changes the String in the specified column to the value of s.

**public void updateString(String columnName, String s)
throws SQLException**

Similar to the previous method, except that the column is specified by its name instead of its index.

There are get methods in the ResultSet interface for each of the eight Java primitive types, as well as common types such as

Updating a Row in the Underlying DB

To update your changes to the row in the database, you need to invoke one of the following methods.

`public void updateRow()`

Updates the current row by updating the corresponding row in the database.

`public void deleteRow()`

Deletes the current row from the database

`public void refreshRow()`

Refreshes the data in the result set to reflect any recent changes in the database.

`public void cancelRowUpdates()`

Cancels any updates made on the current row.

`public void insertRow()`

Inserts a row into the database. This method can only be invoked when the cursor is pointing to the insert row.

RowSet

A RowSet is an extension of a Java ResultSet. Like a result set, a rowset is a Java object that holds tabular data. A RowSet object is scrollable and updatable by default,

One of the main uses of a RowSet object is to make a ResultSet object scrollable and updatable when the DBMS driver does not otherwise have those capabilities.

A RowSet object is considered either connected or disconnected.

- A connected RowSet object uses a JDBC driver to make a connection to a relational database and maintains that connection throughout its life span.
- A disconnected RowSet object makes a connection to a data source only to read in data from a ResultSet object or to write data back to the data source. They do not maintain a connection to a database. Instead, they reconnect only when making database changes or when repopulating the database with data

RowSet

The following interfaces extend the RowSet interface:

- JdbcRowSet: a connected RowSet, which can be considered as a wrapper to a scrollable and updatable ResultSet.
- CachedRowSet: defines the basic capabilities of disconnected RowSet, which can re-connect to the data source to update the changes made while disconnected.
- WebRowSet: sub-interface of CachedRowSet, which uses XML documents.
- JoinRowSet: sub-interface of WebRowSet, which can form the equivalent of a SQL JOIN without having to connect to a data source.
- FilteredRowSet: sub-interface of WebRowSet, which can perform query on a RowSet object without having to connect to a data source.

We only discuss JdbcRowset. The others are left as your reading assignment

Creating JdbcRowSet

Create a JdbcRowSet object by using an instance of RowSetFactory, which is created from the class RowSetProvider. The following example is from JdbcRowSetSample:

```
RowSetFactory factory =  
RowSetProvider.newFactory();  
try (JdbcRowSet myrowset = factory.createJdbcRowSet()) {  
    jdbcRs.setUrl(this.settings.urlString);  
    jdbcRs.setUsername(this.settings.userName);  
    jdbcRs.setPassword(this.settings.password);  
    jdbcRs.setCommand("select * from pay");  
    jdbcRs.execute();  
}
```

In order to get its data, a JdbcRowSet object first needs to connect to a database.

Note

The RowSetFactory interface contains methods to create the different types of RowSet implementations: createCachedRowSet

Navigating JdbcRowSet

A ResultSet object that is not scrollable can use only the next method to move its cursor forward, and it can move the cursor only forward from the first row to the last row.

A default JdbcRowSet object, however, can use all of the cursor movement methods defined in the ResultSet interface.

You could use methods such as first(), last(), previous(), next(), absolute(rowNumber), relative(rowNumber), beforeFirst(), afterLast(), moveToCurrentRow(), moveToInsertRow() to move or position the cursor. You can use getRow() to get the current row number. Take note that row number starts at 1.

```
jdbcRs.absolute(4);
```

```
jdbcRs.previous();
```

Updating JdbcRowSet Row

You update data in a JdbcRowSet object the same way you update data in a ResultSet object.

```
jdbcRs.absolute(3);
```

```
jdbcRs.updateDouble("Rate", 110.99);
```

```
jdbcRs.updateRow();
```

Inserting JdbcRowSet Row

```
jdbcRs.moveToInsertRow();  
jdbcRs.updateInt("Id", 110);  
jdbcRs.updateString("FirstName", "Nohe");  
jdbcRs.updateString("LastName", "Hailu");  
jdbcRs.updateString("Gender", "M");  
jdbcRs.updateDouble("Rate", 120);  
jdbcRs.updateDouble("Hours", 200);  
jdbcRs.insertRow();
```

Deleting JdbcRowSet Row

```
jdbcRs.last();  
jdbcRs.deleteRow();
```