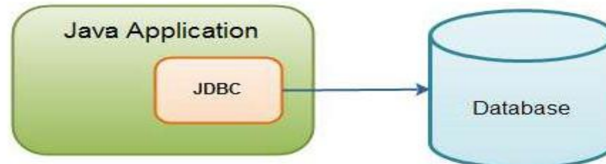
MODULE -5

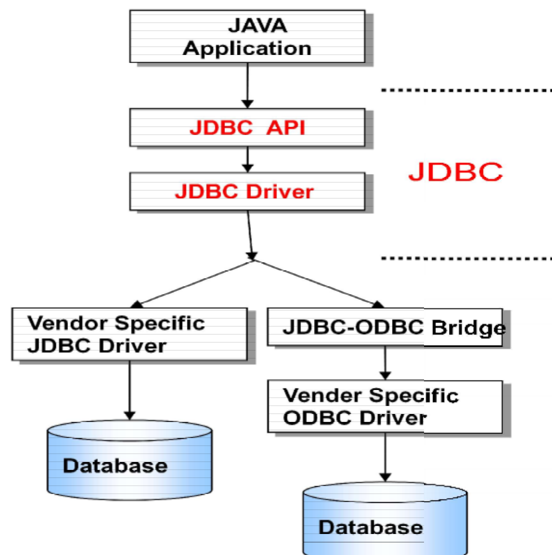
JDBC

- **Java Database Connectivity(JDBC)** is an **Application Programming Interface(API)** used to connect Java application with Database.



- 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 Model



- JDBC consists of two parts:
  - JDBC API, a purely Java-based API
  - JDBC driver
    - Communicates with vendor-specific drivers
  - JDBC driver classified into 4 categories

JDBC Driver

JDBC Driver is a software component that enables java application to interact with the database. There are 4 types of JDBC drivers:

1. JDBC-ODBC bridge driver
2. Native-API driver (partially java driver)
3. Network Protocol driver (fully java driver)
4. Thin driver (fully java driver)

1) JDBC-ODBC bridge driver

The 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.This is now discouraged because of thin driver.

Advantages:

- easy to use.
- can be easily connected to any database.

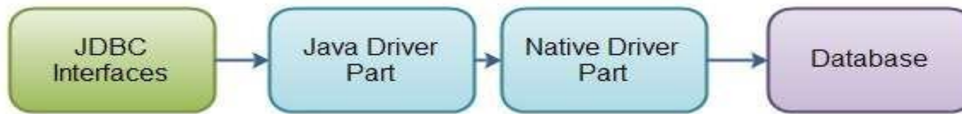
Disadvantages:

- Performance degraded because JDBC method call is converted into the ODBC function calls.
- The ODBC driver needs to be installed on the client machine.

---

2) Native-API driver

The Native API driver uses the client-side libraries of the database.The driver converts JDBC method calls into native calls of the database API. It is not written entirely in java.



Advantage:

- performance upgraded than JDBC-ODBC bridge driver.

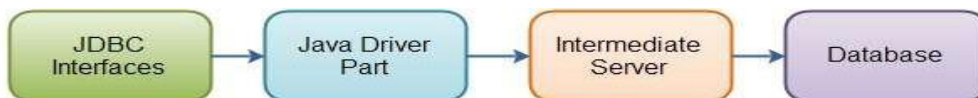
Disadvantage:

- The Native driver needs to be installed on the each client machine.
- The Vendor client library needs to be installed on client machine.

---

3) Network Protocol driver

The Network Protocol driver uses middleware (application server) that converts JDBC calls directly or indirectly into the vendor-specific database protocol. It is fully written in java.



Advantage:

- No client side library is required because of application server that can perform many tasks like auditing, load balancing, logging etc.

Disadvantages:

- Network support is required on client machine.
- Requires database-specific coding to be done in the middle tier.
- Maintenance of Network Protocol driver becomes costly because it requires database-specific coding to be done in the middle tier.

---

4) Thin driver

The thin driver converts JDBC calls directly into the vendor-specific database protocol. That is why it is known as thin driver. It is fully written in Java language.

Advantage:

- Better performance than all other drivers.

- No software is required at client side or server side.

Disadvantage:

- Drivers depends on the Database.

Various Database drivers

Database name	Driver name
MS Access	sun.jdbc.odbc.JdbcOdbcDriver
Oracle	oracle.jdbc.driver.OracleDriver
Microsoft SQL Server 2000	com.microsoft.sqlserver.jdbc.SQLServerDriver
MySQL	org.gjt.mm.mysql.Driver

JDBC Packages

JDBC API is contained in 2 packages.

- **import java.sql.*;**
  - contains core java data objects of JDBC API. It's a part of J2SE.
- **import javax.sql.* ;**
  - It extends java.sql and is in J2EE

5 Steps to connect to the database in java

There are 5 steps to connect any java application with the database in java using JDBC. They are as follows:

- Register the driver class
- Creating connection
- Creating statement
- Executing queries
- Closing connection

---

1) Register the driver class

The forName() method of Class class is used to register the driver class. This method is used to dynamically load the driver class.

Class.forName("driverClassName");

2) Create the connection object

The getConnection() method of DriverManager class is used to establish connection with the database.

Connection con = DriverManager.getConnection(url, user, password);

3) Create & Execute query

The createStatement() method of Connection interface is used to create statement. The object of statement is responsible to execute queries with the database.

The executeQuery() method of Statement interface is used to execute queries to the database. This method returns the object of ResultSet that can be used to get all the records of a table.

Statement st = con.createStatement();

ResultSet rs = st.executeQuery(sql);

4) Process data returned form DBMS

```
while(rs.next()){
        System.out.println(rs.getInt(1)+" "+rs.getString(2));
}
```

5) Close the connection object

By closing connection object statement and ResultSet will be closed automatically. The close() method of Connection interface is used to close the connection.

rs.close();

st.close();

con.close();

complete example

```
import java.sql.*;
import java.sql.*;
Class Dbconnection
```

```
{         public static void main(String args[])
  {       //Dynamically loads a driver class, for Oracle database
Class.forName("oracle.jdbc.driver.OracleDriver");

          //Establishes connection to database by obtaining a    Connection object
Connection con = DriverManager.getConnection( "jdbc:oracle:thin:@localhost:1521:XE",
“scott”, “tiger”);
         /*       jdbc- is API
                  oracle- is DB name
                  thin- is the driver
                  localhost-is sever name on which oracle is running (can giv IP address)
                  1521- is port number
                  XE- is oracle service name        */
Statement statement = con.createStatement();
String sql = "select * from users";
ResultSet result = statement.executeQuery(sql);
while(result.next()) {
   String name = result.getString("name");
   long age = result.getInt(“age");
   System.out.println(name);
   System.out.println(age);
}
result.close();
statement.close();
con.close();
}
```

Process results

When you execute an SQL query you get back a ResultSet. The ResultSet contains the result of your SQL query. The result is returned in rows with columns of data. You iterate the rows of the ResultSet like this:

```
while(result.next()) {
```

```
    String name = result.getString("name");
    long   age  = result.getLong ("age");

}
```

The ResultSet.next() method moves to the next row in the ResultSet, if there are anymore rows.If there are anymore rows, it returns true. If there were no more rows, it will return false.

You can also pass an index of the column instead, like this:

```
while(result.next()) {
   result.getString(1);
    result.getInt (2);
}
```

Statement Object

There are 3 types of statement objects to execute the sql query

Interfaces	Recommended Use
Statement	Use the 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 the when you plan to use the SQL statements many times. The PreparedStatement interface accepts input parameters at runtime.
CallableStatement	Use the when you want to access the database stored procedures. The CallableStatement interface can also accept runtime input parameters.

1. Statement object

- The **Statement interface** provides methods to execute queries with the database.
- The statement interface is a factory of ResultSet i.e. it provides factory method to get the object of ResultSet.

The important methods of Statement interface are as follows:

1) ResultSet executeQuery(String sql): is used to execute SELECT query. It returns the object of ResultSet.

Example show above

2) int executeUpdate(String sql): is used to execute specified query, it may be create, drop, insert, update, delete etc.

Snippet

```
Statement stmt=con.createStatement();
// for insert
int result=stmt.executeUpdate("insert into emp values(33,'Irfan',50000)");
// for update
int result=stmt.executeUpdate("update empset name='Vimal',salary=10000 where id=33");
// for delete
int result=stmt.executeUpdate("delete from emp where id=33");
System.out.println(result+" records affected");
con.close();
```

3) boolean execute(String sql): is used to execute queries that may return multiple results.

```
boolean status = stmt.execute(anyquery);
      if(status){
        //query is a select query.
        ResultSet rs = stmt.getResultSet()
```

2. PreparedStatement object

The PreparedStatement interface is a subinterface of Statement. It is used to execute parameterized query.

Improves performance: The performance of the application will be faster if you use PreparedStatement interface because query is compiled only once.

```
// PreparedStatement to insert record
PreparedStatement stmt=con.prepareStatement("insert into Emp values(?,?)");
        stmt.setInt(1,101);//1 specifies the first parameter in the query   stmt.setString(2,"Ratan");
        int i=stmt.executeUpdate();
```

```
System.out.println(i+" records inserted");
```

The **setXXX()** methods are used to supply values to the parameters

All of the **Statement object's** methods for interacting with the database execute(), executeQuery(), and executeUpdate() also work with the PreparedStatement object.

```
// PreparedStatement to update record
PreparedStatement stmt=con.prepareStatement("update emp set name=? where id=?");
stmt.setString(1," Ratan ");//1 specifies the first parameter in the query i.e. name
stmt.setInt(2,101);
int i=stmt.executeUpdate();
System.out.println(i+" records updated");
```

```
// PreparedStatement to delete record
PreparedStatement stmt=con.prepareStatement("delete from emp where id=?");
stmt.setInt(1,101);
int i=stmt.executeUpdate();
System.out.println(i+" records deleted");
```

3. CallableStatement Objects

CallableStatement interface is used to call the **stored procedures and functions**.

Suppose you need the get the age of the employee based on the date of birth, you may create a function that receives date as the input and returns age of the employee as the output.

Snippet

```
String SQL = "{call getEmpName (?, ?)}"; // stored procedure called
  cs = conn.prepareCall (SQL);
cs.setInt(100);
// resisterOutParameter() used to register OUT type used by stored procedure
cs.resisterOutParameter(2, VARCHAR);
cs.execute();
String Name = cs.getString(1);
Cs.close();
```

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.

Parameter	Description
IN	Data that needs to be passed to stored procedure. values to IN parameters with the setXXX() methods.
OUT	contains value supplied by the SQL statement it returns. You retrieve values from the OUT parameters with the getXXX() methods.
INOUT	A parameter that provides both input and output values.

ResultSet interface

The object of ResultSet maintains a cursor pointing to a row of a table. Initially, cursor points to before the first row.

Method Name	Description
first()	Moves the cursor to the first row
last()	Moves the cursor to the last row.
previous()	Moves the cursor to the previous row. This method returns false if the previous row is off the result set
next()	Moves the cursor to the next row. This method returns false if there are no more rows in the result set

absolute(int row)	Moves the cursor to the specified row
relative(int row)	Moves the cursor the given number of rows from where it currently is pointing.
getRow()	Returns the row number that the cursor is pointing to.
beforeFirst()	Moves the cursor to just before the first row
afterLast()	Moves the cursor to just after the last row

When you create a ResultSet there are three attributes you can set. These are:

1. **Type**
   1. **ResultSet.TYPE_FORWARD_ONLY** (default type)- TYPE_FORWARD_ONLY means that the ResultSet can only be navigated forward
   2. **ResultSet.TYPE_SCROLL_INSENSITIVE-** TYPE_SCROLL_INSENSITIVE means that the ResultSet can be navigated (scrolled) both forward and backwards. The ResultSet is insensitive to changes while the ResultSet is open. That is, if a record in the ResultSet is changed in the database by another thread or process, it will not be reflected in already opened ResulsSet's of this type.
   3. **ResultSet.TYPE_SCROLL_SENSITIVE-** means that the ResultSet can be navigated (scrolled) both forward and backwards. The ResultSet is sensitive to changes in the underlying data source while the ResultSet is open.
2. **Concurrency** determines whether the ResultSet can be updated, or only read.
   1. **ResultSet.CONCUR_READ_ONLY-** means that the ResultSet can only be read.

   2. **ResultSet.CONCUR_UPDATABLE-** means that the ResultSet can be both read and updated.

//Inserting Rows into a ResultSet u have updateXXX( )

rs.updateString (1, "raj");

rs.updateInt (2, 55);

rs.**insertRow(); // call this method**

//updating Rows into a ResultSet u have updateXXX( )

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

rs.updateString ("name" , "ram");

rs.updateInt ("age" , 55);

rs.**updateRow();// call this method**

//Delete Rows from a ResultSet

Deletes the current row from the database

rs.**deleteRow(3);** //delete current row

3. **<u>Holdability</u>-** determines if a ResultSet is closed when the commit() method of the underlying connection is called.
   1. **ResultSet.CLOSE_CURSORS_OVER_COMMIT-** means that all ResultSet instances are closed when connection.commit() method is called on the connection that created the ResultSet.
   2. **ResultSet.HOLD_CURSORS_OVER_COMMIT-** means that the ResultSet is kept open when the connection.commit() method is called on the connection that created the ResultSet.

The HOLD_CURSORS_OVER_COMMIT holdability might be useful if you use the ResultSet to update values in the database. Thus, you can open a ResultSet, update rows in it, call connection.commit() and still keep the same ResultSet open for future transactions on the same rows.

- U can set resultset attribute for Statement or PreparedStatement, like this:

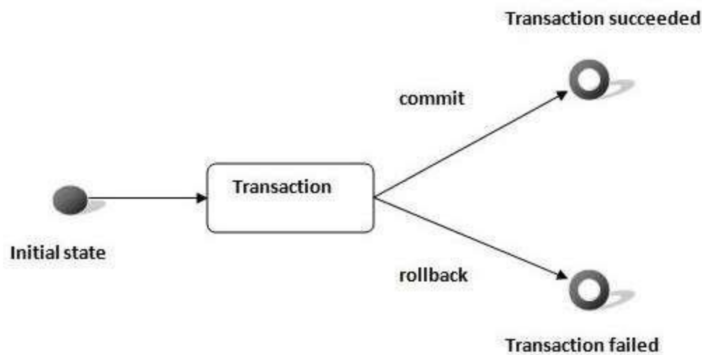
```
Statement statement = connection.createStatement(
   ResultSet.TYPE_FORWARD_ONLY, ResultSet.CONCUR_READ_ONLY,
   ResultSet.CLOSE_CURSORS_OVER_COMMIT
  );


PreparedStatement statement = connection.prepareStatement(sql,
   ResultSet.TYPE_FORWARD_ONLY,ResultSet.CONCUR_READ_ONLY,
   ResultSet.CLOSE_CURSORS_OVER_COMMIT
  );
```

Transaction

- A transaction is a set of actions to be carried out as a single, atomic action. Either all of the actions are carried out, or none of them are.

- *Advantage **is** fast performance It makes the performance fast because database is hit at the time of commit.*



In JDBC, **Connection interface** provides methods to manage transaction.

Method	Description
void setAutoCommit(boolean status)	It is true bydefault means each transaction is committed bydefault.
void commit()	commits the transaction.
void rollback()	cancels the transaction.

```
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();
}
```

Savepoints

- 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.

```
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);
}
```

Batch Processing in JDBC

Instead of executing a single query, we can execute a batch (group) of queries. It makes the performance fast.

Methods of Statement interface

The required methods for batch processing are given below:

Method	Description
void addBatch(String query)	It adds query into batch.
int[] executeBatch()	It executes the batch of queries.
clearBatch()	Clears the batch

```
Statement stmt=con.createStatement();
stmt.addBatch("insert into user values(190,'abhi',40000)");
stmt.addBatch("insert into user values(191,'umesh',50000)");
int[] count = stmt.executeBatch();
// u can get number of sql stmt that was executed by count[] array.
```

MetaData

DatabaseMetaData interface provides methods to get meta data of a database such as database product name, database product version, driver name, name of total number of tables, name of total number of views etc.

methods of DatabaseMetaData interface

- **String getDriverName():** it returns the name of the JDBC driver.
- **String getDriverVersion():** it returns the version number of the JDBC driver.
- **String getUserName():** it returns the username of the database.
- **String getDatabaseProductName():** it returns the product name of the database.
- **String getDatabaseProductVersion():** it returns the product version of the database.
- **String getSchemas()**
- String getPrimaryKeys()
- String getProcedures()
- String getTables()

```
DatabaseMetaData dbmd=con.getMetaData();
 System.out.println("Driver Name: "+dbmd.getDriverName());
System.out.println("Driver Version: "+dbmd.getDriverVersion());
System.out.println("UserName: "+dbmd.getUserName());
System.out.println("Database Product Name: "+dbmd.getDatabaseProductName());
System.out.println("Database Product
```

ResultSetMetaData Interface

The metadata means data about data i.e. we can get further information from the data.

If you have to get metadata of a table like total number of column, column name, column type etc. , ResultSetMetaData interface is useful because it provides methods to get metadata from the ResultSet object.

methods of ResultSetMetaData interface

Method	Description
public int getColumnCount()	it returns the total number of columns in the ResultSet object.
public String getColumnName(int index)	it returns the column name of the specified column index.
public String getColumnTypeName(int	it returns the column type name for the specified index.

index)	
public String getTableName(int index)	it returns the table name for the specified column index.

Exceptions

JDBC methods throws 3 types of exceptions

1. SQLException
2. SQLWarnings
3. DataTruncation.

1. **SQLException Methods** - An SQLException can occur both in the driver and the database.

Method	Description
getErrorCode( )	Gets the error number associated with the exception.
getMessage( )	Gets the JDBC driver's error message for an error, handled by the driver or gets the Oracle error number and message for a database error.
getNextException( )	Gets the next Exception object in the exception chain.

2. **SQLWarnings** - An exception that provides information on database access warnings.

getWarnings()-Retrieves the first warning reported by calls on this Connection object.

getNextWarning()-Retrieves subsequent warnings.

3. **DataTruncation** – this exception is thrown when a data values is unexpectedly truncated