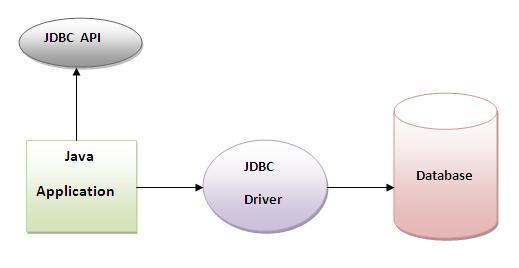
# **Java JDBC Tutorial**

Java JDBC is a java API to connect and execute query with the database. JDBC API uses jdbc drivers to connect with the database.



Why use JDBC

Before JDBC, ODBC API was the database API to connect and execute 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).

What is API

API (Application programming interface) is a document that contains description of all the features of a product or software. It represents classes and interfaces that software programs can follow to communicate with each other. An API can be created for applications, libraries, operating systems, etc

# **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.

# **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. |

**Syntax of forName() method**

**public** **static** **void** forName(String className)**throws** ClassNotFoundException

Example to register the OracleDriver class

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

2) Create the connection object

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

**Syntax of getConnection() method**

1) **public** **static** Connection getConnection(String url)**throws** SQLException

2) **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");

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();

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));

}

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();

# **Example to connect to the Oracle database**

|  |
| --- |
| For connecting java application with the oracle database, you need to follow 5 steps to perform database connectivity. In this example we are using Oracle10g as the database. So we need to know following informations for the oracle database:   1. **Driver class:**The driver class for the oracle database is **oracle.jdbc.driver.OracleDriver**. 2. **Connection URL:**The connection URL for the oracle10G database is **jdbc:oracle:thin:@localhost:1521:xe** where jdbc is the API, oracle is the database, thin is the driver, localhost is the server name on which oracle is running, we may also use IP address, 1521 is the port number and XE is the Oracle service name. You may get all these informations from the tnsnames.ora file. 3. **Username:**The default username for the oracle database is **system**. 4. **Password:**Password is given by the user at the time of installing the oracle database. |

|  |
| --- |
| Let's first create a table in oracle database. |

1. create table emp(id number(10),name varchar2(40),age number(3));

### Example to Connect Java Application with Oracle database

In this example, system is the username and oracle is the password of the Oracle database.

**import** java.sql.\*;

**class** OracleCon{

**public** **static** **void** main(String args[]){

**try**{

//step1 load the driver class

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

//step2 create  the connection object

Connection con=DriverManager.getConnection(

"jdbc:oracle:thin:@localhost:1521:xe","system","oracle");

//step3 create the statement object

Statement stmt=con.createStatement();

//step4 execute query

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

**while**(rs.next())

System.out.println(rs.getInt(1)+"  "+rs.getString(2)+"  "+rs.getString(3));

//step5 close the connection object

con.close();

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

}

}

The above example will fetch all the records of emp table.

To connect java application with the Oracle database ojdbc14.jar file is required to be loaded.

### Two ways to load the jar file:

1. paste the ojdbc14.jar file in jre/lib/ext folder
2. set classpath

### 1) paste the ojdbc14.jar file in JRE/lib/ext folder:

|  |
| --- |
| Firstly, search the ojdbc14.jar file then go to JRE/lib/ext folder and paste the jar file here. |

### 2) set classpath:

|  |
| --- |
| There are two ways to set the classpath:   * temporary * permanent |

### How to set the temporary classpath:

|  |
| --- |
| Firstly, search the ojdbc14.jar file then open command prompt and write: |

C:>set classpath=c:\folder\ojdbc14.jar;.;

### How to set the permanent classpath:

|  |
| --- |
| Go to environment variable then click on new tab. In variable name write **classpath** and in variable value paste the path to ojdbc14.jar by appending ojdbc14.jar;.; as C:\oraclexe\app\oracle\product\10.2.0\server\jdbc\lib\ojdbc14.jar;.; |
| To see the slides of seting parmanent path [click here](http://www.javatpoint.com/how-to-set-path-in-java) |

# **Example to connect to the mysql database**

For connecting java application with the mysql database, you need to follow 5 steps to perform database connectivity.

In this example we are using MySql as the database. So we need to know following informations for the mysql database:

1. **Driver class:**The driver class for the mysql database is **com.mysql.jdbc.Driver**.
2. **Connection URL:**The connection URL for the mysql database is **jdbc:mysql://localhost:3306/sonoo** where jdbc is the API, mysql is the database, localhost is the server name on which mysql is running, we may also use IP address, 3306 is the port number and sonoo is the database name. We may use any database, in such case, you need to replace the sonoo with your database name.
3. **Username:**The default username for the mysql database is **root**.
4. **Password:**Password is given by the user at the time of installing the mysql database. In this example, we are going to use root as the password.

Let's first create a table in the mysql database, but before creating table, we need to create database first.

create database sonoo;

use sonoo;

create table emp(id **int**(10),name varchar(40),age **int**(3));

### Example to Connect Java Application with mysql database

In this example, sonoo is the database name, root is the username and password.

**import** java.sql.\*;

**class** MysqlCon{

**public** **static** **void** main(String args[]){

**try**{

Class.forName("com.mysql.jdbc.Driver");

Connection con=DriverManager.getConnection(

"jdbc:mysql://localhost:3306/sonoo","root","root");

//here sonoo is database name, root is username and password

Statement stmt=con.createStatement();

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

**while**(rs.next())

System.out.println(rs.getInt(1)+"  "+rs.getString(2)+"  "+rs.getString(3));

con.close();

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

}

}

The above example will fetch all the records of emp table.

To connect java application with the mysql database mysqlconnector.jar file is required to be loaded.

### Two ways to load the jar file:

1. paste the mysqlconnector.jar file in jre/lib/ext folder
2. set classpath

### 1) paste the mysqlconnector.jar file in JRE/lib/ext folder:

|  |
| --- |
| Download the mysqlconnector.jar file. Go to jre/lib/ext folder and paste the jar file here. |

### 2) set classpath:

|  |
| --- |
| There are two ways to set the classpath:   * temporary * permament |

### How to set the temporary classpath

|  |
| --- |
| open comman prompt and write: |

C:>set classpath=c:\folder\mysql-connector-java-5.0.8-bin.jar;.;

### How to set the permanent classpath

Go to environment variable then click on new tab. In variable name write **classpath** and in variable value paste the path to the mysqlconnector.jar file by appending mysqlconnector.jar;.; as C:\folder\mysql-connector-java-5.0.8-bin.jar;.;

# **Connectivity with Access without DSN**

There are two ways to connect java application with the access database.

1. Without DSN (Data Source Name)
2. With DSN

Java is mostly used with Oracle, mysql, or DB2 database. So you can learn this topic only for knowledge.

### Example to Connect Java Application with access without DSN

In this example, we are going to connect the java program with the access database. In such case, we have created the login table in the access database. There is only one column in the table named name. Let's get all the name of the login table.

**import** java.sql.\*;

**class** Test{

**public** **static** **void** main(String ar[]){

**try**{

   String database="student.mdb";//Here database exists in the current directory

   String url="jdbc:odbc:Driver={Microsoft Access Driver (\*.mdb)};

                    DBQ=" + database + ";DriverID=22;READONLY=**true**";

   Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");

   Connection c=DriverManager.getConnection(url);

   Statement st=c.createStatement();

   ResultSet rs=st.executeQuery("select \* from login");

**while**(rs.next()){

    System.out.println(rs.getString(1));

   }

}**catch**(Exception ee){System.out.println(ee);}

}}

### Example to Connect Java Application with access with DSN

Connectivity with type1 driver is not considered good. To connect java application with type1 driver, create DSN first, here we are assuming your dsn name is mydsn.

**import** java.sql.\*;

**class** Test{

**public** **static** **void** main(String ar[]){

**try**{

   String url="jdbc:odbc:mydsn";

   Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");

   Connection c=DriverManager.getConnection(url);

   Statement st=c.createStatement();

   ResultSet rs=st.executeQuery("select \* from login");

**while**(rs.next()){

    System.out.println(rs.getString(1));

   }

}**catch**(Exception ee){System.out.println(ee);}

}}

# **DriverManager class:**

The DriverManager class acts as an interface between user and drivers. It keeps track of the drivers that are available and handles establishing a connection between a database and the appropriate driver. The DriverManager class maintains a list of Driver classes that have registered themselves by calling the method DriverManager.registerDriver().

### Commonly used methods of DriverManager class:

|  |  |
| --- | --- |
| 1) public static void registerDriver(Driver driver): | is used to register the given driver with DriverManager. |
| 2) public static void deregisterDriver(Driver driver): | is used to deregister the given driver (drop the driver from the list) with DriverManager. |
| 3) public static Connection getConnection(String url): | is used to establish the connection with the specified url. |
| 4) public static Connection getConnection(String url,String userName,String password): | is used to establish the connection with the specified url, username and password. |

# **Connection interface:**

A Connection is the session between java application and database. The Connection interface is a factory of Statement, PreparedStatement, and DatabaseMetaData i.e. object of Connection can be used to get the object of Statement and DatabaseMetaData. The Connection interface provide many methods for transaction management like commit(),rollback() etc.

#### By default, connection commits the changes after executing queries.

### Commonly used methods of Connection interface:

|  |
| --- |
| **1) public Statement createStatement():** creates a statement object that can be used to execute SQL queries. |
| **2) public Statement createStatement(int resultSetType,int resultSetConcurrency):** Creates a Statement object that will generate ResultSet objects with the given type and concurrency. |
| **3) public void setAutoCommit(boolean status):** is used to set the commit status.By default it is true. |
| **4) public void commit():** saves the changes made since the previous commit/rollback permanent. |
| **5) public void rollback():** Drops all changes made since the previous commit/rollback. |
| **6) public void close():** closes the connection and Releases a JDBC resources immediately. |

# **Statement interface**

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.

### 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. |

### Example of Statement interface

Let’s see the simple example of Statement interface to insert, update and delete the record.

**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();

}}

# **ResultSet interface**

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

#### By default, ResultSet object can be moved forward only and it is not updatable.

But we can make this object to move forward and backward direction by passing either TYPE\_SCROLL\_INSENSITIVE or TYPE\_SCROLL\_SENSITIVE in createStatement(int,int) method as well as we can make this object as updatable by:

Statement stmt = con.createStatement(ResultSet.TYPE\_SCROLL\_INSENSITIVE,  ResultSet.CONCUR\_UPDATABLE);

### Commonly used methods of ResultSet interface

|  |  |
| --- | --- |
| **1) public boolean next():** | is used to move the cursor to the one row next from the current position. |
| **2) public boolean previous():** | is used to move the cursor to the one row previous from the current position. |
| **3) public boolean first():** | is used to move the cursor to the first row in result set object. |
| **4) public boolean last():** | is used to move the cursor to the last row in result set object. |
| **5) public boolean absolute(int row):** | is used to move the cursor to the specified row number in the ResultSet object. |
| **6) public boolean relative(int row):** | is used to move the cursor to the relative row number in the ResultSet object, it may be positive or negative. |
| **7) public int getInt(int columnIndex):** | is used to return the data of specified column index of the current row as int. |
| **8) public int getInt(String columnName):** | is used to return the data of specified column name of the current row as int. |
| **9) public String getString(int columnIndex):** | is used to return the data of specified column index of the current row as String. |
| **10) public String getString(String columnName):** | is used to return the data of specified column name of the current row as String. |

### Example of Scrollable ResultSet

Let’s see the simple example of ResultSet interface to retrieve the data of 3rd row.

**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(ResultSet.TYPE\_SCROLL\_SENSITIVE,ResultSet.CONCUR\_UPDATABLE);

ResultSet rs=stmt.executeQuery("select \* from emp765");

//getting the record of 3rd row

rs.absolute(3);

System.out.println(rs.getString(1)+" "+rs.getString(2)+" "+rs.getString(3));

con.close();

}}

# **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.

### Why use PreparedStatement?

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

#### 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{}

### Methods of PreparedStatement interface

The important methods of PreparedStatement interface are given below:

|  |  |
| --- | --- |
| **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. |

### Example of PreparedStatement interface that inserts the record

First of all create table as given below:

create table emp(id number(10),name varchar2(50));

Now insert records in this table by the code given below:

**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);//1 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);}

}

}

### Example of PreparedStatement interface that updates the record

PreparedStatement stmt=con.prepareStatement("update emp set name=? where id=?");

stmt.setString(1,"Sonoo");//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");

### Example of PreparedStatement interface that deletes the record

PreparedStatement stmt=con.prepareStatement("delete from emp where id=?");

stmt.setInt(1,101);

**int** i=stmt.executeUpdate();

System.out.println(i+" records deleted");

### Example of PreparedStatement interface that retrieve the records of a table

PreparedStatement stmt=con.prepareStatement("select \* from emp");

ResultSet rs=stmt.executeQuery();

**while**(rs.next()){

System.out.println(rs.getInt(1)+" "+rs.getString(2));

}

### 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 emp130 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();

}}

# **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.

## Commonly used methods of ResultSetMetaData interface

|  |  |
| --- | --- |
| Method | Description |
| 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. |

### How to get the object of ResultSetMetaData:

|  |
| --- |
| The getMetaData() method of ResultSet interface returns the object of ResultSetMetaData. Syntax: |

**public** ResultSetMetaData getMetaData()**throws** SQLException

### Example of ResultSetMetaData interface :

**import** java.sql.\*;

**class** Rsmd{

**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 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));

con.close();

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

}

}

Output:Total columns: 2

Column Name of 1st column: ID

Column Type Name of 1st column: NUMBER

# **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. |

### How to get the object of DatabaseMetaData:

The getMetaData() method of Connection interface returns the object of DatabaseMetaData. Syntax:

**public** DatabaseMetaData getMetaData()**throws** SQLException

### Simple Example of DatabaseMetaData interface :

**import** java.sql.\*;

**class** Dbmd{

**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");

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());

con.close();

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

}

}

Output:Driver Name: Oracle JDBC Driver

Driver Version: 10.2.0.1.0XE

Database Product Name: Oracle

Database Product Version: Oracle Database 10g Express Edition

Release 10.2.0.1.0 -Production

### Example of DatabaseMetaData interface that prints total number of tables :

**import** java.sql.\*;

**class** Dbmd2{

**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");

DatabaseMetaData dbmd=con.getMetaData();

String table[]={"TABLE"};

ResultSet rs=dbmd.getTables(**null**,**null**,**null**,table);

**while**(rs.next()){

System.out.println(rs.getString(3));

}

con.close();

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

}

}

### Example of DatabaseMetaData interface that prints total number of views :

**import** java.sql.\*;

**class** Dbmd3{

**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");

DatabaseMetaData dbmd=con.getMetaData();

String table[]={"VIEW"};

ResultSet rs=dbmd.getTables(**null**,**null**,**null**,table);

**while**(rs.next()){

System.out.println(rs.getString(3));

}

con.close();

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

}

}

# **Example to store image in Oracle database**

You can store images in the database in java by the help of **PreparedStatement** interface.

The **setBinaryStream()** method of PreparedStatement is used to set Binary information into the parameterIndex.

### Signature of setBinaryStream method

The syntax of setBinaryStream() method is given below:

1) **public** **void** setBinaryStream(**int** paramIndex,InputStream stream) **throws** SQLException

2) **public** **void** setBinaryStream(**int** paramIndex,InputStream stream,**long** length) **throws** SQLException

For storing image into the database, BLOB (Binary Large Object) datatype is used in the table. For example:

CREATE TABLE  "IMGTABLE"

   (    "NAME" VARCHAR2(4000),

    "PHOTO" BLOB

   )

/

Let's write the jdbc code to store the image in the database. Here we are using d:\\d.jpg for the location of image. You can change it according to the image location.

**import** java.sql.\*;

**import** java.io.\*;

**public** **class** InsertImage {

**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 ps=con.prepareStatement("insert into imgtable values(?,?)");

ps.setString(1,"sonoo");

FileInputStream fin=**new** FileInputStream("d:\\g.jpg");

ps.setBinaryStream(2,fin,fin.available());

**int** i=ps.executeUpdate();

System.out.println(i+" records affected");

con.close();

}**catch** (Exception e) {e.printStackTrace();}

}

}

If you see the table, record is stored in the database but image will not be shown. To do so, you need to retrieve the image from the database which we are covering in the next page.

# **Example to retrieve image from Oracle database**

By the help of **PreparedStatement** we can retrieve and store the image in the database.

The **getBlob()** method of PreparedStatement is used to get Binary information, it returns the instance of Blob. After calling the **getBytes()** method on the blob object, we can get the array of binary information that can be written into the image file.

### Signature of getBlob() method of PreparedStatement

**public** Blob getBlob()**throws** SQLException

### Signature of getBytes() method of Blob interface

**public**  **byte**[] getBytes(**long** pos, **int** length)**throws** SQLException

We are assuming that image is stored in the imgtable.

CREATE TABLE  "IMGTABLE"

   (    "NAME" VARCHAR2(4000),

    "PHOTO" BLOB

   )

/

Now let's write the code to retrieve the image from the database and write it into the directory so that it can be displayed.

In AWT, it can be displayed by the Toolkit class. In servlet, jsp, or html it can be displayed by the img tag.

**import** java.sql.\*;

**import** java.io.\*;

**public** **class** RetrieveImage {

**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 ps=con.prepareStatement("select \* from imgtable");

ResultSet rs=ps.executeQuery();

**if**(rs.next()){//now on 1st row

Blob b=rs.getBlob(2);//2 means 2nd column data

**byte** barr[]=b.getBytes(1,(**int**)b.length());//1 means first image

FileOutputStream fout=**new** FileOutputStream("d:\\sonoo.jpg");

fout.write(barr);

fout.close();

}//end of if

System.out.println("ok");

con.close();

}**catch** (Exception e) {e.printStackTrace();  }

}

}

Now if you see the d drive, sonoo.jpg image is created.

# **Example to store file in Oracle database:**

The setCharacterStream() method of PreparedStatement is used to set character information into the parameterIndex.

### Syntax:

|  |
| --- |
| 1) public void setBinaryStream(int paramIndex,InputStream stream)throws SQLException |
| 2) public void setBinaryStream(int paramIndex,InputStream stream,long length)throws SQLException |

For storing file into the database, CLOB (Character Large Object) datatype is used in the table. For example:

CREATE TABLE  "FILETABLE"

   (    "ID" NUMBER,

    "NAME" CLOB

   )

/

**import** java.io.\*;

**import** java.sql.\*;

**public** **class** StoreFile {

**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 ps=con.prepareStatement(

"insert into filetable values(?,?)");

File f=**new** File("d:\\myfile.txt");

FileReader fr=**new** FileReader(f);

ps.setInt(1,101);

ps.setCharacterStream(2,fr,(**int**)f.length());

**int** i=ps.executeUpdate();

System.out.println(i+" records affected");

con.close();

}**catch** (Exception e) {e.printStackTrace();}

}

}

# **Example to retrieve file from Oracle database:**

The getClob() method of PreparedStatement is used to get file information from the database.

### Syntax of getClob method

**public** Clob getClob(**int** columnIndex){}

Let's see the table structure of this example to retrieve the file.

CREATE TABLE  "FILETABLE"

   (    "ID" NUMBER,

    "NAME" CLOB

   )

/

The example to retrieve the file from the Oracle database is given below.

**import** java.io.\*;

**import** java.sql.\*;

**public** **class** RetrieveFile {

**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 ps=con.prepareStatement("select \* from filetable");

ResultSet rs=ps.executeQuery();

rs.next();//now on 1st row

Clob c=rs.getClob(2);

Reader r=c.getCharacterStream();

FileWriter fw=**new** FileWriter("d:\\retrivefile.txt");

**int** i;

**while**((i=r.read())!=-1)

fw.write((**char**)i);

fw.close();

con.close();

System.out.println("success");

}**catch** (Exception e) {e.printStackTrace();  }

}

}

# **CallableStatement Interface**

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

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.

### What is the difference between stored procedures and functions.

The differences between stored procedures and functions are given below:

|  |  |
| --- | --- |
| Stored Procedure | Function |
| is used to perform business logic. | is used to perform calculation. |
| must not have the return type. | must have the return type. |
| may return 0 or more values. | may return only one values. |
| We can call functions from the procedure. | Procedure cannot be called from function. |
| Procedure supports input and output parameters. | Function supports only input parameter. |
| Exception handling using try/catch block can be used in stored procedures. | Exception handling using try/catch can't be used in user defined functions. |

### 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(?,?...?)}");

The example to get the instance of CallableStatement is given below:

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

It calls the procedure myprocedure that receives 2 arguments.

### Full example to call the stored procedure using JDBC

To call the stored procedure, you need to create it in the database. Here, we are assuming that stored procedure looks like this.

create or replace procedure "INSERTR"

(id IN NUMBER,

name IN VARCHAR2)

is

begin

insert into user420 values(id,name);

end;

/

The table structure is given below:

create table user420(id number(10), name varchar2(200));

In this example, we are going to call the stored procedure INSERTR that receives id and name as the parameter and inserts it into the table user420. Note that you need to create the user420 table as well to run this application.

**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 insertR(?,?)}");

stmt.setInt(1,1011);

stmt.setString(2,"Amit");

stmt.execute();

System.out.println("success");

}

}

Now check the table in the database, value is inserted in the user420 table.

### Example to call the function using JDBC

In this example, we are calling the sum4 function that receives two input and returns the sum of the given number. Here, we have used the **registerOutParameter** method of CallableStatement interface, that registers the output parameter with its corresponding type. It provides information to the CallableStatement about the type of result being displayed.

The **Types** class defines many constants such as INTEGER, VARCHAR, FLOAT, DOUBLE, BLOB, CLOB etc.

Let's create the simple function in the database first.

create or replace function sum4

(n1 in number,n2 in number)

**return** number

is

temp number(8);

begin

temp :=n1+n2;

**return** temp;

end;

/

Now, let's write the simple program to call the function.

**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 stmt=con.prepareCall("{?= call sum4(?,?)}");

stmt.setInt(2,10);

stmt.setInt(3,43);

stmt.registerOutParameter(1,Types.INTEGER);

stmt.execute();

System.out.println(stmt.getInt(1));

}

}

Output: 53

# **Transaction Management in JDBC**

Transaction represents **a single unit of work**.

The ACID properties describes the transaction management well. ACID stands for Atomicity, Consistency, isolation and durability.

**Atomicity** means either all successful or none.

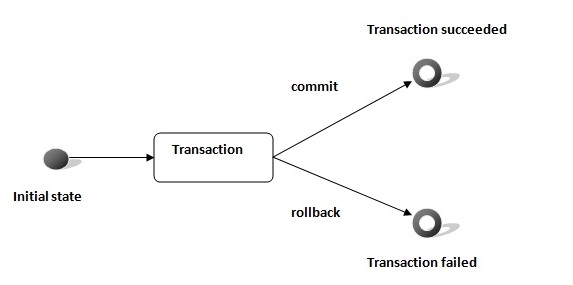
**Consistency** ensures bringing the database from one consistent state to another consistent state.

**Isolation** ensures that transaction is isolated from other transaction.

**Durability** means once a transaction has been committed, it will remain so, even in the event of errors, power loss etc.

#### Advantage of Transaction Mangaement

**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. |

### Simple example of transaction management in jdbc using Statement

Let's see the simple example of transaction management using Statement.

**import** java.sql.\*;

**class** FetchRecords{

**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.executeUpdate("insert into user420 values(190,'abhi',40000)");

stmt.executeUpdate("insert into user420 values(191,'umesh',50000)");

con.commit();

con.close();

}}

If you see the table emp400, you will see that 2 records has been added.

### Example of transaction management in jdbc using PreparedStatement

Let's see the simple example of transaction management using PreparedStatement.

**import** java.sql.\*;

**import** java.io.\*;

**class** TM{

**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");

con.setAutoCommit(**false**);

PreparedStatement ps=con.prepareStatement("insert into user420 values(?,?,?)");

BufferedReader br=**new** BufferedReader(**new** InputStreamReader(System.in));

**while**(**true**){

System.out.println("enter id");

String s1=br.readLine();

**int** id=Integer.parseInt(s1);

System.out.println("enter name");

String name=br.readLine();

System.out.println("enter salary");

String s3=br.readLine();

**int** salary=Integer.parseInt(s3);

ps.setInt(1,id);

ps.setString(2,name);

ps.setInt(3,salary);

ps.executeUpdate();

System.out.println("commit/rollback");

String answer=br.readLine();

**if**(answer.equals("commit")){

con.commit();

}

**if**(answer.equals("rollback")){

con.rollback();

}

System.out.println("Want to add more records y/n");

String ans=br.readLine();

**if**(ans.equals("n")){

**break**;

}

}

con.commit();

System.out.println("record successfully saved");

con.close();//before closing connection commit() is called

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

}}

It will ask to add more records until you press n. If you press n, transaction is committed.

# **Batch Processing in JDBC**

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

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

#### Advantage of Batch Processing

Fast Performance

#### 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. |

### Example of batch processing in jdbc

Let's see the simple example of 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** FetchRecords{

**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();

}}

If you see the table user420, two records has been added.

### Example of batch processing using PreparedStatement

**import** java.sql.\*;

**import** java.io.\*;

**class** BP{

**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 ps=con.prepareStatement("insert into user420 values(?,?,?)");

BufferedReader br=**new** BufferedReader(**new** InputStreamReader(System.in));

**while**(**true**){

System.out.println("enter id");

String s1=br.readLine();

**int** id=Integer.parseInt(s1);

System.out.println("enter name");

String name=br.readLine();

System.out.println("enter salary");

String s3=br.readLine();

**int** salary=Integer.parseInt(s3);

ps.setInt(1,id);

ps.setString(2,name);

ps.setInt(3,salary);

ps.addBatch();

System.out.println("Want to add more records y/n");

String ans=br.readLine();

**if**(ans.equals("n")){

**break**;

}

}

ps.executeBatch();

System.out.println("record successfully saved");

con.close();

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

}}

It will add the queries into the batch until user press n. Finally it executes the batch. Thus all the added queries will be fired.

# **JDBC RowSet**

The instance of **RowSet** is the java bean component because it has properties and java bean notification mechanism. It is introduced since JDK 5.

It is the wrapper of ResultSet. It holds tabular data like ResultSet but it is easy and flexible to use.

The implementation classes of RowSet interface are as follows:

* JdbcRowSet
* CachedRowSet
* WebRowSet
* JoinRowSet
* FilteredRowSet

Let's see how to create and execute RowSet.

JdbcRowSet rowSet = RowSetProvider.newFactory().createJdbcRowSet();

rowSet.setUrl("jdbc:oracle:thin:@localhost:1521:xe");

rowSet.setUsername("system");

rowSet.setPassword("oracle");

rowSet.setCommand("select \* from emp400");

rowSet.execute();

#### It is the new way to get the instance of JdbcRowSet since JDK 7.

#### Advantage of RowSet

The advantages of using RowSet are given below:

1. It is easy and flexible to use
2. It is Scrollable and Updatable bydefault

### Simple example of JdbcRowSet

Let's see the simple example of JdbcRowSet without event handling code.

**import** java.sql.Connection;

**import** java.sql.DriverManager;

**import** java.sql.ResultSet;

**import** java.sql.Statement;

**import** javax.sql.RowSetEvent;

**import** javax.sql.RowSetListener;

**import** javax.sql.rowset.JdbcRowSet;

**import** javax.sql.rowset.RowSetProvider;

**public** **class** RowSetExample {

**public** **static** **void** main(String[] args) **throws** Exception {

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

    //Creating and Executing RowSet

        JdbcRowSet rowSet = RowSetProvider.newFactory().createJdbcRowSet();

        rowSet.setUrl("jdbc:oracle:thin:@localhost:1521:xe");

        rowSet.setUsername("system");

        rowSet.setPassword("oracle");

        rowSet.setCommand("select \* from emp400");

        rowSet.execute();

**while** (rowSet.next()) {

                        // Generating cursor Moved event

                        System.out.println("Id: " + rowSet.getString(1));

                        System.out.println("Name: " + rowSet.getString(2));

                        System.out.println("Salary: " + rowSet.getString(3));

                }

        }

}

The output is given below:

Id: 55

Name: Om Bhim

Salary: 70000

Id: 190

Name: abhi

Salary: 40000

Id: 191

Name: umesh

Salary: 50000

### Full example of Jdbc RowSet with event handling

To perform event handling with JdbcRowSet, you need to add the instance of **RowSetListener** in the addRowSetListener method of JdbcRowSet.

The RowSetListener interface provides 3 method that must be implemented. They are as follows:

1) public void cursorMoved(RowSetEvent event);

2) public void rowChanged(RowSetEvent event);

3) public void rowSetChanged(RowSetEvent event);

Let's write the code to retrieve the data and perform some additional tasks while cursor is moved, cursor is changed or rowset is changed. The event handling operation can't be performed using ResultSet so it is preferred now.

**import** java.sql.Connection;

**import** java.sql.DriverManager;

**import** java.sql.ResultSet;

**import** java.sql.Statement;

**import** javax.sql.RowSetEvent;

**import** javax.sql.RowSetListener;

**import** javax.sql.rowset.JdbcRowSet;

**import** javax.sql.rowset.RowSetProvider;

**public** **class** RowSetExample {

**public** **static** **void** main(String[] args) **throws** Exception {

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

    //Creating and Executing RowSet

    JdbcRowSet rowSet = RowSetProvider.newFactory().createJdbcRowSet();

    rowSet.setUrl("jdbc:oracle:thin:@localhost:1521:xe");

    rowSet.setUsername("system");

    rowSet.setPassword("oracle");

        rowSet.setCommand("select \* from emp400");

        rowSet.execute();

    //Adding Listener and moving RowSet

    rowSet.addRowSetListener(**new** MyListener());

**while** (rowSet.next()) {

                        // Generating cursor Moved event

                        System.out.println("Id: " + rowSet.getString(1));

                        System.out.println("Name: " + rowSet.getString(2));

                        System.out.println("Salary: " + rowSet.getString(3));

                }

        }

}

**class** MyListener **implements** RowSetListener {

**public** **void** cursorMoved(RowSetEvent event) {

                System.out.println("Cursor Moved...");

      }

**public** **void** rowChanged(RowSetEvent event) {

                System.out.println("Cursor Changed...");

     }

**public** **void** rowSetChanged(RowSetEvent event) {

                System.out.println("RowSet changed...");

     }

}

The output is as follows:

Cursor Moved...

Id: 55

Name: Om Bhim

Salary: 70000

Cursor Moved...

Id: 190

Name: abhi

Salary: 40000

Cursor Moved...

Id: 191

Name: umesh

Salary: 50000

Cursor Moved...

# **Jdbc New Features**

The latest version of JDBC is 4.0 currently. Java has updated jdbc api to ease and simplify the coding to database interactivity.

Here, we are going to see the features included in Jdbc 3.0 and Jdbc 4.0.

## Jdbc 3.0 Features

The important features of JDBC API 3.0 are as follows:

* **Jdbc RowSet** We have done the great discussion on JdbcRowSet in the previous page.
* **Savepoint in transaction management** Now you are able to create, rollback and release the savepoint by Connection.setSavepoint(), Connection.rollback(Savepoint svpt) and Connection.releaseSavepoint(Savepoint svpt) methods.
* **Statement and ResultSet Caching for Connection Pooling** Now you are able to reuse the statement and result set because jdbc 3 provides you the facility of statement caching and result set caching.
* **Switching between Global and Local Transactions**
* **Retrieval of auto generated keys** Now you are able to get the auto generated keys by the method getGeneratedKeys().

## Jdbc 4.0 Features

The important features of JDBC API 4.0 are given below:

* **Automatic Loading of Driver class** You don't need to write Class.forName() now because it is loaded bydefault since jdbc4.
* **Subclasses of SQLException** Jdbc 4 provides new subclasses of SQLException class for better readability and handling.
* **New methods** There are many new methods introduced in Connection, PreparedStatement, CallableStatement, ResultSet etc.
* **Improved DataSource** Now data source implementation is improved.
* **Event Handling support in Statement for Connection Pooling** Now Connection Pooling can listen statement error and statement closing events.