

## Java Database Connectivity (JDBC)

## Basic Steps in Using JDBC

1. Import required packages
2. Load driver
3. Define Connection URL
4. Establish Connection
5. Create a Statement object

JDBC

## Basic Steps in Using JDBC (cont.)

6. Execute query / DML
7. Process results
8. Close connection

JDBC



## JDBC Details of Process

## JDBC: Details of Process

### 1. Import package

- Import java.sql package
  - `import java.sql.*;`

JDBC

## JDBC: Details of Process

### 2. Loading driver

- Need to load suitable driver for underlying database
- Different drivers for different databases are available
  - For MS Access
 

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```
  - For Oracle
 

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

JDBC

## JDBC: Details of Process

### 3. Define Connection URL

- To get a connection, we need to specify URL of database.
- If you are using a JDBC-ODBC driver you need to create a DSN. DSN is the name of your DataSource
- If the name of your DSN is "personDSN" then the url of the database will be
  - `String conURL = "jdbc:odbc:personDSN"`

JDBC

## JDBC: Details of Process, cont.

### 4. Establish Connection

- `Connection con = null;`
  - Use driver manager to get the connection object
 

```
con = DriverManager.getConnection(conURL);
```
  - If the Db requires username and password you can use overloaded version
    - `String usr = "umair";`
    - `String pswd = "java";`
    - `Connection con = null;`
- ```
CON = DriverManager.getConnection(conURL,usr,pswd);
```

JDBC

## JDBC: Details of Process, cont.

### 5. Create Statement

- A statement is obtained from a Connection object.
 

```
Statement statement = con.createStatement();
```
- Once you have a statement, you can use it for various kind of SQL queries

JDBC

## JDBC: Details of Process, cont.

### 6(a) Execute Query / DML

- `executeQuery(sql)` method
    - Used for SQL SELECT queries
    - Returns the *ResultSet* object which is used to access the rows of the query results
- ```
String sql = "SELECT * FROM sometable";
ResultSet rs = statement.executeQuery(sql);
```

JDBC

## JDBC: Details of Process, cont.

### 6(b) Execute Query / DML

- `executeUpdate(sql)` method
    - Used for an update statement ( INSERT, UPDATE or DELETE)
    - Returns an integer value representing the number of rows updated.
- ```
String sql = "INSERT INTO tableName " +
            "(columnNames) Values (values)";

int count = statement.executeUpdate(sql);
```

JDBC

## JDBC: Details of Process, cont.

### 7. Process Results

- *ResultSet* provides various `getXxx` methods that take a column index or name and returns the data
- First column has index 1, not 0
 

```
while(resultSet.next()) {
    //by using column name
    String name = rs.getString("columnName");
    //or by using index
    String name = rs.getString(1);
}
```

JDBC

## JDBC: Details of Process, cont.

### 8. Close Connection

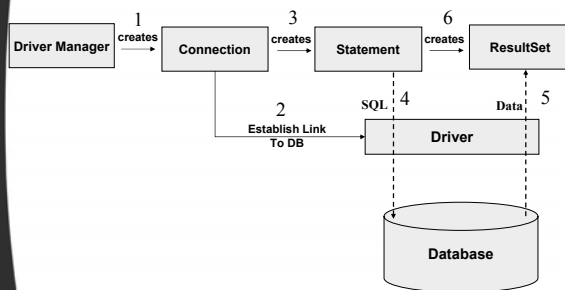
```
connection.close();
```

- As opening a connection is expensive, postpone this step if additional database operations are expected

## In a nut shell

- `Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");`
- `Connection con = null;`  
`con = DriverManager.getConnection(url, usr, pwd);`
- `Statement st = con.createStatement();`
- `ResultSet rs = st.executeQuery("Select * from Person");`

## JDBC Architecture



## Example Code

### Retrieving Data from ResultSet

## Example Code

### Retrieving Data from ResultSet

```
//Step 1: import package
import java.sql.*;

public class JdbcEx {

    public static void main (String args []){

        try {

            //Step 2: Load driver
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");

            //Step 3: Define the connection URL
            String url = "jdbc:odbc:personDSN";

            //Step 4: Establish the connection
            Connection con = null;
            con = DriverManager.getConnection (url , "", "");
```

## Example Code 14.1

### Retrieving Data from ResultSet (cont.)

```
//Step 5: create the statement
Statement st = con.createStatement();

//Step 6: Execute the query
String sql = "SELECT * FROM Person";
ResultSet rs = st.executeQuery(sql);

//Step 7: Process the results
while ( rs.next() ) {

    String name = rs.getString("name");
    String add = rs.getString("address");
    String pNum = rs.getString("phoneNum");

    System.out.println(name + " " +add + " " +pNum);
} // end while
```

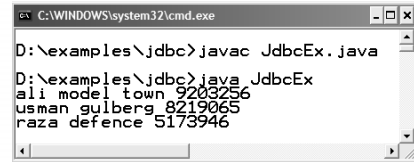
### Example Code 14.1 Retrieving Data from ResultSet (cont.)

```
//Step 8: close the connection
con.close();

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

} //end main
} //end class
```

### Compile & Execute



```
C:\WINDOWS\system32\cmd.exe
D:\examples\jdbc>javac JdbcEx.java
D:\examples\jdbc>java JdbcEx
ali model town 9203256
usman gulberg 8219065
raza defence 5175946
```



## More on JDBC

### Useful Statement Methods

- **executeQuery**
    - Executes the SQL query and returns the data in a table (ResultSet)
    - The resulting table may be empty but never null
- ```
ResultSet rs = stmt.executeQuery("select * from table");
```

### Useful Statement Methods

- **executeUpdate**
    - Used to execute for INSERT, UPDATE, or DELETE SQL statements
    - The return is the number of rows that were affected in the database
    - Supports Data Definition Language (DDL) statements CREATE TABLE, DROP TABLE and ALTER TABLE
- ```
int num = stmt.executeUpdate("DELETE FROM Person " +
    "WHERE id = 2");
```

### Example Code Executing SQL DML Statements

```
/* This program will take two command line argument
that are used to update records in the database */

import java.sql.*; //step 1

public class JdbcDmlEx {

    public static void main (String args []){

        try {

            //steps 2 to 5
            Class.forName("driver name");

            Connection con=null;
            con = DriverManager.getConnection(url, usr, pwd);

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

## Example Code Executing SQL DML Statements (cont.)

//Step 6: Execute the Query / DML

```
String addVar = args[0];
String nameVar = args[1];

String sql = "UPDATE Person " +
    "SET address = '" + addVar + "' " +
    "WHERE name = '" + nameVar + "' ";

int num = st.executeUpdate(sql);
```

//Step 7: Process the results of the query

```
System.out.println(num + " records updated");
```

JDBC

## Example Code Executing SQL DML Statements (cont.)

//Step 8: close the connection  
con.close();

```
}catch (Exception sqlEx) {
    System.out.println(sqlEx);
}
```

```
} //end main
```

```
}//end class
```

JDBC

## Compile & Execute

Person : Table

| id | name  | address     | phonenum |
|----|-------|-------------|----------|
| 1  | ali   | mosque lane | 9203256  |
| 2  | usman | gulberg     | 8219065  |
| 3  | raza  | defence     | 5173946  |
| 0  |       |             |          |

Before execution

```
C:\WINDOWS\system32\cmd.exe
D:\examples\jdbc>javac JdbcDmlEx.java
D:\examples\jdbc>java JdbcDmlEx defence ali
1 record updated
```

Person : Table

| id | name  | address | phonenum |
|----|-------|---------|----------|
| 1  | ali   | defence | 9203256  |
| 2  | usman | gulberg | 8219065  |
| 3  | raza  | defence | 5173946  |
| 0  |       |         |          |

After execution

JDBC

## Useful Statement Methods (Continued)

- **getMaxRows( ) / setMaxRows(int)**
  - Determines the number of rows a `ResultSet` may contain
  - Unless explicitly set, the number of rows are unlimited (return value of 0)
- **getQueryTimeout( ) / setQueryTimeout(int)**
  - Specifies the amount of a time (seconds) a driver will wait for a `STATEMENT` to complete before throwing a `SQLException`

JDBC

## Different Types of Statements

### • Overview

- Through the Statement object, SQL statements are sent to the database.
- Three types of statement objects are available:

#### 1. Statement

- for executing a simple SQL statements

#### 2. PreparedStatement

- for executing a precompiled SQL statement passing in parameters

#### 3. CallableStatement

- for executing a database stored procedure

JDBC



## Prepared Statements

## Prepared Statements (Precompiled Queries)

### • Idea

- If you are going to execute similar SQL statements multiple times, using “prepared” (parameterized) statements can be more efficient
- Create a statement in standard form that is sent to the database for compilation before actually being used
- Each time you use it, you simply replace some of the marked parameters (?) using some set methods

## Prepared Statement, Example

```
PreparedStatement pstmt =
    con.prepareStatement("UPDATE tableName " +
        "SET columnName = ? " +
        "WHERE columnName = ?");
```

- First marked parameter(?) has index 1.

```
pstmt.setString(1, stringValue);
pstmt.setInt(2, intValue);

pstmt.executeUpdate();
```



## Example Code Using Prepared Statements

## Example Code: Modify JdbcDmlEx.java Executing Prepared Statements

/\* Modification to the last example code  
to show the usage of prepared statements \*/

```
import java.sql.*; // step1

public class JdbcDmlEx {

    public static void main (String args []){

        try {

            //steps 2 to 4
            Class.forName("driver name");

            Connection con=null;
            con = DriverManager.getConnection(url, usr, pwd);
```

## Example Code: Modify JdbcDmlEx.java Executing Prepared Statements

```
//Step 5: Create the statement
PreparedStatement pstmt = null;

String sql = "UPDATE Person SET address = ? WHERE name = ? ";
pstmt = con.prepareStatement(sql);

//Step 6: Execute the Query
String addVar = args[0];
String nameVar = args[1];

pstmt.setString(1, addVar);
pstmt.setString(2, nameVar);

// sql = "UPDATE Person SET address = "defence" WHERE name = "ali" "
int num = pstmt.executeUpdate();
```

## Example Code: Modify JdbcDmlEx.java 15.1 Executing Prepared Statements

```
//Step 7: Process the results of the query
System.out.println(num + " records updated");

//Step 8: close the connection

} catch (Exception sqlEx) {
    .....
}

} //end main
} //end class
```

## Compile & Execute

Person : Table

| id | name  | address    | phoneNum |
|----|-------|------------|----------|
| 1  | ali   | model town | 9203256  |
| 2  | usman | gulberg    | 8219065  |
| 3  | raza  | defence    | 5173946  |

Before execution

```
C:\WINDOWS\system32\cmd.exe
D:\examples\jdbc> javac JdbcDmlEx.java
D:\examples\jdbc> java JdbcDmlEx defence ali
1 record updated
```

Person : Table

| id | name  | address | phoneNum |
|----|-------|---------|----------|
| 1  | ali   | defence | 9203256  |
| 2  | usman | gulberg | 8219065  |
| 3  | raza  | defence | 5173946  |

After execution

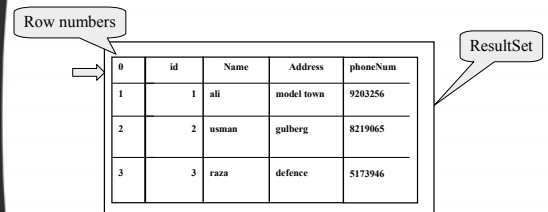


## Result Set

### ResultSet

#### • Overview

- A `ResultSet` contains the results of the SQL query
  - Represented by a table with rows and columns
  - Maintains a cursor pointing to its current row of data.
  - Initially the cursor positioned before the row (0).
  - First row has index 1



### ResultSet (cont.)

#### • A default `ResultSet` object is not updateable and has a cursor that moves forward only

- You can iterate through it only once and only from the first row to last row.

```
String sql = "SELECT * FROM Person";
PreparedStatement pstmt = con.prepareStatement(sql);
ResultSet rs = pstmt.executeQuery();
```

### ResultSet (cont.)

#### • Useful Methods

##### – `next()`

- Attempts to move to the next row in the `ResultSet`
- If successful `true` is returned; otherwise, `false`
- The first call to `next`, moves the cursor to the first row

##### – `close()`

- Releases the JDBC and database resources
- The result set is automatically closed when the associated `Statement` object executes a new query or closed by method call

## ResultSet (cont.)

### • Useful Methods

#### – getters

- Returns the value from the column specified by the column name or index
  - String name = rs.getString("name");
  - String add = rs.getString(3);
  - double sal = rs.getDouble("Salary")

- Returns the value in a specified format

|        |       |      |      |        |
|--------|-------|------|------|--------|
| double | byte  | int  | Date | String |
| float  | short | long | Time | Object |

## ResultSet (cont.)

- It is possible to produce `ResultSet` objects that are scrollable and/or updatable (since JDK 1.2).

```
String sql = "SELECT * FROM Person";
```

```
PreparedStatement pstmt = con.prepareStatement( sql,
    ResultSet.TYPE_SCROLL_INSENSITIVE,
    ResultSet.CONCUR_UPDATABLE );
```

```
ResultSet rs = pstmt.executeQuery( );
```

## ResultSet (cont.)

### • Useful Methods

#### – previous( )

- Moves the cursor to the previous row in the `ResultSet` object.
- Returns `true` if cursor is on a valid row, `false` if it is off the result set.
- Throws exception if result type is `TYPE_FORWARD_ONLY`.

## Example Code: ResultSetEx previous, next & getters methods

```
import java.sql.*;

public class ResultSetEx {

    public static void main ( String args[] ) {

        try {
            // load driver & make connection

            String sql = "SELECT * FROM Person";

            PreparedStatement pstmt = con.prepareStatement( sql,
                ResultSet.TYPE_SCROLL_INSENSITIVE ,
                ResultSet.CONCUR_UPDATABLE );

            ResultSet rs = pstmt.executeQuery( );
```

## Example Code: ResultSetEx previous, next & getters methods

```
rs.next();

System.out.println("moving cursor forward");
String name = rs.getString("name");
System.out.println(name);

rs.next();

rs.previous();

System.out.println("moving cursor backward");
name = rs.getString("name");
System.out.println(name);
```

## Example Code: ResultSetEx previous, next & getters methods

```
con.close();

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

} // end main

} //end class
```



## Compile & Execute



## ResultSet (cont.)

- **Useful Methods**
  - **absolute(int)**
    - Move the cursor to the given row number in the `ResultSet` object
    - If the row number is positive, moves the cursor forward with respect to beginning of the result set.
    - If the given row number is negative, the cursor moves to the absolute row position with respect to the end of the result set.
      - For example, calling `absolute(-1)` positions the cursor on the last row; calling `absolute(-2)` moves the cursor to next-to-last row, and so on.
    - Throws exception if result type is `TYPE_FORWARD_ONLY`.

## ResultSet (cont.)

- **Useful Methods**
  - **updaters** (for primitives, Object & String)
    - Used to update column values in the current row or the *insert row*.
    - Do not update the underlying database
    - Each update method is overloaded.
    - For example of String

```

updateString(String columnName, String value)
updateString(int columnIndex, String value)
          
```

## ResultSet (cont.)

- **Useful Methods**
  - **updateRow()**
    - Updates the underlying database with new contents of the current row of this `ResultSet` object.

## Modify Example : ResultSetEx updating existing rows

```

import java.sql.*;
public class ResultSetEx {
    ..... // main method
    ..... Load driver, make connection
    ..... Make updatable resultset

    //move cursor to 2nd row of rs
    rs.absolute(2);

    //update address column of 2nd row in rs
    rs.updateString("address", "model town");

    //update the row in database
    rs.updateRow();

    ..... // close connection etc

    .... //end main
} // end class
  
```

## Compile & Execute

Before execution

| id | name  | address    | phoneNum |
|----|-------|------------|----------|
| 1  | ali   | model town | 9203256  |
| 2  | usman | defence    | 8219065  |
| 3  | raza  | defence    | 5173946  |

Records: 1 2 3 of 3

After execution

| id | name  | address    | phoneNum |
|----|-------|------------|----------|
| 1  | ali   | model town | 9203256  |
| 2  | usman | model town | 8219065  |
| 3  | raza  | defence    | 5173946  |

Records: 1 2 3 of 3

## ResultSet (cont.)

### • Useful Methods

#### – moveToInsertRow()

- An Updatable `ResultSet` object has a special row associated with it i.e. insert row.
- Insert row – a buffer, where a new row may be constructed by calling the updater methods
- Doesn't insert row into a result set or into a database

## ResultSet (cont.)

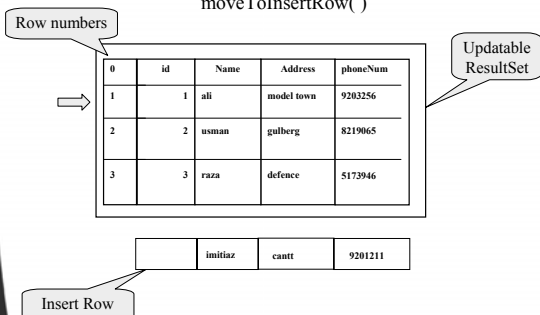
### • Useful Methods

#### – insertRow()

- Inserts the contents of the insert row into this `ResultSet` object and into the database.
- The cursor must be on the insert row when this method is called

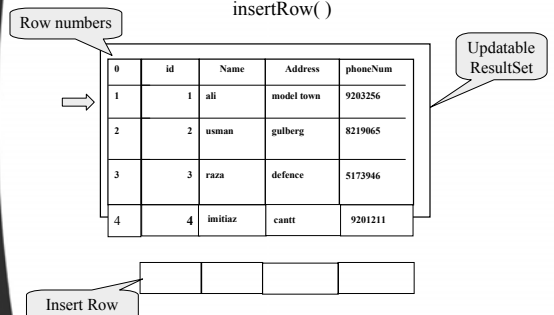
## ResultSet (cont.)

moveToInsertRow()



## ResultSet (cont.)

insertRow()



## Modify Example : ResultSetEx Inserting new row

```

import java.sql.*;
public class ResultSetEx {
    ..... // main method
    ..... Load driver, make connection
    ..... Make updatable resultset

    //move cursor to insert row
    rs.moveToInsertRow();

    // updating values into insert row
    rs.updateString("name", "imitiaz");
    rs.updateString("address", "cantt");
    rs.updateString("phoneNum", "9201211");

    //insert row into rs & db
    rs.insertRow();

    .....

    ..... //end main
} // end class

```

## Compile & Execute

Before execution

| id | name  | address    | phoneNum |
|----|-------|------------|----------|
| 1  | ali   | model town | 9203256  |
| 2  | usman | gulberg    | 8219065  |
| 3  | raza  | defence    | 5173946  |
| 0  |       |            |          |

After execution

| id | name    | address    | phoneNum |
|----|---------|------------|----------|
| 1  | ali     | model town | 9203256  |
| 2  | muanwar | gulberg    | 8219065  |
| 3  | raza    | defence    | 5173946  |
| 4  | imitiaz | cantt      | 9201211  |

## ResultSet (cont.)

### • Useful Methods

#### – last() & first()

- Moves the cursor to the last & first row of the ResultSet object respectively.
- Throws exception if the result set is TYPE\_FORWARD\_ONLY

#### – getRow()

- Returns the current row number
- The first row number is 1, second row number is 2 and so on

## ResultSet (cont.)

### • Useful Methods

#### – deleteRow()

- Deletes the current row from this ResultSet object and from the underlying database.
- Throws exception when the cursor is on the insert row

## Modify Example : ResultSetEx deleting existing row

```

import java.sql.*;
public class ResultSetEx {
    .... // main method
    .... Load driver, make connection
    .... Make updatable resultset

    //moves to last row
    rs.last();

    int rNo = rs.getRow();
    System.out.println("curr row no: "+ rNo );

    //delete current row (4) from rs & db
    rs.deleteRow();

    ....

    .... //end main
} // end class

```

## Compile & Execute

Before execution

| id | name    | address    | phoneNum |
|----|---------|------------|----------|
| 1  | ali     | model town | 9203256  |
| 2  | usman   | gulberg    | 8219065  |
| 3  | raza    | defence    | 5173946  |
| 4  | imtiyaz | cantt      | 9201211  |

After execution

| id | name  | address    | phoneNum |
|----|-------|------------|----------|
| 1  | ali   | model town | 9203256  |
| 2  | usman | gulberg    | 8219065  |
| 3  | raza  | defence    | 5173946  |
| 0  |       |            |          |



## Meta Data

## Meta Data

| id | name    | address | phoneNum |
|----|---------|---------|----------|
| 1  | ali     | new     | 9203256  |
| 2  | usman   | gulberg | 8219065  |
| 3  | raza    | defence | 5173946  |
| 4  | imtiyaz | cantt   | 9201211  |

## Meta Data

- **What if you want to know:**
  - How many columns are in the result set?
  - What is the name of a given column?
  - Are the column names case sensitive?
  - What is the data type of a specific column?
  - What is the maximum character size of a column?
  - Can you search on a given column?

## Using ResultSetMetaData

- **Idea**
  - From a `ResultSet` (the return type of `executeQuery`), derive a `ResultSetMetaData` object
  - Use that object to look up the number, names, and types of columns

## Useful ResultSetMetaData Methods

- **getColumnCount ( )**
  - Returns the number of columns in the result set
- **getColumnDisplaySize (int)**
  - Returns the maximum width of the specified column in characters
- **getColumnName(int) / getColumnLabel (int)**
  - The `getColumnName` method returns the database name of the column
  - The `getColumnLabel` method returns the suggested column label for printouts
- **getColumnType (int)**
  - Returns the SQL type for the column to compare against types in `java.sql.Types`

## Example Code: MetadataEx using ResultSetMetaData

```
import java.sql.*;

public class MetadataEx {

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

            Class.forName("Driver name");

            Connection con = DriverManager.getConnection(url, usr, pwd);

            String sql = "SELECT * FROM Person";
            PreparedStatement pstmt = con.prepareStatement(sql);

            ResultSet rs = pstmt.executeQuery( );
```

## Example Code: MetadataEx (cont.) using ResultSetMetaData

```
ResultSetMetaData rsmd = rs.getMetaData();

int numColumns = rsmd.getColumnCount();
System.out.println("Number of Columns:" + numColumns);

String cName;
for (int i=1; i <= numColumns; i++)
{
    cName = rsmd.getColumnLabel(i);
    System.out.print(cName);
    System.out.print("\t");
}

// changing line
System.out.println("");
```

## Example Code: MetadataEx (cont.) using ResultSetMetaData

```
String id, name, add, ph;
while (rs.next())
{
    id = rs.getString(1);
    name = rs.getString(2);
    add = rs.getString(3);
    ph = rs.getString(4);

    System.out.print(id);
    System.out.print("\t");
    System.out.print(name);
    System.out.print("\t");
    System.out.print(add);
    System.out.print("\t");
    System.out.print(ph);
    System.out.print("\n");
}
```

### Example Code: MetadataEx (cont.) using ResultSetMetaData

```
con.close();

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

}

// end main
//end class
```

### Example Code: MetadataEx Compile & Execute

Person : Table

| id | name    | address | phoneNum |
|----|---------|---------|----------|
| 1  | ali     | new     | 9203256  |
| 2  | usman   | gulberg | 8219065  |
| 3  | raza    | defence | 5173946  |
| 4  | imitiaz | cantt   | 9201211  |

Record: 1 of 4

```
C:\WINDOWS\system32\cmd.exe
D:\examples\jdbc>javac MetadataEx.java
D:\examples\jdbc>java MetadataEx
Number of Columns:4
id      name      address  phoneNum
1       ali       new      9203256
2       usman     gulberg  8219065
3       raza      defence  5173946
4       imitiaz   cantt    9201211
```

## DatabaseMetaData

- **What if we want to know**
  - What SQL types are supported by DBMS to create table?
  - What is the name of a database product?
  - What is the version number of this database product?
  - What is the name of the JDBC driver that is used?
  - Is the database in a read-only mode?

## Using DatabaseMetaData

- **Idea**
  - From a Connection, derive a DatabaseMetaData object
  - Contains the comprehensive information about the database as a whole

## Using DatabaseMetaData

- **Idea**
  - From a Connection, derive a DatabaseMetaData object
  - Contains the comprehensive information about the database as a whole

## Useful DataBaseMetaData Methods

- **getDatabaseProductName ( )**
  - Returns the name of the database product name
- **getDatabaseProductVersion ( )**
  - Returns the version number of this database product
- **getDriverName( )**
  - Returns the name of the JDBC driver used to established the connection
- **isReadOnly ( )**
  - Retrieves whether this database is in read-only mode.
  - Returns true if so, false otherwise.

### Example Code: Modify MetadataEx using DataBaseMetaData

```
import java.sql.*;

public class MetadataEx {

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

            Class.forName("Driver name");

            Connection con = DriverManager.getConnection(url, usr, pwd);

            DatabaseMetaData dbMetaddata = con.getMetaData();

            .....

        }
    }
}
```

JDBC

### Example Code: Modify MetadataEx using DataBaseMetaData

```
String pName = dbMetaData.getDatabaseProductName();
System.out.println("Database: " + pName);

String pVer = dbMetaData.getDatabaseProductVersion();
System.out.println("Version: " + pVer);

String dName = dbMetaData.getDriverName();
System.out.println("Driver: " + dName);

boolean rOnly = dbMetaData.isReadOnly();
System.out.println("Read-Only: " + rOnly);

.....
```

JDBC

### Example Code: Modify MetadataEx using DataBaseMetaData

```
// create Statement & execute query

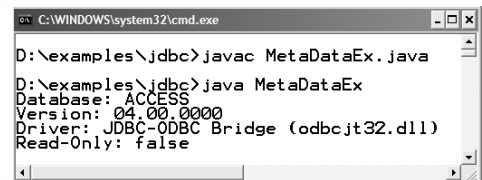
// process results

con.close();

}catch ( Exception ex) {
    System.out.println(ex);
}
} // end main
} // end class
```

JDBC

### Example Code: Modify MetadataEx Compile & Execute



```
C:\WINDOWS\system32\cmd.exe

D:\examples\jdbc>javac MetadataEx.java

D:\examples\jdbc>java MetadataEx
Database: ACCESS
Version: 04.00.0000
Driver: JDBC-ODBC Bridge (odbcjt32.dll)
Read-Only: false
```

JDBC



## RowSet

## RowSet

- A JDBC RowSet object holds tabular data in a way that makes it more flexible and easier to use than a result set.
- Interface RowSet configures the database connection and prepares query statements automatically.
- It is part of package javax.sql.
- It is part of J2SE, but it is normally used in the context of J2EE.

JDBC

## RowSet (cont.)

There are two kinds of RowSet objects:

- **Connected**
  - Makes the connection to the database and stays connected until the application ends
- **Disconnected**
  - Connects, queries the database, then closes.
  - Connection can be reestablished for updates.

JDBC

## RowSet (cont.)

JDBC provides the five versions of the RowSets. Two of them are:

### 1. JdbcRowSet

- Connected RowSet that wraps a ResultSet object, allowing scrolling and updating.
- It is most similar to a ResultSet object.

JDBC

## RowSet (cont.)

### 2. CachedRowSet

- Disconnected RowSet that is scrollable and updateable.
- It caches the data of a ResultSet in memory.
- Manipulate data and make changes to data while it is disconnected.
- Reconnect to the data source to write changes back to it.
- It is also serializable, so it can be sent across a network.

JDBC



## JDBC Drivers Types

## JDBC Driver Types

- **JDBC drivers are divided into four types or levels.**
- **Each type defines a JDBC driver implementation with increasingly higher level of platform independence, performance, deployment and administration.**
- **The four types are:**
  1. Type 1: JDBC – ODBC Bridge
  2. Type 2: Native – API/partly Java driver
  3. Type 3: Net – protocol/all-Java driver
  4. Type 4: Native – protocol/all-Java driver

JDBC

## JDBC Driver Types (cont.)

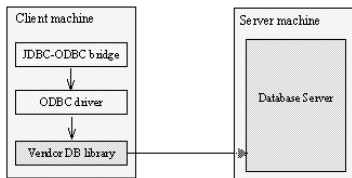
### 1. Type 1: JDBC – ODBC Bridge

- Translates all JDBC calls into ODBC (Open Database Connectivity) calls and send them to the ODBC driver.
- Generally used for Microsoft databases.
- Performance is degraded

JDBC

## JDBC Driver Types (cont.)

### 1. Type 1: JDBC – ODBC Bridge



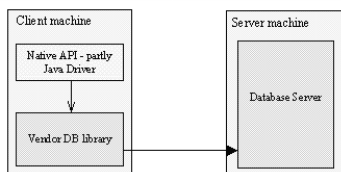
## JDBC Driver Types (cont.)

### 2. Type 2: Native – API/partly Java driver

- Converts JDBC calls into database-specific calls such as SQL Server, Informix, Oracle or Sybase.
- Partly-Java drivers communicate with database-specific API (which may be in C/C++) using the Java Native Interface.
- Significantly better Performance than the JDBC-ODBC bridge.

## JDBC Driver Types (cont.)

### 2. Type 2: Native – API/partly Java driver



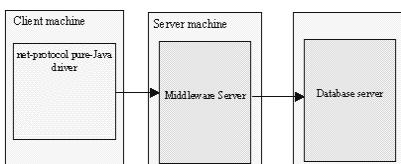
## JDBC Driver Types (cont.)

### 3. Type 3: Net – protocol/all-Java driver

- Follows a three-tiered approach whereby the JDBC database requests ()are passed through the network to the middle-tier server
- Pure Java client to server drivers which send requests that are not database-specific to a server that translates them into a database-specific protocol.
- If the middle-tier server is written in java, it can use a type 1 or type 2/JDBC driver to do this
- No need for any vendor database library to be present on client machines because it is server-based

## JDBC Driver Types (cont.)

### 3. Type 3: Net – protocol/all-Java driver



## JDBC Driver Types (cont.)

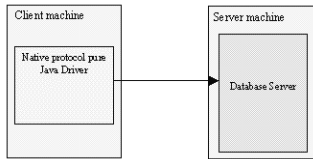
### 4. Type 4: Native – protocol/all-Java driver

- Converts JDBC calls into the vendor-specific DBMS protocol so that client application can communicate directly with the database server.
- Completely implemented in Java to achieve platform independence and eliminate deployment issues.
- Performance is typically very good

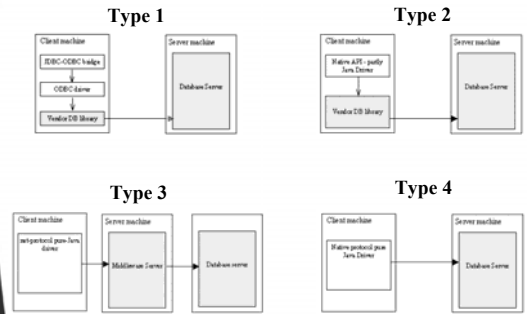


## JDBC Driver Types (cont.)

### 4. Type 4: Native – protocol/all-Java driver



## Summary of Driver Types



## Looking Insight

- JDBC is mostly collection of interfaces.
- Connection, Statement, PreparedStatement, ResultSet and RowSet are all interfaces.
- Why?
  - Any DBMS interested in providing support for java connectivity, need to provide implementation of all the above interfaces.

## General Design Guideline

```
class Employee {
    String name;
    String sal;

    // constructor
    // getter / setters

    void insertEmp() {
        // connect database
        // execute query
        // process results
    }

    void retrieveEmp() { ..... }

    void calculateTax() { ..... }
}
```

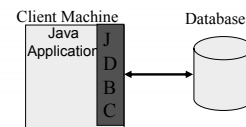
Database connectivity & business logic  
all in one class

## General Design Guideline

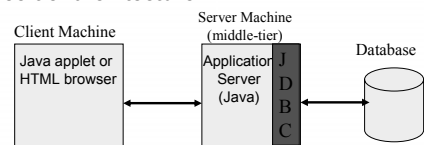
| Business Logic                                                                                                                                                                                                                                         | Database Connectivity                                                                                                                                                                                                                                                                                                                             |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>public class Employee {     String name;     String address;      // constructor     // getter / setters      void insertEmp() {         DAO dao = new DAO();         dao.update(name, address);     }      void calculateTax() { ..... } }</pre> | <pre>//step 1 public class DAO {     public DAO() {         //step 2 - 5         .....     }      public void insert (String n, String s)     {         // make statement         // execute query + process results     }      public void insert(....){.....}     protected void finalize() {         //step 8 - close connection     } }</pre> |

## Architectures

### Two-tier architecture:



### Three-tier architecture:



## On-line Resources

- **Sun's JDBC Site**
  - <http://java.sun.com/products/jdbc/>
- **JDBC Tutorial**
  - <http://java.sun.com/docs/books/tutorial/jdbc/>
- **List of Available JDBC Drivers**
  - <http://industry.java.sun.com/products/jdbc/drivers/>
- **API for java.sql**
  - <http://java.sun.com/j2se/1.3/docs/api/java/sql/package-summary.html>