Java Data Base Connectivity

(JDBC)

By

Apparao G

What is JDBC?

Driver Types

Connection Interface

Statement Interface

PreparedStatement Interface

ResultSet Interface

CallableStatement Interface

Transaction Management

Batch Update

ResultSetMetaData Interface

DatabaseMetaData Interface

Before APIs like JDBC and ODBC, database connectivity was tedious:

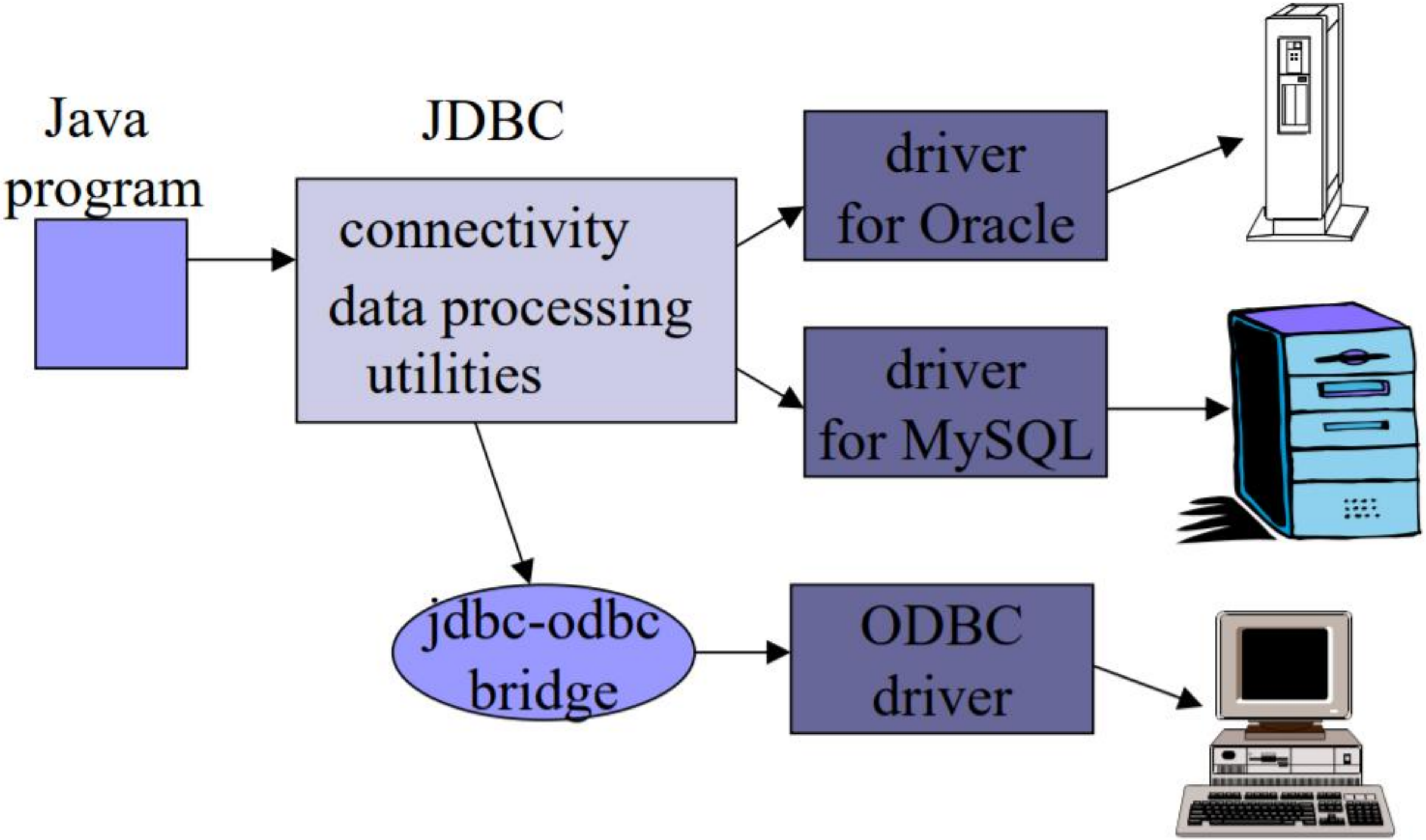
> Database vendor provided function libraries for database access.

> Connectivity library was proprietary.

> Data access portions had to be rewritten with changes in the application.

> Application developers were stuck with a particular database product for a given application

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


Java program
JDBC
connectivity
data processing
utilities
driver for Oracle
driver for MySQL
jdbc-odbc bridge
ODBC driver

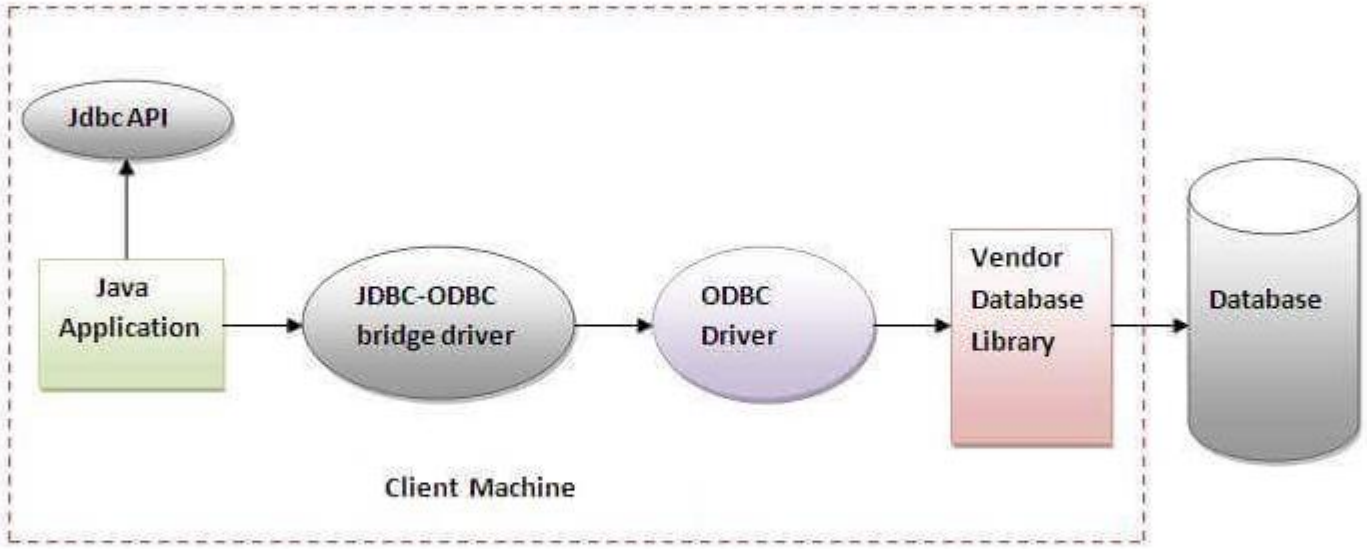
JDBC Driver:

- JDBC Driver is a software component that enables java application to interact with the database.

There are 4 types of JDBC drivers:

- Type1 (JDBC-ODBC bridge driver)
- Type2 (Native-API driver (partially java driver))
- Type3 (Network Protocol driver (fully java driver))
- Type4 (Thin driver (fully java driver))

1. Type1 (JDBC-ODBC bridge driver):



1.ODBC (Open Database Connectivity) is a Microsoft standard from the mid 1990’s.

.It is an API that allows C/C++ programs to execute SQL inside databases.

3.ODBC is supported by many products.

4.The JDBC-ODBC bridge allows Java code to use the C/C++ interface of ODBC.

it means that JDBC can access many different database products

5.The layers of translation (Java --> C -->SQL) can slow down execution.

6.The JDBC-ODBC bridge comes free with the J2SE:

called “sun.jdbc.odbc.JdbcOdbcDriver”

7.The ODBC driver for Microsoft Access comes with MS Office.

so it is easy to connect Java and Access.

Advantages:

- Easy to use.
- It can be easily connected to any database.

Disadvantages:

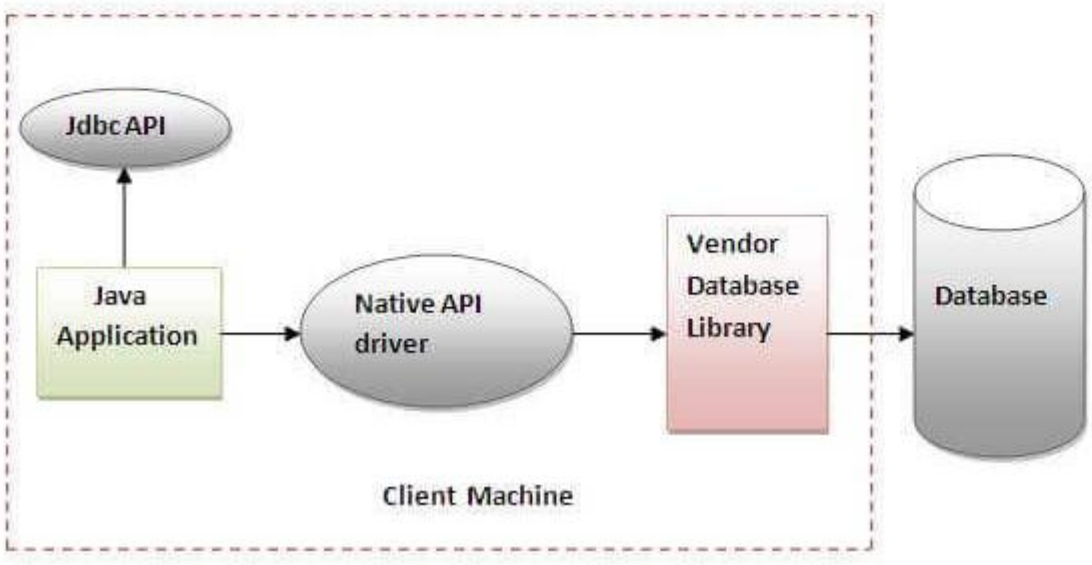
- Performance degraded because layers of translation (java->C->SQL).
- The ODBC driver needs to be installed on the client machine.

2. Type2 (Native-API driver):

In this driver, JDBC API calls are converted into native C/C++ API calls, which are unique to the database.

These drivers are typically provided by the database vendors and used in the same manner as the JDBC-ODBC Bridge.

The vendor-specific driver must be installed on each client machine.



Advantage:

- Performance improved compare to JDBC-ODBC bridge driver.

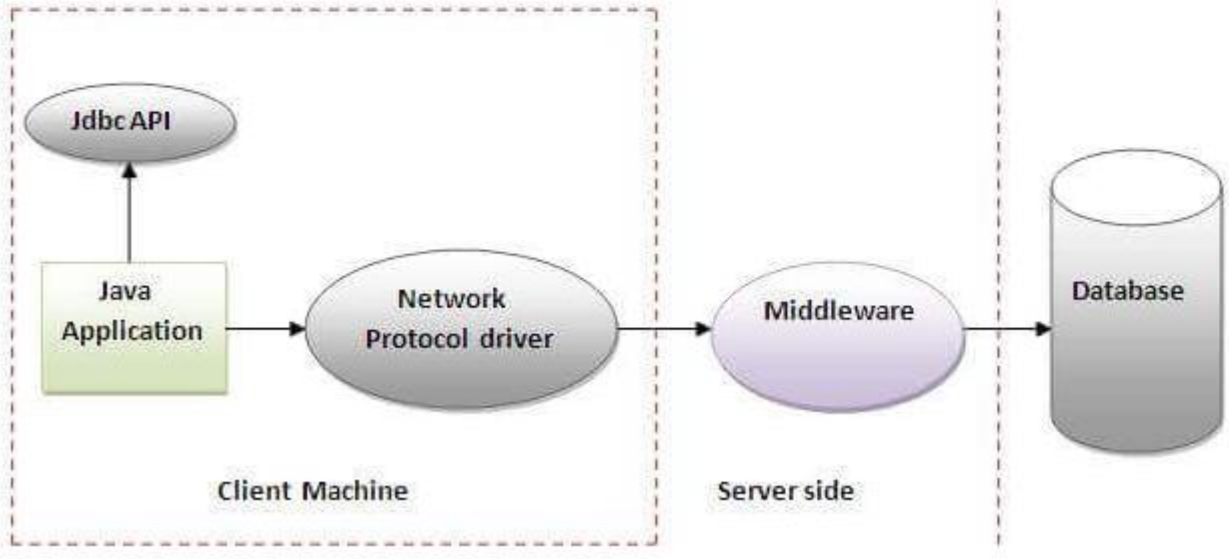
Disadvantage:

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

3. Type3 (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 written in java language.

This kind of driver is extremely flexible, since it requires no code installed on the client and a single driver can actually provide access to multiple databases.



Advantage:

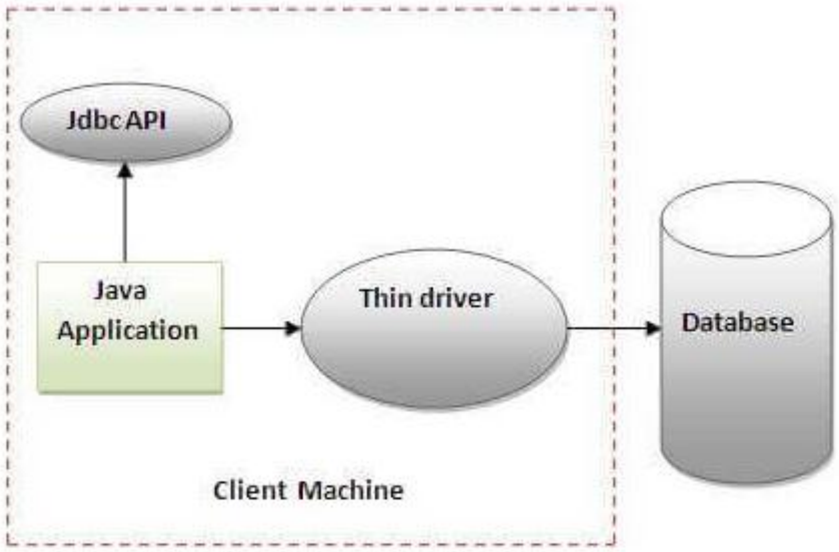
- No client side library is required because of application server that can perform many tasks like auditing, load balancing, logging etc.

Disadvantages:

- Network support is required on client machine.
- Requires database-specific coding to be done in the middle tier.

4.Type4 (Thin driver):
It is a pure Java-based driver communicates directly with the vendor's database through socket connection. This is the highest performance driver available for the database and is usually provided by the vendor itself.

This kind of driver is extremely flexible, you don't need to install special software on the client or server.



Advantage:

- Better performance than all other drivers.
- No software is required at client side or server side.

Disadvantage:

- Drivers depend on the Database.

How to select/choose the Driver?

- If you are accessing one type of database, such as Oracle, Sybase, or IBM, the preferred driver type is 4.
- If your Java application is accessing multiple types of databases at the same time, type 3 is the preferred driver.
- Type 2 drivers are useful in situations, where a type 3 or type 4 driver is not available yet for your database.
- The type 1 driver is not considered a deployment-level driver, and is typically used for development and testing purposes only.

Steps to connect Database:

Step 1. Load/Register the JDBC driver

Step 2. Specify the name and location of the database (URL) being used

Step 3. Create Connection object

Step 4. Create Statement object.

Step 5. Execute a SQL query and get the results in a ResultSet object

Step 6. Closing the ResultSet, Statement and Connection objects

All jdbc related classes and interfaces available in java.sql.* package.

1.Register/load driver:

The most common approach to register a driver is to use Java's **Class.forName()** method, to dynamically load the driver's class file into memory, which automatically registers it.

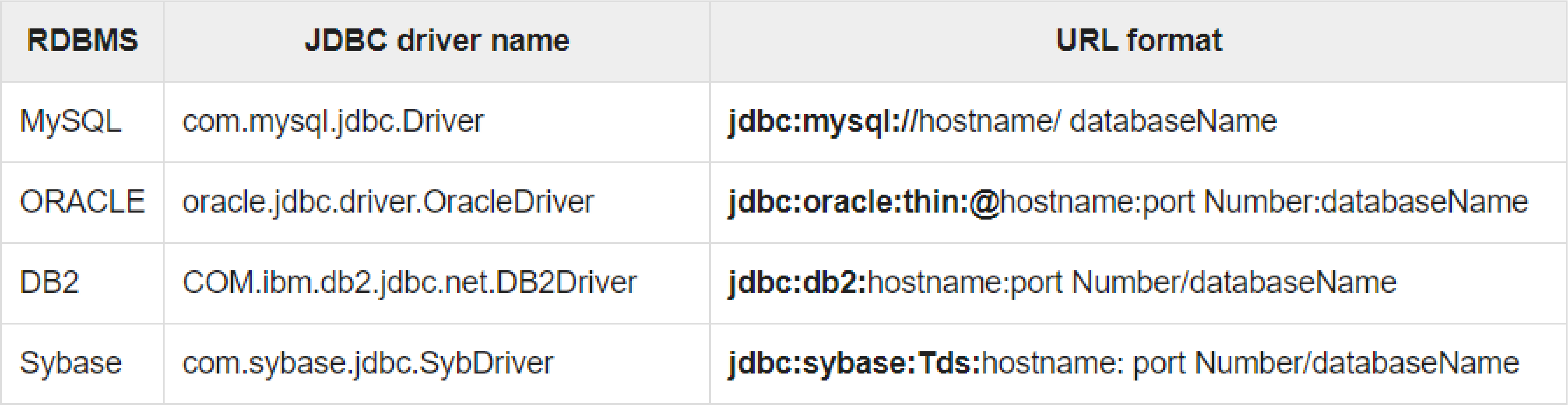
public static void forName(String className)throws ClassNotFoundException

Example for oracle driver:

Class.forName("oracle.jdbc.driver.OracleDriver");

2. Specify the name and location of the database (URL):

Below are popular JDBC driver names and database URL



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

3. Create Connection:

DriverManager.getConnection() method is used to create the Connection object.

Below ate the overloaded DriverManager.getConnection() methods:

```
public static Connection getConnection(String url) throws SQLException;
public static Connection getConnection(String url, Properties prop) throws SQLException;
public static Connection getConnection(String url, String user, String password) throws SQLException;
```

Example to establish connection to oracle database:

```
String url = "jdbc:oracle:thin:username/password@hostname:port number:dbname";
Connection conn = DriverManager.getConnection(url);

String url = "jdbc:oracle:thin:@hostname:port number:dbname";
Properties info = new Properties( );
info.put( "user", "username" );
info.put( "password", "password" );
Connection conn = DriverManager.getConnection(url, info);

Connection connection=DriverManager.getConnection(
                    "jdbc:oracle:thin:@localhost:1521:xe","root",”root");
```

Note: Generally oracle runs on 1521 port number.

4. Create Statement object:

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

Syntax:

```
public Statement createStatement()throws SQLException
```

Example:

```
Statement stmt=con.createStatement();
```

5. Execute query:

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

Syntax:

```
public ResultSet executeQuery(String sql) throws SQLException
```

Example:

```
ResultSet rs=stmt.executeQuery("select eno,ename from emp");
  while(rs.next()){
     System.out.println(rs.getInt(1)+" "+rs.getString(2));
  }
```

6. Clean up resources:

Write the clean up code in finally block. A *finally* block always executes, regardless of an exception occurs or not.

If you closing connection object statement and ResultSet will be closed automatically.

The close() method of Connection interface is used to close the connection.

Syntax:

public void close()**throws** SQLException

Example:

connection.close();

JDBC Example:

```
import java.sql.*;
class EployeeTest{
public static void main(String args[]){
try{
Class.forName("oracle.jdbc.driver.OracleDriver");
Connection con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe",“root",“root");
Statement stmt=con.createStatement();
ResultSet rs=stmt.executeQuery("select eno,fname,lname from emp");
    while(rs.next())  {
         System.out.println(rs.getInt(1)+"  "+rs.getString(2)+"  "+rs.getString(3));
    }
} catch(Exception e){  System.out.println(e);
}
finally {
  con.close();
}
}
}
```

Statement Interface:

It is useful when you are using static SQL statements at runtime. The Statement interface cannot accept parameters.

Commonly used methods of Statement interface:

public 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.

public 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.

public 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.

public int[] executeBatch(): It is used to execute batch of commands.

Example: Implement create, update and delete command example using statement interface.

Statement interface example:

```
import java.sql.*;
class  StatementExample {
public static void main(String args[]){
try {
Class.forName("oracle.jdbc.driver.OracleDriver");
Connection con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe","root","root");
Statement stmt=con.createStatement();
int result = stmt.executeUpdate("insert into emp values(33,’Ramesh',50000)");
System.out.println(“rows affected:“+result);
}catch(SQLException e) { System.out.println(e.getMessage())}
finally { con.close(); }
}
}
```

Note : For Update replace insert query with update query:

```
int result=stmt.executeUpdate("update emp set salary=10000 where id=33");
```

For delete replace insert query with delete query.

```
int result=stmt.executeUpdate("delete from emp where id=33");
```

PreparedStatement:

We will use PreparedStatement when we want to execute SQL statements many times. The PreparedStatement interface accepts input parameters at runtime.

PreparedStatement interface is a subinterface of Statement.

prepareStatement() method of Connection interface is used to return the object of PreparedStatement.

Syntax:

public PreparedStatement prepareStatement(String query)**throws** SQLException;

Example:

```
PreparedStatement pstmt = null;

try {

   String sql = “update emp set age = ?  where id = ?";

   pstmt = conn.prepareStatement(sql);

   pstmt.setInt(1,20); pstmt.setInt(2,999);
  int i=stmt.executeUpdate();

   System.out.println(“Records updated:”+i);

} catch (SQLException e) { System.out.println(e);}

finally { pstmt.close(); }
```

Important methods of preparedstatement:

public void setInt(int paramIndex, int value) :Sets the integer value to the given parameter index.

public void setString(int paramIndex, String value) :Sets the String value to the given parameter index.

public void setFloat(int paramIndex, float value) :Sets the float value to the given parameter index.

public void setDouble(int paramIndex, double value) :Sets the double value to the given parameter index.

public int executeUpdate() :It is executes create, insert, update and delete sql statements.

public ResultSet executeQuery():It executes the select query. It returns an instance of ResultSet.

Note: We will use PreparedStatement interface in jdbc application to sent an outline of the sql command with certain parameters, so that the database engine will prepare a query plan only once with a set of parameters. Same query plan will be executed again and again by receiving the parameters values from the jdbc application at different times, which will reduce the overhead at the data base engine.

ResultSet Interface:

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:

Navigational methods: Used to move the cursor around.

Get methods: Used to view the data in the columns of the current row being pointed by the cursor.

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.

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

createStatement(int RSType, int RSConcurrency);

prepareStatement(String SQL, int RSType, int RSConcurrency);

prepareCall(String sql, int RSType, int RSConcurrency);

The first argument indicates the type of a ResultSet object and the second argument is one of two ResultSet constants for specifying whether a result set is read-only or updatable.

Type of ResultSet:

If you do not specify any ResultSet type, you will automatically get one that is TYPE_FORWARD_ONLY.

ResultSet.TYPE_FORWARD_ONLY : The cursor can only move forward in the result set.

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 others to the database that occur after the result set was created.

Concurrency of ResultSet:

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

ResultSet.CONCUR_READ_ONLY : Creates a read-only result set. This is the default

ResultSet.CONCUR_UPDATABLE : Creates an updateable result set.

Example: Statement object to create a forward-only, read only ResultSet object.

```
try {
  Statement stmt = conn.createStatement(
                       ResultSet.TYPE_FORWARD_ONLY,
                       ResultSet.CONCUR_READ_ONLY);
}
catch(Exception ex) {
  ....
}
finally {
  ....
}
```

Navigating a Result Set:

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.

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.

Viewing a Result Set:

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

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.

Similarly, there are get methods in the ResultSet interface for each of the eight Java primitive types, as well as common types such as java.lang.String, java.lang.Object, and java.net.URL.

There are also methods for getting SQL data types java.sql.Date, java.sql.Time, java.sql.TimeStamp, java.sql.Clob, and java.sql.Blob. Check the documentation for more information about using these SQL data types.

Updating a Result Set:

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

There are two update methods for each data type:

For example, to update a String column of the current row of a result set, 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 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.

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.

Sample code:

```
// Execute a query
String sql = "SELECT id, first, last, age FROM EMP";
ResultSet rs = stmt.executeQuery(sql);
System.out.println("Print the result before set new ages..");
//Loop through result set and add 6 to age value.Move to BFR postion so while-loop works properly
rs.beforeFirst();
//Extract data from result set
while(rs.next()){
  int newAge = rs.getInt("age") + 6;
  rs.updateDouble( "age", newAge );
  rs.updateRow();
}
System.out.println("Print the result set showing new ages..");
// Insert a record into the table.Move to insert row and add column data with updateXXX()
System.out.println("Inserting a new record...");
rs.moveToInsertRow();
rs.updateInt("id",104);
rs.updateString("first","John");
rs.updateString("last","Paul");
rs.updateInt("age",40);
//Commit row
rs.insertRow();
```

Callable Statement:

when you want to access the database stored procedures, CallableStatement is good option to use.

It accept runtime input parameters.

Sample Oracle stored procedure:

```
create or replace PROCEDURE getEmpSal
  (EMP_ID IN NUMBER, salary OUT NUMBER) AS
BEGIN
  SELECT sal INTO salary FROM employee WHERE eid = EMP_ID;
END;
```

Three types of parameters exist: IN, OUT, and INOUT

IN : A parameter whose value is unknown when the SQL statement is created. You bind values to IN parameters with the setXXX() methods.

OUT : A parameter whose value is supplied by the SQL statement it returns. You retrieve values from theOUT parameters with the getXXX() methods.

INOUT : A parameter that provides both input and output values. You bind variables with the setXXX() methods and retrieve values with the getXXX() methods.

```
public class EmployeeSalaryProcedureTest {
    public static void main(String[] args) throws SQLException, ClassNotFoundException {
        Connection con = null;
        try {
            int empId = 2;
            Class.forName("oracle.jdbc.driver.OracleDriver");
            con = DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe", "system", "oracle");
            String sql = "{call getEmpSal (?, ?)}";
            CallableStatement stmt = con.prepareCall(sql);
            stmt.setInt(1, empId);
            stmt.registerOutParameter(2, java.sql.Types.INTEGER);
            stmt.execute();
            int sal = stmt.getInt(2);
            System.out.println("salary:: " + sal);
        } finally {
            if (con != null) {
                con.close();
            }
        }
    }

}
```

Transaction Management:

Set of statements executes under the ACID properties is called transaction.

Atomicity: All the instructions within a transaction will successfully execute, or none of them will execute.

Example: Transfer 20 dollars from Anil account to Suresh account.

Read balance from Anil account

Substract 20 dollers from balance

Write/update balance amount to Anil account

Read balance from Suresh account

Add 20 dollers to balance

Write balance to Suresh account

Consistency: A database is initially in a consistent state, and it should remain consistent after every transaction.

Suppose that the transaction in the previous example fails after Write(Anil balance) and the transaction is not rolled back;

then, the database will be inconsistent as the sum of Anil and Suresh’s money, after the transaction, will not be equal to the amount of money they had before the transaction.

Isolation:

If the multiple transactions are running concurrently, they should not be affected by each other; i.e., the result should be the same as the result obtained if the transactions were running sequentially.

Suppose suresh_bal is initially 100. If a context switch occurs after suresh_bal *= 20, T2 will read the incorrect value of 100 as the updated value will not have been written back to the database. This violates the isolation property as the result is different from the answer that would have been obtained if T1 had finished before T2.

T1:	T2:
Read (suresh_bal)	Read(suresh_bal)
suresh_bal *=0.2	suresh_bal +=20
write(suresh_bal)	write(suresh_bal)

T1 adds 20% interest to Suresh's savings account and T2 adds 20 dollers to Suresh's account.

Durability:

Changes that have been committed to the database should remain even in the case of software , hardware failure and power failure.

For example, if suresh’s account contains $120, this information should not disappear upon hardware or software failure or power failure.

Connection interface provides below methods to manage transaction.

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{
   Connection conn = DriverManager.getConnection(url,username,password);
   conn.setAutoCommit(false);
   Statement stmt = conn.createStatement();

   String SQL = "INSERT INTO EMP  VALUES (200,'Ram', 'Kumar',2000)";
   stmt.executeUpdate(SQL);
   String SQL = "UPDATE EMP SET SAL=1000 WHERE SAL < 5000";
   stmt.executeUpdate(SQL);
   conn.commit();
}catch(SQLException se){
   conn.rollback();
}
```

Savepoints:

The Connection object has two new methods that helps 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.

```
try{
  Connection conn = DriverManager.getConnection(url,username,password);
  conn.setAutoCommit(false);
  Statement stmt = conn.createStatement();
  //set a Savepoint
  Savepoint savepoint1 = conn.setSavepoint("Savepoint1");
  String SQL = "INSERT INTO EMP VALUES (202,'Anil', 'Kumar',25000)";
  stmt.executeUpdate(SQL);
  String SQL = "INSERTED IN EMP VALUES (203,'Rama', 'Rao',30000)";
  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:

Batch Processing allows to group the related SQL statements into a batch and submit them with one call to the database.

addBatch() : method of Statement, PreparedStatement, and CallableStatement is used to add individual statements to the batch.

executeBatch(): returns an array of integers, and each element of the array represents the update count for the respective update statement.

clearBatch() :This method removes all the statements you added with the addBatch() method. However, you cannot selectively choose which statement to remove.

Batching with Statement Object:

Here is a typical sequence of steps to use Batch Processing with Statement Object:

Create a Statement object using either createStatement() methods.

Set auto-commit to false using setAutoCommit().

Add as many as SQL statements you like into batch using addBatch() method on created statement object.

Execute all the SQL statements using executeBatch() method on created statement object.

Finally, commit all the changes using commit() method.

Sample code:

```
public void executeBath(Connection conn) {
try {
Statement stmt = conn.createStatement();
conn.setAutoCommit(false);
String SQL = "INSERT INTO EMP (id, first_name, last_lane, sal)  VALUES(204,'Ramu', 'N', 3000)";
// Add above SQL statement in the batch.
stmt.addBatch(SQL);
String SQL = "INSERT INTO EMP (id, first, last, sal)  VALUES(205,'Sandeep', 'Kumar', 4000)";
// Add above SQL statement in the batch.
stmt.addBatch(SQL);
String SQL = "UPDATE EMP SET sal = 10000 "  WHERE id = 200";
// Add above SQL statement in the batch.
stmt.addBatch(SQL);
//Execute the batch
int[] count = stmt.executeBatch();
//commit the above statements
conn.commit();
} catch(SQLException e) {
  conn.rollback();
}
}
```

ResultSetMetaData Interface:

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.

public int getColumnCount()throws SQLException:It returns the total number of columns in the ResultSet object.

public String getColumnName(int index)throws SQLException: It returns the column name of the specified column index.

public String getColumnTypeName(int index)throws SQLException: It returns the column type name for the specified index.

public String getTableName(int index)throws SQLException:It returns the table name for the specified column index.

Sample Code:

```
PreparedStatement ps=con.prepareStatement("select * from emp");
ResultSet rs=ps.executeQuery();
ResultSetMetaData rsmd=rs.getMetaData();
System.out.println("Total columns: "+rsmd.getColumnCount());
System.out.println("Column Name of 1st column: "+rsmd.getColumnName(1));
System.out.println("Column Type Name of 1st column: "+rsmd.getColumnTypeName(1));
```

DatabaseMetaData interface:

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.

Commonly used methods of DatabaseMetaData interface:

public String getDriverName()throws SQLException: it returns the name of the JDBC driver.

public String getDriverVersion()throws SQLException: it returns the version number of the JDBC driver.

public String getUserName()throws SQLException: it returns the username of the database.

public String getDatabaseProductName()throws SQLException: it returns the product name of the database.

public String getDatabaseProductVersion()throws SQLException: it returns the product version of the database.

public ResultSet getTables(String catalog, String schemaPattern, String tableNamePattern, String[] types)throws SQLException: it returns the description of the tables of the specified catalog. The table type can be TABLE, VIEW, ALIAS, SYSTEM TABLE, SYNONYM etc.

Sample code:

```
try{
Class.forName("oracle.jdbc.driver.OracleDriver");
Connection con=DriverManager.getConnection(
"jdbc:oracle:thin:@localhost:1521:xe","root","root");
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 Version: "+dbmd.getDatabaseProductVersion());

}catch(Exception e){ System.out.println(e);}
finally {
  con.close();
}
```