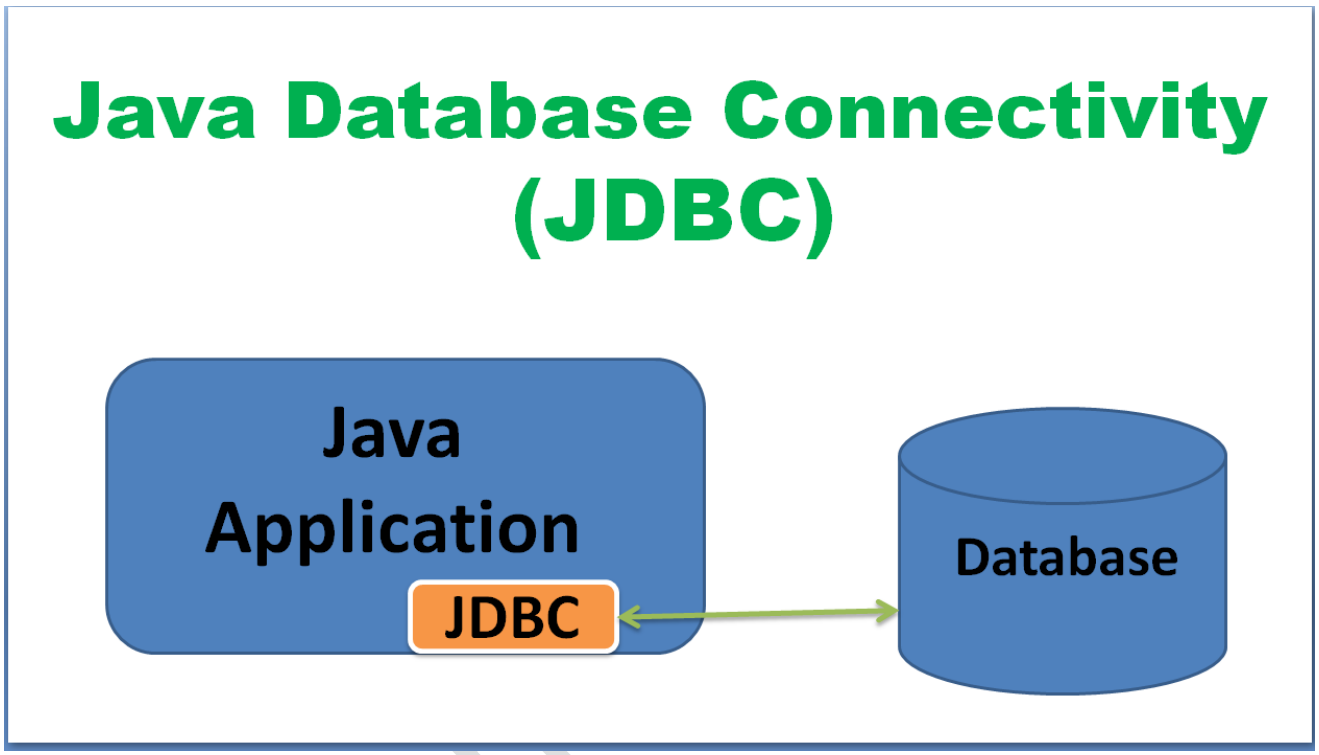
JDBC



Java Database Connectivity (JDBC) is an application programming interface (API) for the programming language Java, which defines how a client may access a database.

Java Database Connectivity (JDBC) is an application program interface (API) specification for connecting programs written in Java to the data in popular databases. The application program interface lets you encode access request statements in Structured Query Language (SQL) that are then passed to the program that manages the database. It returns the results through a similar interface.

Architecture of JDBC

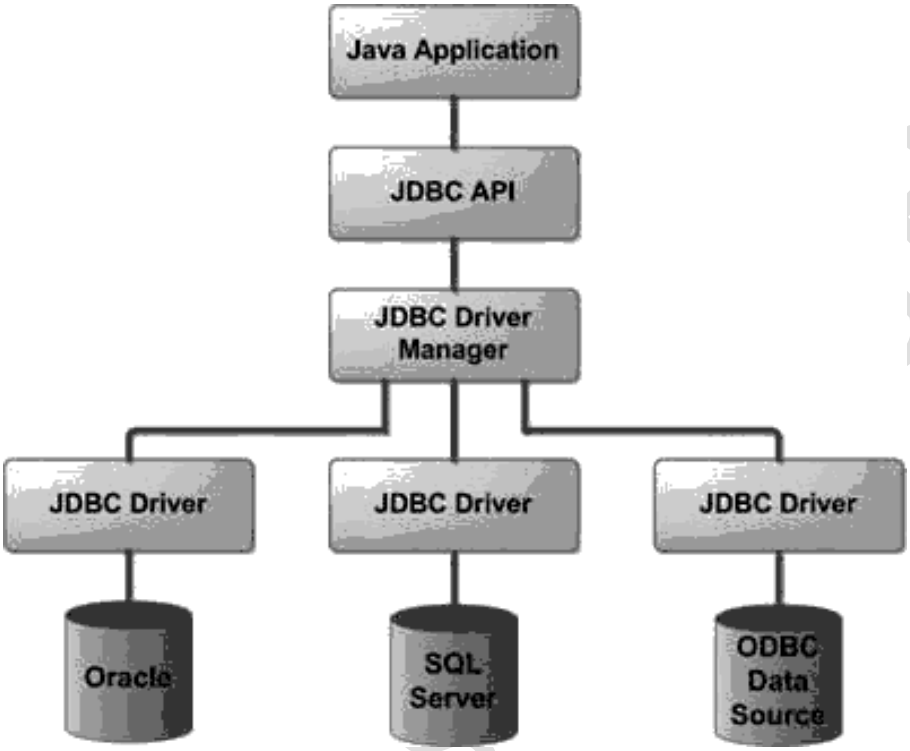
JDBC is designed on the basis of the agreement between Sun and Database vendors. As per this understanding Sun provide Java API for SQL access along with a driver manager and database vendors could provide their own drivers to plug in to the driver manager.

By making this design two API were created:

i) **JDBC API** for application programmer.

ii) **JDBC Driver API** for database vendors and tool providers.

Note:JDBC also provide support for ODBC as JDBC-to-ODBC bridge in older version.



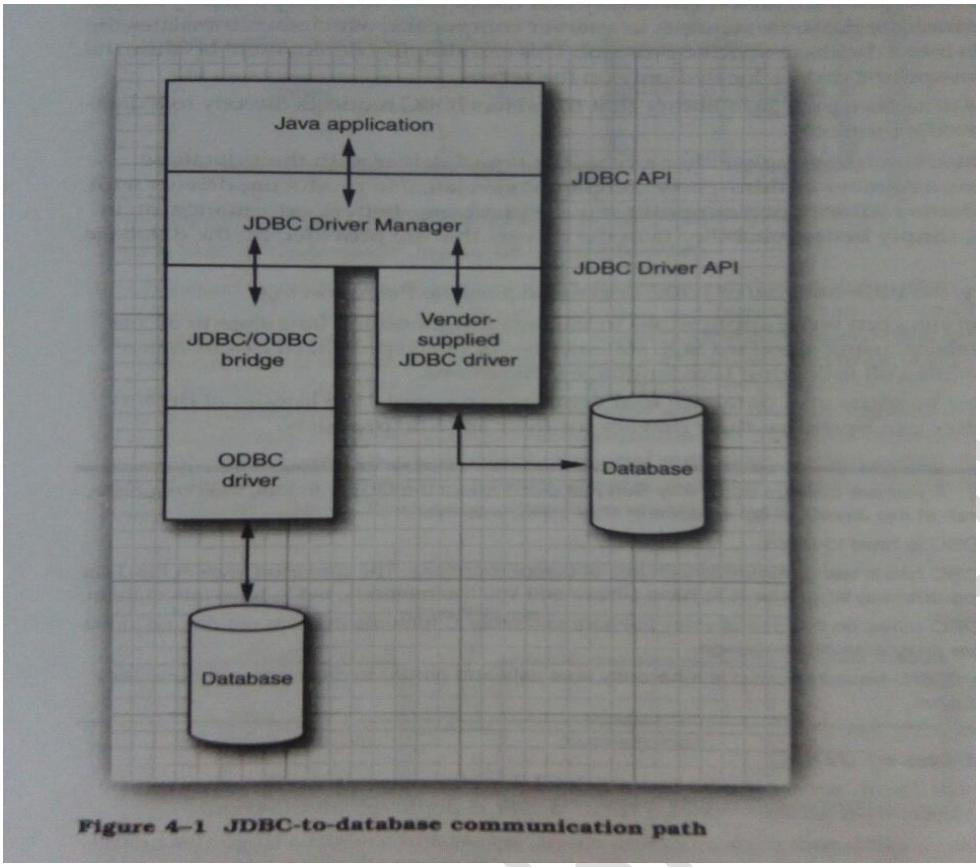


Figure 4–1 JDBC-to-database communication path

The latest version, JDBC 4.3, is specified by a maintenance release 3 of JSR 221 and is included in Java SE 9.

JDBC Driver

A JDBC driver is a set of Java classes that implement the JDBC interfaces, targeting a specific database.

The JDBC interface comes with standard Java, but the implementation of these interfaces is specific to the database you need to connect to. Such an implementation is called a JDBC driver.

JAR FILE

Database driver can be downloaded from the official website of Oracle, which are available in jar files.

Type of JDBC Driver

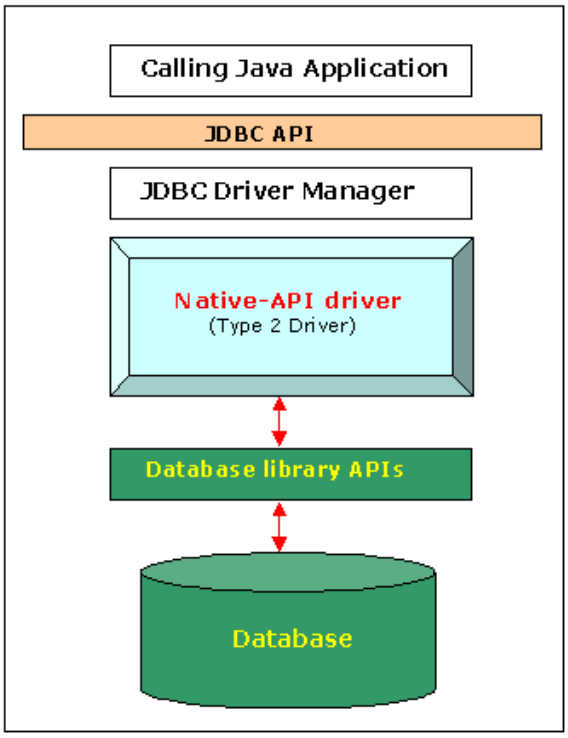
There are 4 different types of JDBC drivers:

Type 1 Driver:

A type 1 JDBC driver consists of a Java part that translates the JDBC interface calls to ODBC calls. A JDBC-to-ODBC bridge then calls the ODBC driver of the given database. Type 1 drivers are (were) mostly intended to be used in the beginning, when there were no type 4 drivers (all Java drivers). Any database for which an ODBC driver is installed can be accessed, and data can be retrieved.

Type 2 Driver (Native API Driver)

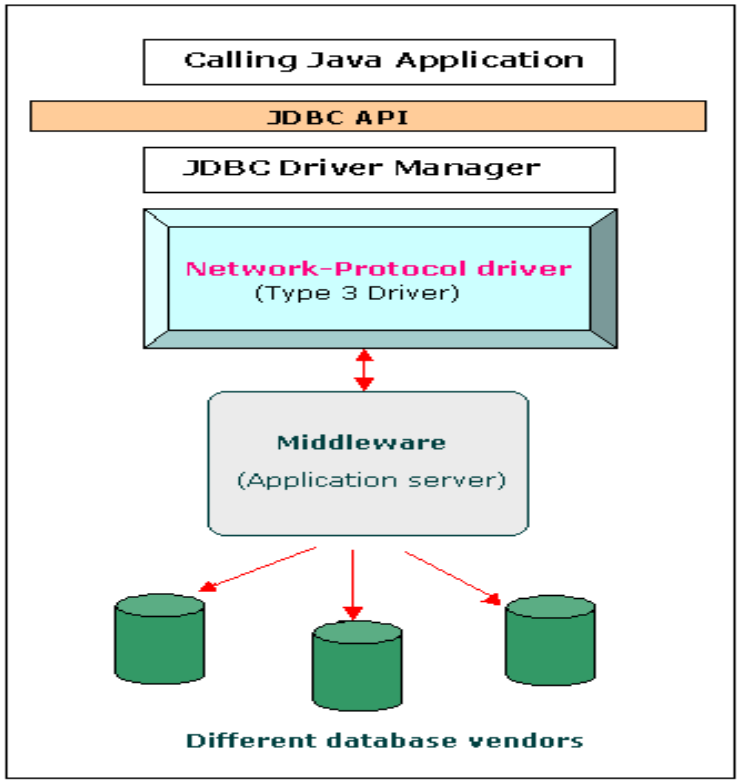
The JDBC type 2 driver, also known as the **Native-API driver**, is a database driver implementation that uses the client-side libraries of the database. The driver converts JDBC method calls into native calls of the database API.



Type 3 driver – Network-Protocol driver (middleware driver)

The JDBC type 3 driver, also known as the Pure Java driver for database middleware, is a database driver implementation which makes use of a middle tier between the calling program and the database. The middle-tier (application server) converts JDBC calls directly or indirectly into a vendor-specific database protocol.

The same client-side JDBC driver may be used for multiple databases. It depends on the number of databases the middleware has been configured to support. The type 3 driver is platform-independent as the platform-related differences are taken care of by the middleware. Also, making use of the middleware provides additional advantages of security and firewall access.



Type 4 driver – Database-Protocol driver (Pure Java driver)

The JDBC type 4 driver, also known as the Direct to Database Pure Java Driver, is a database driver implementation that converts JDBC calls directly into a vendor-specific database protocol.

Written completely in Java, type 4 drivers are thus platform independent. They install inside the Java Virtual Machine of the client.

Type 4 driver differ from type 3 driver is that the protocol conversion logic resides not at the client, but in the middle-tier. Type 4 drivers and type 3 driver are written entirely in Java.

Requirement for JDBC

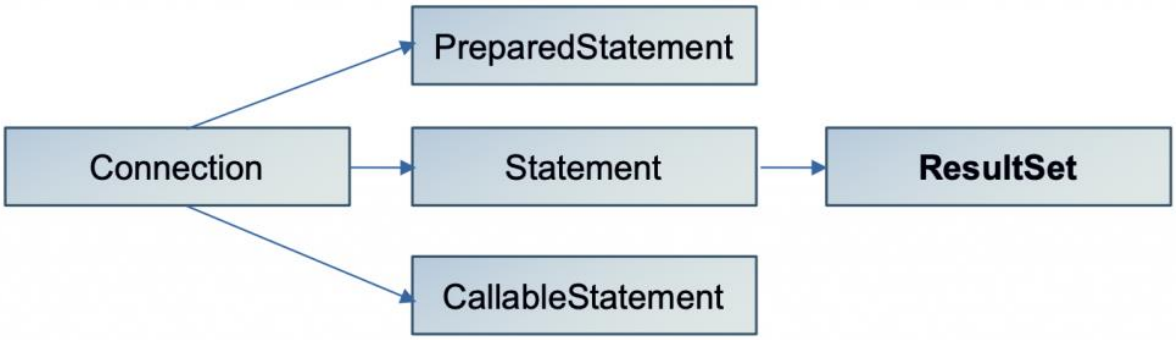
- Database Software
  - Database
  - Tables
  - SQL Query
- Database Driver
- IP Address
- Port No

Steps to access the database through JDBC

Followings are the steps involve connecting and accessing the database from Java Application through JDBC

- Import the package java.sql.*;

- Register the driver
- Establish the connection
- Create the statement object
- Create the result set
- Use the result set
- Close the connection.



1. Registration of Driver:

A database driver is a computer program that implements a protocol (JDBC) for a database connection. The driver works like an adaptor which connects a generic interface to a specific database vendor implementation.

Database driver can be downloaded from the official website of Oracle, which are available in jar files.

You must register the driver in your program before you use it. Registering the driver is the process by which the Data Base vendor (e.g. Oracle) driver's class file is loaded into the memory, so that it can be utilized as an implementation of the JDBC interfaces.

Method to register driver :

- Class.forName Method

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

Note: java.lang.Class is the class instance represent classes and interfaces in a running Java application.

Note : Applications no longer need to explicitly load JDBC drivers using Class.forName(). Existing programs which currently load JDBC drivers using Class.forName() will continue to work without modification. When the method getConnection is called, the DriverManager will attempt to locate a suitable driver from amongst those loaded at initialization and those loaded explicitly using the same classloader as the current applet or application. Starting with the Java 2 SDK, Standard Edition, version 1.3, a logging stream can be set only if the proper permission has been granted.

Note: java.lang.Class is the class instance represent classes and interfaces in a running Java application.

Driver can also be registerd through the DriverManager.RegisterDriver() method:

It is use for non jdk compliant JVM, for example supplied by MicroSoft.

```
try{
DriverManager.registerDriver(new oracle.jdbc.OracleDriver());
```

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

2. Establishing the Connection:

Database URL

After you've loaded the driver, you can establish a connection using the **DriverManager.getConnection()** method.

There are 3 overloaded DriverManager.getConnection() methods :

- getConnection(String url)
- getConnection(String url, Properties prop)
- getConnection(String url, String user, String password)

Here each form requires a database **URL**. A database URL is an address that points to database.

Formulating a database URL is where most of the problems associated with establishing a connection occurs.

Following table lists down the popular JDBC driver names and database URL.

RDBMS	JDBC driver name	URL format
MySQL	com.mysql.jdbc.Driver com.mysql.cj.jdbc.Driver (For java version higher than 8)	**jdbc:mysql://**hostname:Port Number/ databaseName

ORACLE	oracle.jdbc.driver.OracleDriver	**jdbc:oracle:thin:**@hostname:portNumber:databaseName
PostGreSQL	**org.postgresql.Driver**	**jdbc:postgresql**:// hostname : server-port / database-name
MS-SQL Server	com.microsoft.jdbc.sqlserver.SQLServerDriver	**jdbc:sqlserver**://{HOST}:{PORT};databaseName={DB}
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

All the highlighted part in URL format is static.

Remaining portion need to change.

Default Port No.

Oracle : 1521

MySql : 3306

PostGreSQL :5432

MS-SQLServer : 1433

Create Connection Object

To create a connection **Database URL with a username and password** are required.

Assuming you are using Oracle's **thin** driver, you'll specify a host:port:databaseName value for the database portion of the URL.

If you have a host at TCP/IP address 192.0.0.1 with a host name of SBIT and Oracle listener is configured to listen on port 1521, and database name is EMP, then complete database URL would be –

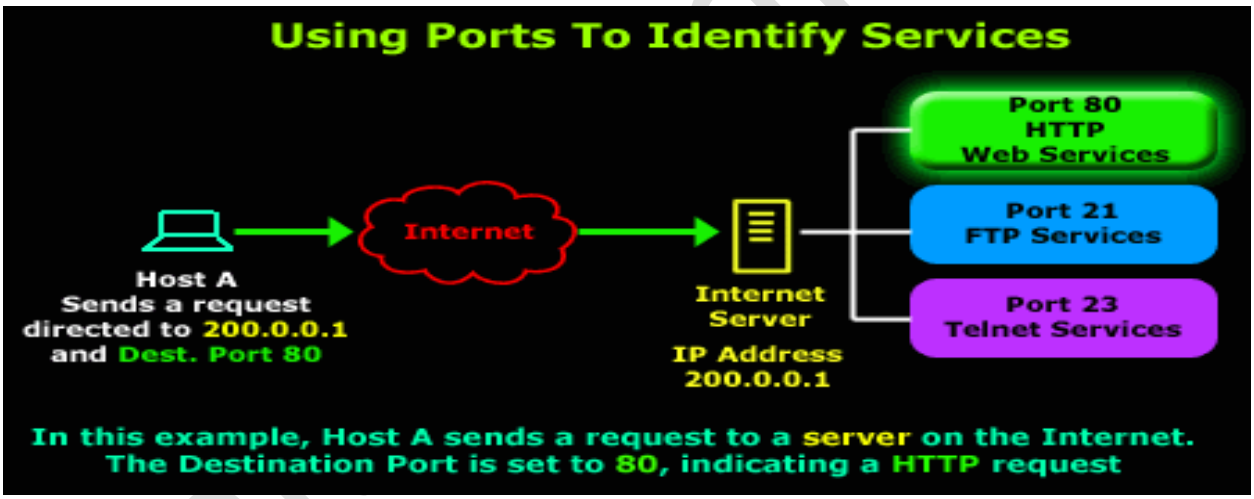
jdbc:oracle:thin:@GVM:1521:EMP

Now you have to call getConnection() method with appropriate username and password to get a Connection object as follows –

String URL = "jdbc:oracle:thin:@GVM:1521:EMP";
String USER = "username";
String PASS = "password"

Connection conn = DriverManager.getConnection(URL, USER, PASS);

What is Port No:
Port numbers are part of the addressing information used to identify the senders and receivers of messages. Port numbers allow different applications on the same computer to share network resources simultaneously. These port numbers work like telephone extensions. Just as a business telephone switchboard can use the main phone number and assign each employee an extension number.



JDBC THIN : *It is pure java driver and does not require any Oracle software installed on client computer.*

Hostname: *A hostname is a name that is assigned to a device connected to a computer network and that is used to identify the device in various forms of electronic communication*

- **Create the statement object**

Statement object used to execute SQL query. Before execution of SQL query. There is need to create a Statement Object.

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

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

- **Result Set**

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. The term "result set" refers to the row and column data contained in a ResultSet object.

ResultSet rs;

To get the result in rs use one of the following **Method of Statement Object**:

- **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 are expected to get a number of rows affected - for example, an INSERT, UPDATE, or DELETE statement.

For Example:

Statement st = conn.createStatement();

Int reslt = st.executeUpdate(“Update Emp set salary = salary+10/100*salary where city = ‘delhi’ ”);

- **ResultSet executeQuery (String SQL)**: Returns a ResultSet object. This method is used when you expect to get a result set, as you would with a SELECT statement.

For example:

ResultSet Rs = st.executeQuery (Select * from emp);

- **execute()** --- If you don’t know which method to be used for executing SQL statements, this method can be used. This will return a boolean. TRUE indicates the result is a ResultSet and FALSE indicates it has the int value which denotes number of rows affected by the query.

boolean rslt;

rslt = st.execute(“Select name from emp where salary>100000”);

In the above example if rslt value is true it means ResultSet is created and if it is false it means no ResultSet only it will tells us the no of row affected.

- **Closing the connection**

To close the connection in JDBC

conn.close();

Working on Result Set:

Function of result set with complete signature:

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

1	public void beforeFirst() throws SQLException Moves the cursor just before the first row.
2	public void afterLast() throws SQLException Moves the cursor just after the last row.
3	public boolean first() throws SQLException Moves the cursor to the first row.
4	public void last() throws SQLException Moves the cursor to the last row.
5	public boolean absolute(int row) throws SQLException Moves the cursor to the specified row.
6	public boolean relative(int row) throws SQLException Moves the cursor the given number of rows forward or backward, from where it is currently pointing.
7	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.
8	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.
9	public intgetRow() throws SQLException

	Returns the row number that the cursor is pointing to.
10	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.
11	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 does nothing

By using column name:

while(rs.next())

{
System.out.println (rs.getString("item")+rs.getInt("age"));
}

Here "item" and "age" are the name of column.

Example Database Connectivity

```
package databasepackage;
import java.sql.*;
public class DataConnectionTest {

	public Connection cn;
	public Statement stmt;
	ResultSet rs;
	public void connect()
	{
		try {
		Class.forName("com.mysql.cj.jdbc.Driver");
		}catch(ClassNotFoundException cnfe)
		{
			System.out.println(cnfe);
		}

		try {
```

```
        cn  =  DriverManager.getConnection
        ("jdbc:mysql://localhost:3306/mca","root","Abc@123");
        ------------------------------------------
              stmt = cn.createStatement();

              rs=stmt.executeQuery("Select * from login");

              while(rs.next())
              {
                    System.out.println(rs.getString(1));
                    System.out.println(rs.getString(2));
              }
              }
              catch(SQLException sqe)
              {
                    System.out.println(sqe);
              }
        }

        public static void main(String arg[])
        {
              DataConnectionTest dc = new DataConnectionTest();
              dc.connect();
        }
}
```

ResultSetMetaData

ResultSetMetaData is also one of the interface of java.sql package. This interface provides overview about a *ResultSet* object like number of columns, column name, data type of a column etc. We need this info about a *ResultSet* object before processing the actual data of a *ResultSet*.

Instance of ResultSetMetaData can be instantiated by the method of ResultSet i.e.

getMetaData()

for example :

ResultSet rs;

ResultSetMetaData rsmd = rs.getMetaData();

Methods of ResultSetMetaDatainterface :

Method Name	Description
Int getColumnCount() throws SQLException	Returns the number of columns in a ResultSet.
String getColumnName(int column) throws SQLException	Returns the column name.
String getColumnTypeName(int column) throws SQLException	Returns the database specific datatype of the column.
String getTableName(int column) throws SQLException	Returns the column's table name.
String getSchemaName(int column) throws SQLException	Returns the name of the schema of the column's table.

Complete Example of ResultSetMetaData

```
Import java.sql.*;

public class ResultSetMetaDataExample
{
   static
   {
      //Registering The Driver Class

      try
      {
         Class.forName("oracle.jdbc.driver.OracleDriver");
      }
      catch (ClassNotFoundException e)
      {
         System.out.println("Unable To Load The Driver class");
      }
   }

   public static void main(String[] args)
   {
      Connection con = null;
```

```
Statement stmt = null;

ResultSetrs = null;

try
{
   //Database Credentials

   String URL = "jdbc:oracle:thin:@localhost:1521:XE";

   String username = "username";

   String password = "password";

   //Creating The Connection Object

   con = DriverManager.getConnection(URL, username, password);

   //Creating The Statement Object

   stmt = con.createStatement();

   //Constructing The SQL Query

   String sql = "SELECT * FROM EMPLOYEE";

   //Executing The Query

   rs = stmt.executeQuery(sql);

   //getting ResultSetMetaData object

   ResultSetMetaData rsmd = rs.getMetaData();

   //getting number of columns in 'rs'

   Int colCount = rsmd.getColumnCount();

   System.out.println("Number Of Columns : "+colCount);

   System.out.println("column Details :");

   for (int i = 1; i<= colCount; i++)
   {
      //getting column name of index 'i'
```

```
            String colName = rsmd.getColumnName(i);

            //getting column's data type of index 'i'

            String colType = rsmd.getColumnTypeName(i);

            System.out.println(colName+" is of type "+colType);
        }
    }
    catch (SQLException e)
    {
        e.printStackTrace();
    }
    finally
    {
        //Closing The DB Resources

        //Closing the ResultSet object

        try
        {
            if(rs!=null)
            {
                rs.close();
                rs=null;
            }
        }
        catch (SQLException e)
        {
            e.printStackTrace();
        }

        //Closing the Statement object
        try
        {
            if(stmt!=null)
            {
                stmt.close();
                stmt=null;
            }
        }
        catch (SQLException e)
        {
            e.printStackTrace();
        }
```

```
            //Closing the Connection object

            try
            {
               if(con!=null)
               {
                  con.close();
                  con=null;
               }
            }
            catch (SQLException e)
            {
               e.printStackTrace();
            }
         }
      }
}
```

OUTPUT :

```
Number Of Columns : 4
column Details :
ID is of type NUMBER
FIRST_NAME is of type VARCHAR2
LAST_NAME is of type VARCHAR2
DISIGNATION is of type VARCHAR2
```

Types of Statement:

1. Statement
2. PreparedStatement
3. CallableStatement

Difference between Statement Object, PreparedStatement Object and CallableStatement Object in Java:

Statement	PreparedStatement	CallableStatement
It is used to execute normal	It is used to execute	It is used to call the stored

SQL queries.	parameterized or dynamic SQL queries.	procedures.
It is preferred when a particular SQL query is to be executed only once.	It is preferred when a particular query is to be executed multiple times.	It is preferred when the stored procedures are to be executed.
You cannot pass the parameters to SQL query using this interface.	You can pass the parameters to SQL query at run time using this interface.	You can pass 3 types of parameters using this interface. They are – IN, OUT and IN OUT.
This interface is mainly used for DDL statements like CREATE, ALTER, DROP etc.	It is used for any kind of SQL queries which are to be executed multiple times.	It is used to execute stored procedures and functions.
The performance of this interface is very low.	The performance of this interface is better than the Statement interface (when used for multiple execution of same query).	The performance of this interface is high.

PreparedStatement:

- The PreparedStatement interface is a **sub interface of Statement**. It is used to execute parameterized query.
- PreparedStatementis use to improve the performance of the application because query is compiled only once.

Method to prepare the object :

prepareStatement() is the method of Connection interface, it return the object of PreparedStatement.

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

PreparedStatement pstmt=con.prepareStatement("insert into Emp values(?,?)");
```

Here '?' is the parameter where relevant values can be passed using method of PreparedStatement interface.

The important methods of PreparedStatement interface are below:

Method	Description
public void setInt(intparamIndex, int value)	sets the integer value to the given parameter index.
public void setString(intparamIndex, String value)	sets the String value to the given parameter index.
public void setFloat(intparamIndex, float value)	sets the float value to the given parameter index.
public void setDouble(intparamIndex, double value)	sets the double value to the given parameter index.
public intexecuteUpdate()	executes the query. It is used for create, drop, insert, update, delete etc.
public ResultSetexecuteQuery()	executes the select query. It returns an instance of ResultSet.

For example:

pstmt.setInt(1,100); // here 1 is the first parameter and 100 is the value of that parameter.

Int i = pstmt.executeUpdate(); // this will execute the statement with the passed parameters.

Example of PreparedStatement to insert records until user press n

```
import java.sql.*;
import java.io.*;
class RS{
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");

PreparedStatement ps=con.prepareStatement("insert into emp values(?,?,?)");
BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
do {
System.out.println("enter id:");
int id=Integer.parseInt(br.readLine());
System.out.println("enter name:");
String name=br.readLine();
System.out.println("enter salary:");
float salary=Float.parseFloat(br.readLine());

ps.setInt(1,id);
```

```
ps.setString(2,name);
ps.setFloat(3,salary);
int i=ps.executeUpdate();
System.out.println(i+" records affected");

System.out.println("Do you want to continue: y/n");
String s=br.readLine();
if(s.startsWith("n")){
break;
}
}while(true);

con.close();
}}
```

CallableStatement

CallableStatement interface is used to call the **stored procedures and functions, made by using PL/SQL**.

Stored procedure and functions in SQL are the collection of SQL statements to perform business logic on the database and to made calculations.

Difference between procedure and function is that function must return a value. A procedure can call a function but reverse is not true. A procedure can have input and output but function can only have input. Procedure can handle exception but function cannot.

Making Instance of CallableStatement :

```
CallableStatement stmt=con.prepareCall("{call procedureName(?,?)}");
```

Where procedure name is the name of procedure which we want to call.

The said procedure can receive two arguments (?,?).

For Example :

Following is the procedure with name Proc :

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

Table Structure is like this :

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

Following is the complete program to run the above procedure which can accept 2 argument for id and name:

```
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 cstmt=con.prepareCall("{call proc(?,?)}");
cstmt.setInt(1,101);
cstmt.setString(2,"XYZ");
cstmt.execute();

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

Calling Function through CallableStatement

```
CallableStatement cstmt=con.prepareCall("{?= call FunctionName(?,?)}");
```

Here first '?' mark represent the return value.

To accept the return value CallableStatement have special function :

Cstmt.**registerOutParameter**(1,Type.INTEGER)

Here 1 represent first argument, Type tells the CallableStatement about the data type of return value.

Example :

SQL Function :

```
create or replace function sum4
(n1 in number,n2 in number)
return number
is
temp number(8);
begin
temp :=n1+n2;
```

```
return temp;
end;
/
```

```
import java.sql.*;
public class FuncSum {
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 cstmt=con.prepareCall("{?= call sum4(?,?)}");
cstmt.setInt(2,10);
cstmt.setInt(3,43);
cstmt.registerOutParameter(1,Types.INTEGER);
cstmt.execute();

System.out.println(cstmt.getInt(1));
} }
```

Creating Table

```
import java.sql.*;

public class TestApplication {

  static final String DB_URL = "jdbc:mysql://localhost/TUTORIALSPOINT";
  static final String USER = "guest";
  static final String PASS = "guest123";

  public static void main(String args[]) {
    try{
      Connection conn = DriverManager.getConnection(DB_URL, USER, PASS);
      Statement stmt = conn.createStatement();

      String QUERY1 = "CREATE TEMPORARY TABLE EMPLOYEES_COPY SELECT * FROM
EMPLOYEES";
      stmt.execute(QUERY1);
      String QUERY2 = "SELECT * FROM EMPLOYEES_COPY";
      ResultSet rs = stmt.executeQuery(QUERY2);

      while (rs.next()){
        System.out.print("Id: " + rs.getInt("id"));
        System.out.print(" Age: " + rs.getInt("age"));
        System.out.print(" First: " + rs.getString("first"));
        System.out.println(" Last: " + rs.getString("last"));
        System.out.println("----------------------------------------");
      }
    }catch (SQLException e){
      e.printStackTrace();
    }
  }
```