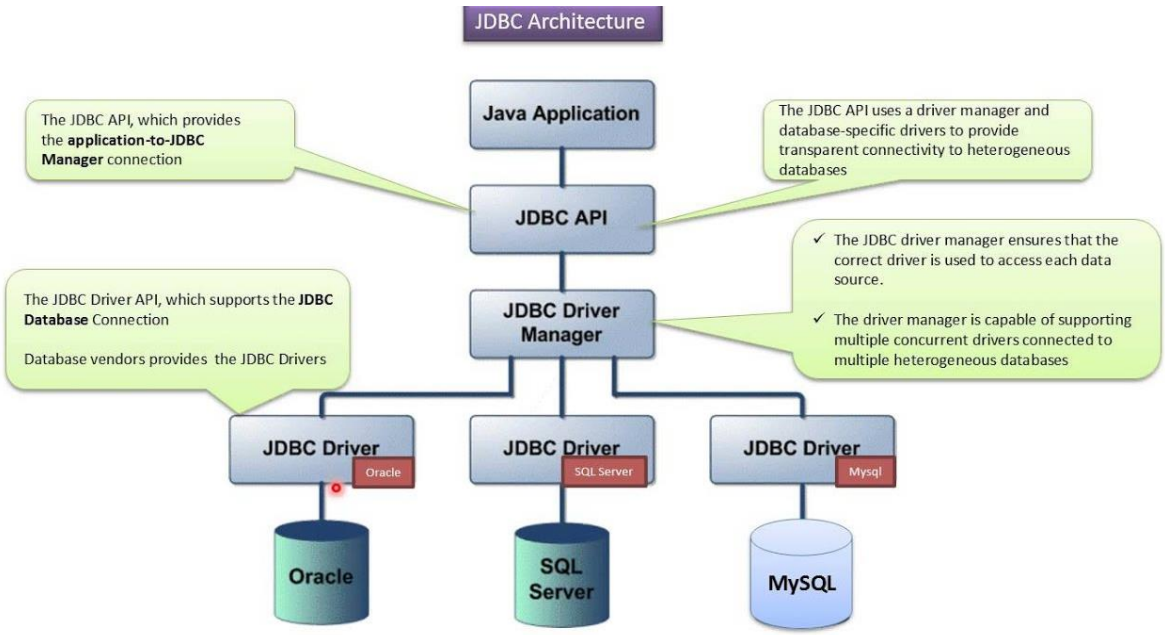
JDBC -Java Database Connectivity

Introduction

- **JDBC** is used to interact with various type of Database such as Oracle, MS Access, My SQL and SQL Server.
- **JDBC** can also be defined as the platform-independent interface between a relational database and Java programming.
- It allows java program to execute SQL statement and retrieve result from database.

JDBC Architecture



Types of drivers

Type-1 driver

- Type-1 driver or JDBC-ODBC bridge driver uses ODBC driver to connect to the database.
- The JDBC-ODBC bridge driver converts JDBC method calls into the ODBC function calls.
- Type-1 driver is also called Universal driver because it can be used to connect to any of the databases.
- As a common driver is used in order to interact with different databases, the data transferred through this driver is not so secured.
- The ODBC bridge driver is needed to be installed in individual client machines.
- Type-1 driver isn't written in java, that's why it isn't a portable driver.

Type-2 driver

- The Native API driver uses the client -side libraries of the database.
- This driver converts JDBC method calls into native calls of the database API.
- In order to interact with different database, this driver needs their local API, that's why data transfer is much more secure as compared to type-1 driver.
- Driver needs to be installed separately in individual client machines
- The Vendor client library needs to be installed on client machine.
- Type-2 driver isn't written in java, that's why it isn't a portable driver

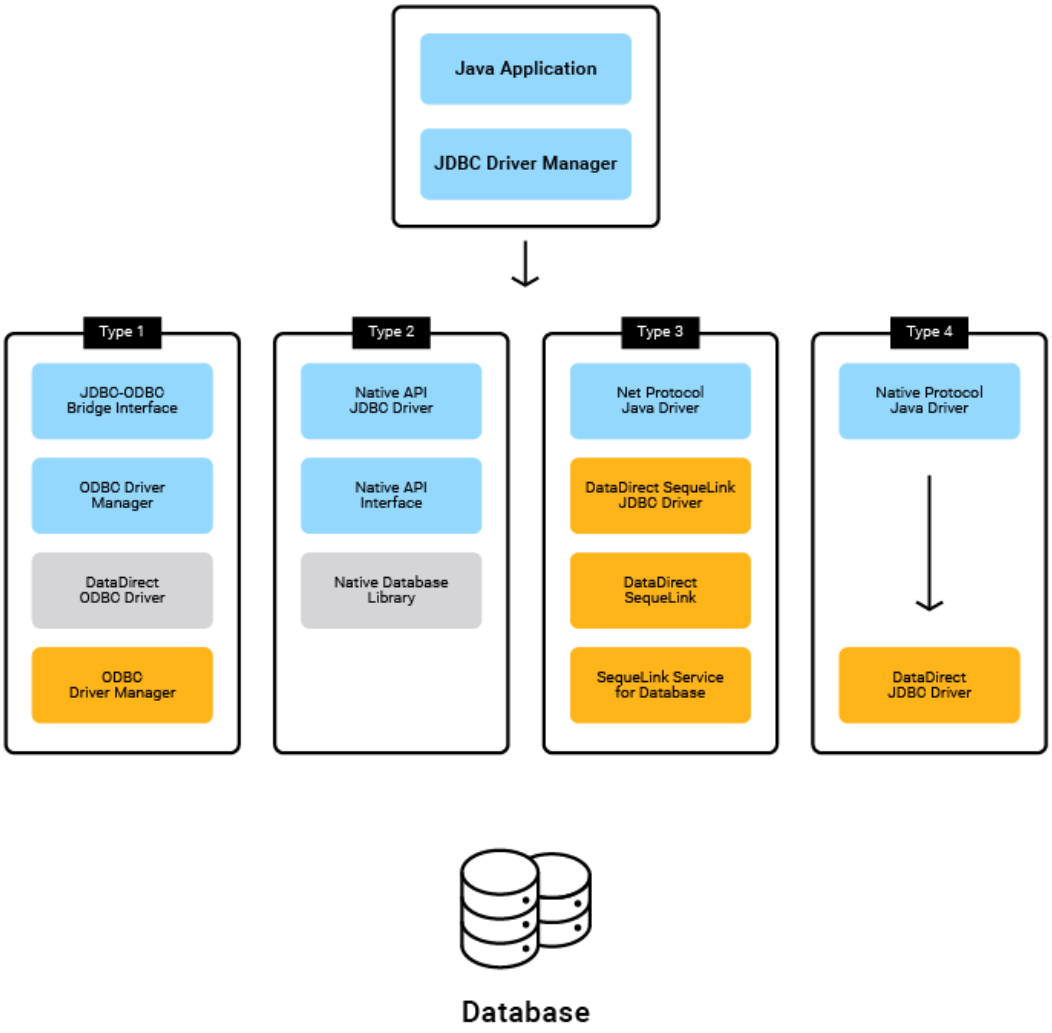
Type-3 driver

- The Network Protocol driver uses middleware **(application server)** that converts JDBC calls directly or indirectly into the vendor-specific database protocol.
- Here all the database connectivity drivers are present in a single server, hence no need of individual client-side installation.
- Fully written in Java, hence they are portable drivers.
- No client side library is required because of application server that can perform many tasks like auditing, load balancing, logging etc.
- Maintenance of Network Protocol driver becomes costly because it requires database-specific coding to be done in the middle tier.

Type-4 driver

- Type-4 driver is also called native protocol driver.
- This driver interacts directly with database.
- It does not require any native database library, that is why it is also known as Thin Driver.
- Does not require any native library and Middleware server, so no client-side or server-side installation.
- It is fully written in Java language, hence they are portable drivers.

The following figure shows a side-by-side comparison of the implementation of each JDBC driver type. All four implementations show a Java application or applet using the JDBC API to communicate through the JDBC Driver Manager with a specific JDBC driver.



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.

Eg :

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

Methods For Execute Queries

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.

Prepare statement :

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

Eg:

```
PreparedStatement pstmt = null;
try {
       String SQL = "Update Employees SET age = ? WHERE id = ?";
       pstmt = conn.prepareStatement(SQL);
       pstmt.setInt(1,value);
       ...
 }catch (SQLException e) {
   . . .
}
finally {
       pstmt.close();
}
```

- 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.
- The first marker represents position 1, the next position 2, and so forth.
- 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.

Callable statement :

- 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.

Eg. Store Procedure

```
DELIMITER $$
DROP PROCEDURE IF EXISTS `EMP`.`getEmpName` $$
CREATE PROCEDURE `EMP`.`getEmpName`
   (IN EMP_ID INT, OUT EMP_FIRST VARCHAR(255))
BEGIN
   SELECT first INTO EMP_FIRST
   FROM Employees
   WHERE ID = EMP_ID;
END $$
DELIMITER ;
```

Eg :

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

- use the setXXX() method that corresponds to the Java data type you are binding.
- 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.

Resultset :

- 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.

Type of ResultSet :

Type	Description

Forward only ResultSet	• The cursor can only move forward in the result set.
	• Default TYPE_FORWARD_ONLY
	• next() : moves pointer to next record and return true if record not found it will return false.
	• getXXX(), close() methods.
	
	Eg.
	`try {`
	`Statement stmt = conn.createStatement(ResultSet.TYPE_FORWARD_ONLY, ResultSet.CONCUR_READ_ONLY);`
	`}`
	`catch(Exception ex) {`
	`}`
	`finally {`
	`}`

Scroable Resultset	• public void beforeFirst() throws SQLException
	• public void afterLast() throws SQLException
	• public boolean first() throws SQLException
	• public void last() throws SQLException
	• public boolean absolute(int row) throws SQLException
	• public boolean relative(int row) throws SQLException
	• public boolean previous() throws SQLException
	• public boolean next() throws SQLException
	• public int getRow() throws SQLException
	• public void moveToInsertRow() throws SQLException
	• public void moveToCurrentRow() throws SQLException
	
	Eg.
	`try {`
	`Statement stmt = conn.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE, ResultSet.CONCUR_READ_ONLY);`
	`}`
	`catch(Exception ex) {`
	`}`
	`finally {`
	`}`

Scroable and Updatable	• moveToInsertRow() : First set will contain empty row in the resultset object • update() : The only corresponding row resultSet will update but make sure there is primary key set.

Concurrency of ResultSet :

- If you do not specify any Concurrency type, you will automatically get one that is CONCUR_READ_ONLY.

Concurrency	Description
ResultSet.CONCUR_READ_ONLY	Creates a read-only result set. This is the default
ResultSet.CONCUR_UPDATABLE	Creates an updateable result set.

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

S.N.	Methods & Description
1	**public int getInt(String columnName) throws SQLException** Returns the int in the current row in the column named columnName.
2	**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 –

S.N.	Methods & Description
1	**public void updateString(int columnIndex, String s) throws SQLException** Changes the String in the specified column to the value of string
2	**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 update methods for the eight primitive data types, as well as String, Object, URL, and the SQL data types in the java.sql package.
- Updating a row in the result set changes the columns of the current row in the ResultSet object, but not in the underlying database.
- To update your changes to the row in the database, you need to invoke one of the following methods.

S.N.	Methods & Description
1	**public void updateRow()**

	Updates the current row by updating the corresponding row in the database.
2	**public void deleteRow()** Deletes the current row from the database
3	**public void refreshRow()** Refreshes the data in the result set to reflect any recent changes in the database.
4	**public void cancelRowUpdates()** Cancels any updates made on the current row.
5	**public void insertRow()** Inserts a row into the database. This method can only be invoked when the cursor is pointing to the insert row.

Connection Modes

- If your JDBC Connection is in *auto-commit* mode, which it is by default, then every SQL statement is committed to the database upon its completion.
- That may be fine for simple applications, but there are three reasons why you may want to turn off the auto-commit and manage your own transactions –
    - To increase performance.
    - To maintain the integrity of business processes.
    - To use distributed transactions.
- Transactions enable you to control if, and when, changes are applied to the database.
- It treats a single SQL statement or a group of SQL statements as one logical unit, and if any statement fails, the whole transaction fails.
- To enable manual- transaction support instead of the *auto-commit* mode that the JDBC driver uses by default, use the Connection object's **setAutoCommit()** method.
- If you pass a boolean false to setAutoCommit( ), you turn off auto-commit.
- You can pass a boolean true to turn it back on again.

For example, if you have a Connection object named conn, code the following to turn off auto-commit –

```
conn.setAutoCommit(false);
```

Commit & Rollback

- Once you are done with your changes and you want to commit the changes then call **commit()** method on connection object as follows –

```
conn.commit( );
```

- Otherwise, to roll back updates to the database made using the Connection named conn, use the following code –

```
conn.rollback( );
```

The following example illustrates the use of a commit and rollback object –

```
try{
   //Assume a valid connection object conn
   conn.setAutoCommit(false);
   Statement stmt = conn.createStatement();

   String SQL = "INSERT INTO Employees  " +
                "VALUES (106, 20, 'Rita', 'Tez')";
   stmt.executeUpdate(SQL);
   //Submit a malformed SQL statement that breaks
   String SQL = "INSERTED IN Employees  " +
                "VALUES (107, 22, 'Sita', 'Singh')";
   stmt.executeUpdate(SQL);
   // If there is no error.
   conn.commit();
}catch(SQLException se){
   // If there is any error.
   conn.rollback();
}
```

In this case, none of the above INSERT statement would success and everything would be rolled back.

Using Savepoints :

The new JDBC 3.0 Savepoint interface gives you the additional transactional control. Most modern DBMS, support savepoints within their environments such as Oracle's PL/SQL.

When you set a savepoint you define a logical rollback point within a transaction. If an error occurs past a savepoint, you can use the rollback method to undo either all the changes or only the changes made after the savepoint.

The Connection object has two new methods that help you manage savepoints −

- **setSavepoint(String savepointName):** Defines a new savepoint. It also returns a Savepoint object.
- **releaseSavepoint(Savepoint savepointName):** Deletes a savepoint. Notice that it requires a Savepoint object as a parameter. This object is usually a savepoint generated by the setSavepoint() method.

There is one **rollback (String savepointName)** method, which rolls back work to the specified savepoint.

The following example illustrates the use of a Savepoint object −

```
try{
   //Assume a valid connection object conn
   conn.setAutoCommit(false);
   Statement stmt = conn.createStatement();

   //set a Savepoint
   Savepoint savepoint1 = conn.setSavepoint("Savepoint1");
   String SQL = "INSERT INTO Employees " +
                "VALUES (106, 20, 'Rita', 'Tez')";
   stmt.executeUpdate(SQL);
   //Submit a malformed SQL statement that breaks
   String SQL = "INSERTED IN Employees " +
                "VALUES (107, 22, 'Sita', 'Tez')";
   stmt.executeUpdate(SQL);
   // If there is no error, commit the changes.
   conn.commit();

}catch(SQLException se){
   // If there is any error.
   conn.rollback(savepoint1);
}
```

In this case, none of the above INSERT statement would success and everything would be rolled back.

Batch Updating :

A *JDBC batch update* is a batch of updates grouped together, and sent to the database in one *batch*, rather than sending the updates one by one.

Sending a batch of updates to the database in one go, is faster than sending them one by one, waiting for each one to finish. There is less network traffic involved in sending one batch of updates (only 1 round trip), and the database might be able to execute some of the updates in parallel. The speed up compared to executing the updates one by one, can be quite big.

You can batch both SQL inserts, updates and deletes. It does not make sense to batch select statements.

There are two ways to execute a *JDBC batch update*:

1. Using a **Statement**
2. Using a **PreparedStatement**

This JDBC batch update tutorial explains both ways in the following sections.

Statement Batch Updates

You can use a Statement object to execute batch updates. You do so using the addBatch() and executeBatch() methods. Here is an example:

```
Statement statement = null;

try{
    statement = connection.createStatement();
    statement.addBatch("update people set firstname='John' where id=123");
    statement.addBatch("update people set firstname='Eric' where id=456");
    statement.addBatch("update people set firstname='May'  where id=789");
    int[] recordsAffected = statement.executeBatch();
} finally {
    if(statement != null) statement.close();
}
```

First you add the SQL statements to be executed in the batch, using the addBatch() method.

Then you execute the SQL statements using the executeBatch(). The int[] array returned by the executeBatch() method is an array of int telling how many records were affected by each executed SQL statement in the batch.

PreparedStatement Batch Updates

You can also use a PreparedStatement object to execute batch updates. The PreparedStatementenables you to reuse the same SQL statement, and just insert new parameters into it, for each update to execute. Here is an example:

```
String sql = "update people set firstname=? , lastname=? where id=?";
PreparedStatement preparedStatement = null;
try{
    preparedStatement = connection.prepareStatement(sql);

    preparedStatement.setString(1, "Gary");
    preparedStatement.setString(2, "Larson");
    preparedStatement.setLong  (3, 123);

    preparedStatement.addBatch();
    preparedStatement.setString(1, "Stan");
    preparedStatement.setString(2, "Lee");
    preparedStatement.setLong  (3, 456);

    preparedStatement.addBatch();

    int[] affectedRecords = preparedStatement.executeBatch();

}finally {
    if(preparedStatement != null) {
        preparedStatement.close();
    }
}
```

First a PreparedStatement is created from an SQL statement with question marks in, to show where the parameter values are to be inserted into the SQL.

Second, each set of parameter values are inserted into the preparedStatement, and the addBatch()method is called. This adds the parameter values to the batch internally. You can now add another set of values, to be inserted into the SQL statement. Each set of parameters are inserted into the SQL and executed separately, once the full batch is sent to the database.

Third, the executeBatch() method is called, which executes all the batch updates. The SQL statement plus the parameter sets are sent to the database in one go. The int[] array returned by the executeBatch() method is an array of int telling how many records were affected by each executed SQL statement in the batch.

Blob/clob

Blob and Clob together are known as LOB(Large Object Type). Following are the major differences between Blob and Clob data types.

Blob	Clob
The full form of Blob is Binary Large Object.	The full form of Clob is Character Large Object.
This is used to store large binary data.	This is used to store large textual data.
This stores values in the form of binary streams.	This stores values in the form of character streams.
Using this you can store files like text files, PDF documents, word documents etc.	Using this you can stores files like videos, images, gifs, and audio files.
MySQL supports this with the following datatypes: • TINYBLOB • BLOB • MEDIUMBLOB • LONGBLOB	MySQL supports this with the following datatypes: • TINYTEXT • TEXT • MEDIUMTEXT • LONGTEXT
In JDBC API it is represented by java.sql.Blob Interface.	In JDBC it is represented by java.sql.Clob Interface.
The Blob object in JDBC points to the location of BLOB instead of holding its binary data.	The Blob object in JDBC points to the location of BLOB instead of holding its character data.
To store Blob JDBC (PreparedStatement) provides methods like: • setBlob() • setBinaryStream()	To store Clob JDBC (PreparedStatement) provides methods like: • setClob() • setCharacterStream()
And to retrieve (ResultSet) Blob it provides methods like: • getBlob() • getBinaryStream	And to retrieve (ResultSet) Clob it provides methods like: • getClob() • getCharacterStream()