JDBC

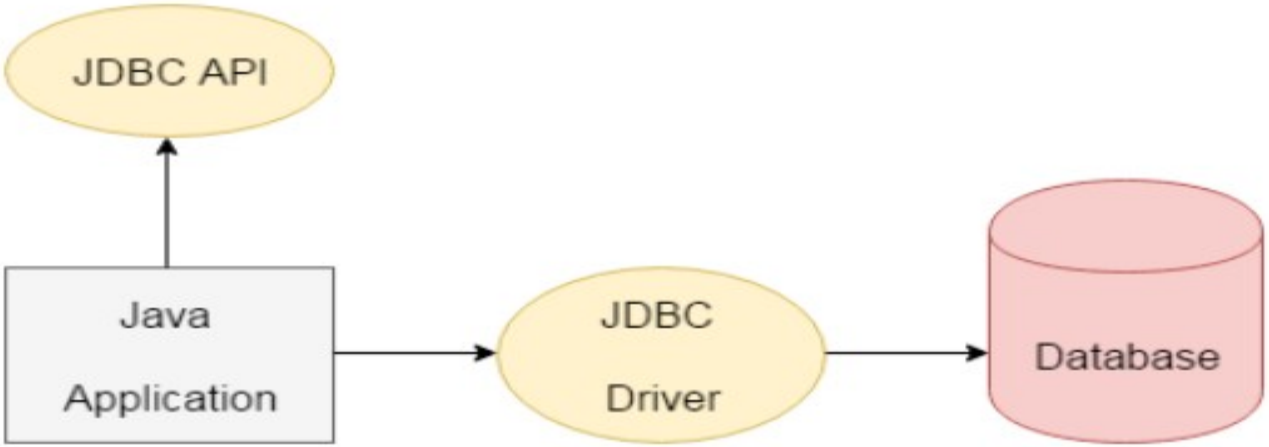
Introduction

JDBC stands for Java Database Connectivity. JDBC is a Java API to connect and execute the query with the database. It is a part of JavaSE (Java Standard Edition). JDBC API uses JDBC drivers to connect with the database.

JDBC stands for Java Database Connectivity. JDBC is a Java API to connect and execute the query with the database. It is a part of JavaSE (Java Standard Edition). JDBC API uses JDBC drivers to connect with the database.

We can use JDBC API to access tabular data stored in any relational database. By the help of JDBC API, we can save, update, delete and fetch data from the database.

Introduction



JDBC Architecture

The JDBC API supports both two-tier and three-tier processing models for database access but in general, 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.

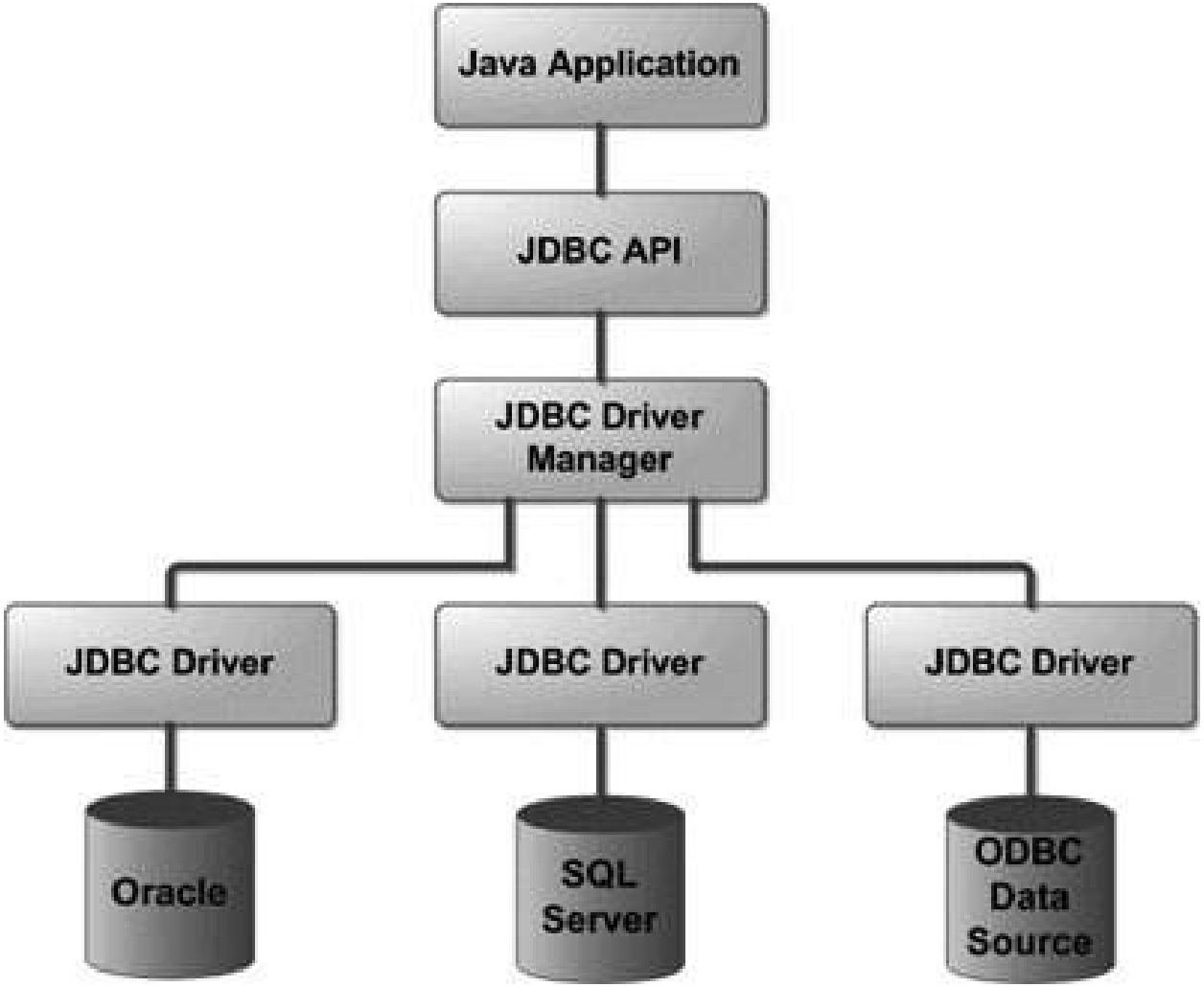
The JDBC API uses a driver manager and database-specific drivers to provide transparent connectivity to heterogeneous databases.

JDBC Architecture

The JDBC driver manager ensures that the correct driver is used to access each data source. The driver manager is capable of supporting multiple concurrent drivers connected to multiple heterogeneous databases.

Following is the architectural diagram, which shows the location of the driver manager with respect to the JDBC drivers and the Java application –

JDBC Architecture



JDBC Architecture

Common JDBC Components

The JDBC API provides the following interfaces and classes –

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.

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.

JDBC Architecture

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.

Statement − You use objects created from this interface to submit the SQL statements to the database.

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.

SQLException − This class handles any errors that occur in a database application.

Introduction

Why Should We Use JDBC

Before JDBC, ODBC API was the database API to connect and execute the query with the database. But, ODBC API uses ODBC driver which is written in C language (i.e. platform dependent and unsecured). That is why Java has defined its own API (JDBC API) that uses JDBC drivers (written in Java language).

We can use JDBC API to handle database using Java program and can perform the following activities:

- Connect to the database
- Execute queries and update statements to the database
- Retrieve the result received from the database.

Types of JDBC Drivers

There are 4 types of JDBC drivers:

JDBC-ODBC bridge driver
Native-API driver (partially java driver)
Network Protocol driver (fully java driver)
Thin driver (fully java driver)

Types of JDBC Drivers

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.

The JDBC-ODBC Bridge driver is recommended only for experimental use and is typically used for development and testing purposes only.

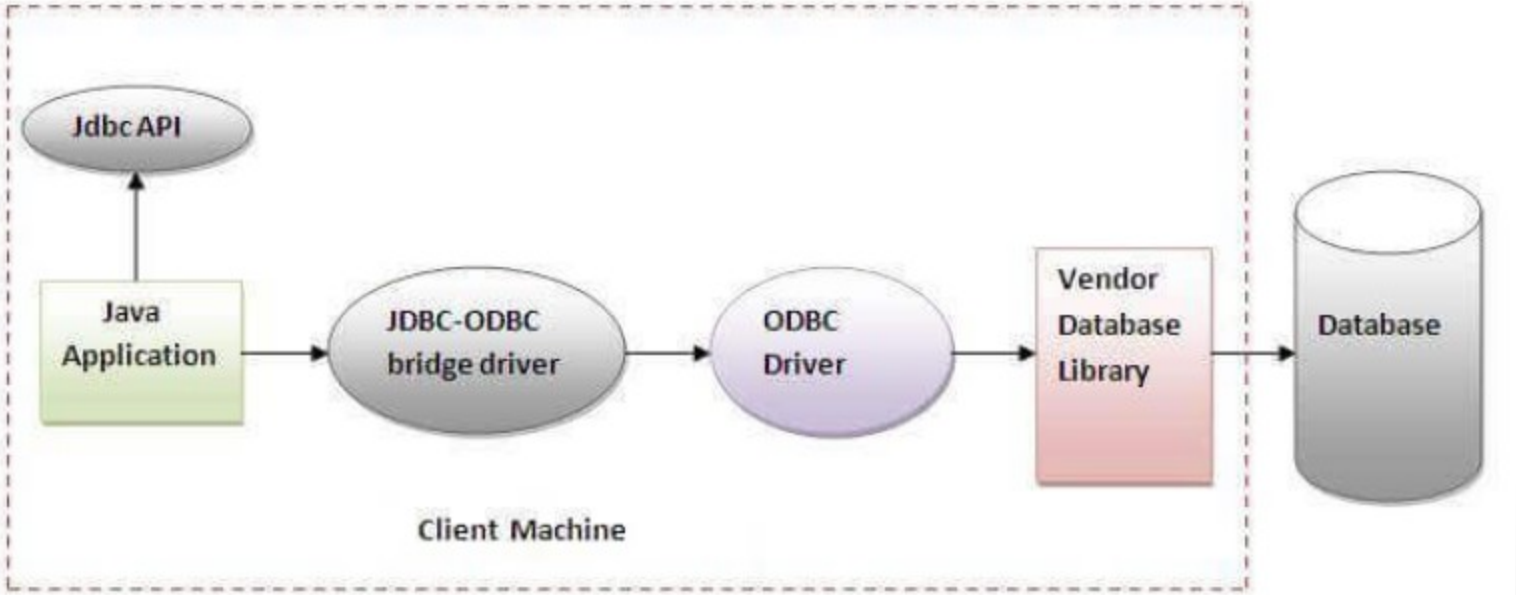


Figure- JDBC-ODBC Bridge Driver

Con't

Advantage

- The JDBC-ODBC Bridge allows access to almost any database, since the database's ODBC drivers are already available.

Disadvantages

- Since the Bridge driver is not written fully in Java, Type 1 drivers are not portable.
- A performance issue is seen as a JDBC call goes through the bridge to the ODBC driver, then to the database, and this applies even in the reverse process.
- They are the slowest of all driver types.
- The client system requires the ODBC Installation to use the driver.
- Not good for the Web.
-

Types of JDBC Drivers

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.

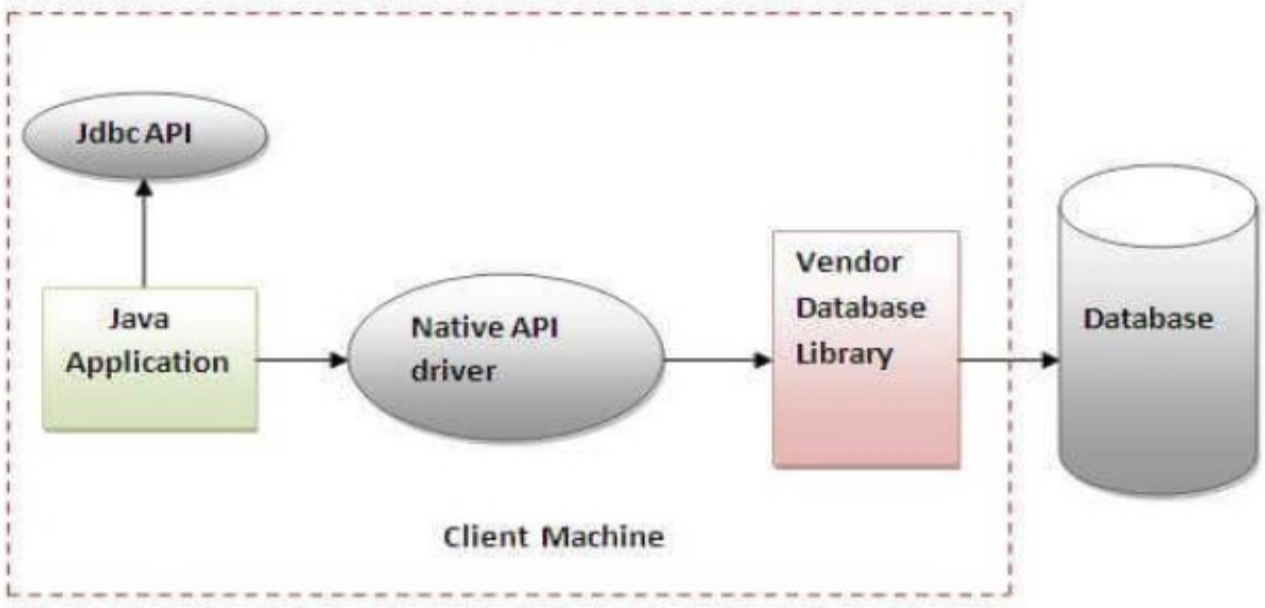


Figure- Native API Driver

Con't

Advantage

- They typically offer better performance than the JDBC-ODBC Bridge as the layers of communication (tiers) are less than that of Type1 and also it uses Native api which is Database specific.

Disadvantage

- Native API must be installed in the Client System and hence type 2 drivers cannot be used for the Internet.
- Like Type 1 drivers, it's not written in Java Language which forms a portability issue.
- If we change the Database we have to change the native api as it is specific to a database.
- Mostly obsolete now.
- Usually not thread safe.

Types of JDBC Drivers

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.

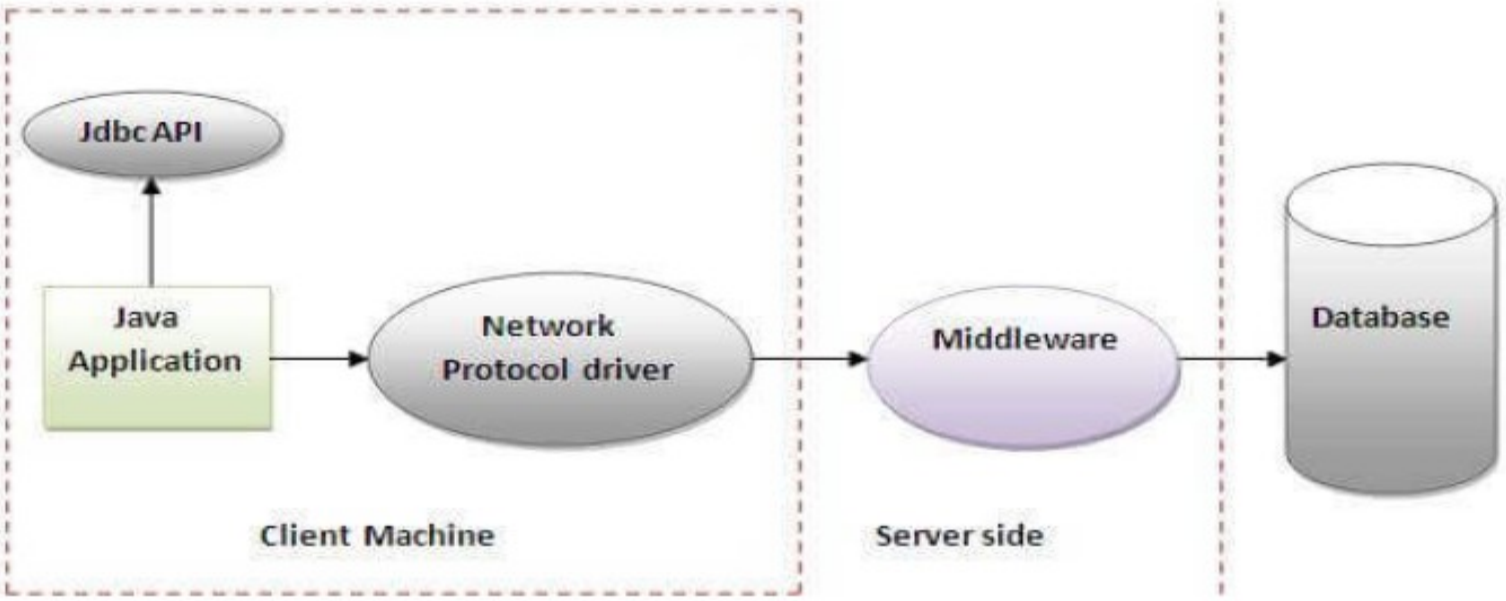


Figure- Network Protocol Driver

Con't

Advantage

- This driver is server-based, so there is no need for any vendor database library to be present on client machines.
- This driver is fully written in Java and hence Portable. It is suitable for the web.
- There are many opportunities to optimize portability, performance, and scalability.
- The net protocol can be designed to make the client JDBC driver very small and fast to load.
- This driver is very flexible allows access to multiple databases using one driver.
- They are the most efficient amongst all driver types.

Disadvantage

- It requires another server application to install and maintain.

Types of JDBC Drivers

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.

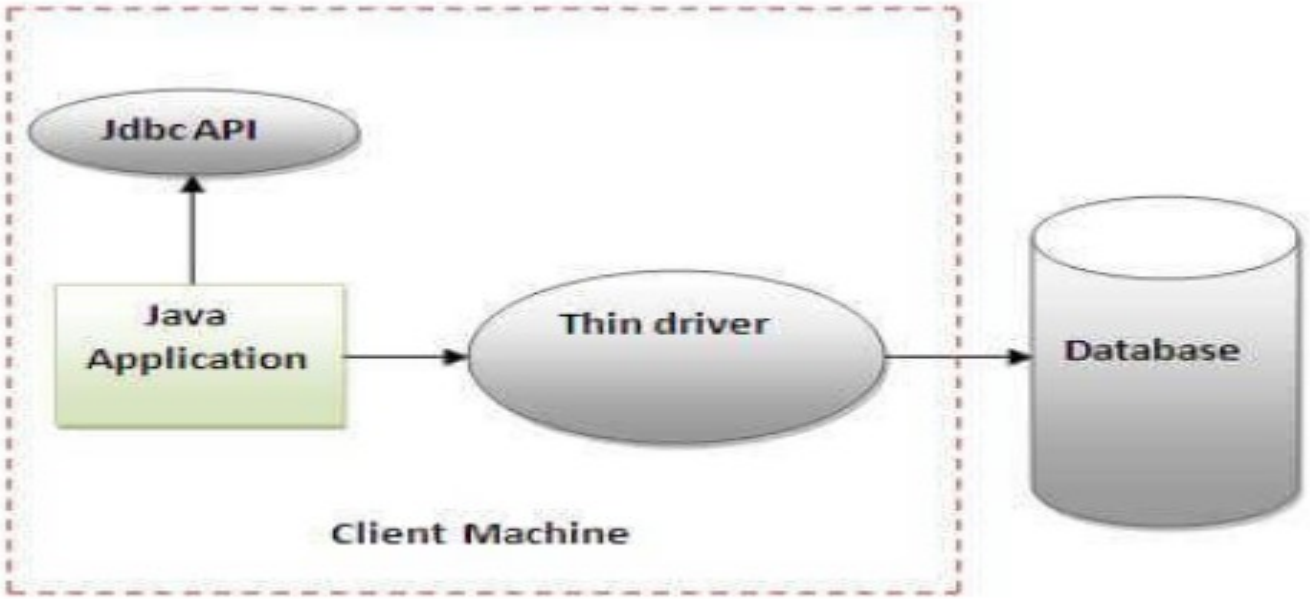


Figure- Thin Driver

Con't

Advantage

- They are completely written in Java to achieve platform independence.
- It is most suitable for the web.
- Number of translation layers is very less
- i.e. type 4 JDBC drivers don't have to translate database requests to ODBC or a native connectivity interface or to pass the request on to another server, performance is typically quite good.
- You don't need to install special software on the client or server. Further, these drivers can be downloaded dynamically.

Disadvantage

- With type 4 drivers, the user needs a different driver for each database.

Which Driver should be used?

- 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.
- The type 1 is typically used for development and testing purposes only.

Java Database Connectivity

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

Register the Driver class
Create connection
Create statement
Execute queries
Close connection

Java Database Connectivity

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.

Syntax of forName() method

public static void forName(String className)throws ClassNotFoundException

Example to register the OracleDriver class

Here, Java program is loading oracle driver to esteblish database connection.

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

Java Database Connectivity

2) Create the connection object

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

Syntax of getConnection() method

```
public static Connection getConnection(String url,String name,String password)
  throws SQLException
```

Example to establish connection with the Oracle database

```
Connection con=DriverManager.getConnection(
"jdbc:oracle:thin:@localhost:1521:xe","system","password");
```

Java Database Connectivity

3) Create the Statement object

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

Syntax of createStatement() method

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

Example to create the statement object

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

Java Database Connectivity

Three kinds of Statements:

Statement - Execute simple sql queries without parameters.

Useful when you are using static SQL statements at runtime.

Statement createStatement ()

Statement statement = dbConnection.createStatement ();

Prepared Statement - Execute precompiled sql queries with or without parameters.

It can accept input parameters at runtime.

PreparedStatement prepareStatement (String sql)

PreparedStatement objects are precompiled SQL statements.

Callable Statement - Execute a call to a database stored procedure.

It can accept runtime input parameters.

CallableStatement prepareCall (String sql)

CallableStatement objects are SQL stored procedure call statements.

Java Database Connectivity

4) Execute the query

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.

Syntax of executeQuery() method

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

Example to execute query

```
ResultSet rs=stmt.executeQuery("select * from emp");

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

Java Database Connectivity

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.

Syntax of close() method

```
public void close()throws SQLException
```

Example to close connection

```
con.close();
```

Java Database Connectivity

Statement interface

The Statement interface provides methods to execute queries with the database.
Commonly used methods of Statement interface:

The important methods of Statement interface are as follows:
1) public ResultSet executeQuery(String sql): is used to execute SELECT query. It returns the object of ResultSet.
2) public int executeUpdate(String sql): is used to execute specified query, it may be create, drop, insert, update, delete etc.
3) public boolean execute(String sql): is used to execute queries that may return multiple results.
4) public int[] executeBatch(): is used to execute batch of commands.

```
import java.sql.*;
class FetchRecord{
public static void main(String args[])throws Exception{
Class.forName("oracle.jdbc.driver.OracleDriver");
Connection con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe","system","oracle");
Statement stmt=con.createStatement();

//stmt.executeUpdate("insert into emp765 values(33,'Irfan',50000)");
//int result=stmt.executeUpdate("update emp765 set name='Vimal',salary=10000 where id=33");
int result=stmt.executeUpdate("delete from emp765 where id=33");
System.out.println(result+" records affected");
con.close();
}}
```

Java Database Connectivity

PreparedStatement interface

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

Let's see the example of parameterized query:

```
String sql="insert into emp values(?,?,?)";
```

As you can see, we are passing parameter (?) for the values. Its value will be set by calling the setter methods of PreparedStatement.

Java Database Connectivity

How to get the instance of PreparedStatement?

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

public PreparedStatement prepareStatement(String query)throws SQLException{}

Java Database Connectivity

Methods of PreparedStatement interface

Method	Description
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()	executes the query. It is used for create, drop, insert, update, delete etc.
public ResultSet executeQuery()	executes the select query. It returns an instance of ResultSet.

```
import java.sql.*;
class InsertPrepared{
public static void main(String args[]){
try{
Class.forName("oracle.jdbc.driver.OracleDriver");

Connection con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe","system","oracle");

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

int i=stmt.executeUpdate();
System.out.println(i+" records inserted");

con.close();

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

}
}
```

```
PreparedStatement stmt=con.prepareStatement("update emp set name=? where id=?");
stmt.setString(1,"vjv");//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 stmt=con.prepareStatement("delete from emp where id=?");
stmt.setInt(1,101);

int i=stmt.executeUpdate();
System.out.println(i+" records deleted");

PreparedStatement stmt=con.prepareStatement("select * from emp");
ResultSet rs=stmt.executeQuery();
while(rs.next()){
System.out.println(rs.getInt(1)+" "+rs.getString(2));
}
```

Java Database Connectivity

Java CallableStatement Interface

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

We can have business logic on the database by the use of stored procedures and functions that will make the performance better because these are precompiled.

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.

Java Database Connectivity

How to get the instance of CallableStatement?

The prepareCall() method of Connection interface returns the instance of CallableStatement. Syntax is given below:

```
public CallableStatement prepareCall("{ call procedurename(?,?...?)}");
```

Java Database Connectivity

example to call the stored procedure using JDBC

```
create or replace procedure "INSERTRECORD"
(id IN NUMBER,
name IN VARCHAR2)
is
begin
insert into user1 values(id,name);
end;
/
```

The table structure is given below:

```
create table user1(id number(10), name varchar2(200));
```

Java Database Connectivity

```
import java.sql.*;
public class Proc {
public static void main(String[] args) throws Exception{

Class.forName("oracle.jdbc.driver.OracleDriver");
Connection con=DriverManager.getConnection(
"jdbc:oracle:thin:@localhost:1521:xe","system","oracle");

CallableStatement stmt=con.prepareCall("{call INSERTRECORD(?,?)}");
stmt.setInt(1,1011);
stmt.setString(2,"Amit");
stmt.execute();

System.out.println("success");
}
}
```

Java Database Connectivity

Batch Processing in JDBC

Instead of executing a single query, we can execute a batch (group) of queries. It makes the performance fast. It is because when one sends multiple statements of SQL at once to the database, the communication overhead is reduced significantly.

The java.sql.Statement and java.sql.PreparedStatement interfaces provide methods for batch processing.

Java Database Connectivity

Batch processing in JDBC. It follows following steps:

Load the driver class
Create Connection
Create Statement
Add query in the batch
Execute Batch
Close Connection

```
import java.sql.*;
class FetchRecords1{
public static void main(String args[])throws Exception{
Class.forName("oracle.jdbc.driver.OracleDriver");
Connection con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe","system","oracle");
con.setAutoCommit(false);

Statement stmt=con.createStatement();
stmt.addBatch("insert into user420 values(190,'abhi',40000)");
stmt.addBatch("insert into user420 values(191,'umesh',50000)");

stmt.executeBatch();//executing the batch

con.commit();
con.close();
}}
```