JDBC

Java Database Connectivity

JDBC API

- JDBC 4.0 (part of JSE 6) is an API that provides standard for connectivity to variety of data sources like SQL databases, spreadsheets and flat.

- Therefore before writing JDBC code we must make sure that we have the library with respect to the data source that we intend to use.
- JDBC is based on the X/Open SQL Call Level Interface (CLI). JDBC 4.0 complies with the SQL 2003 standard.

Steps to write database code

1. Load the driver
2. Obtain connection
3. Create and execute statements
4. [Use result sets to navigate the results]
5. Close the connection

Load the driver

- **`java.sql.DriverManager`** class is used to get drivers and get connection to the database.
- To register (Load) the driver with the application explicitly:

```
Class.forName("<driver class name>");
```

Or

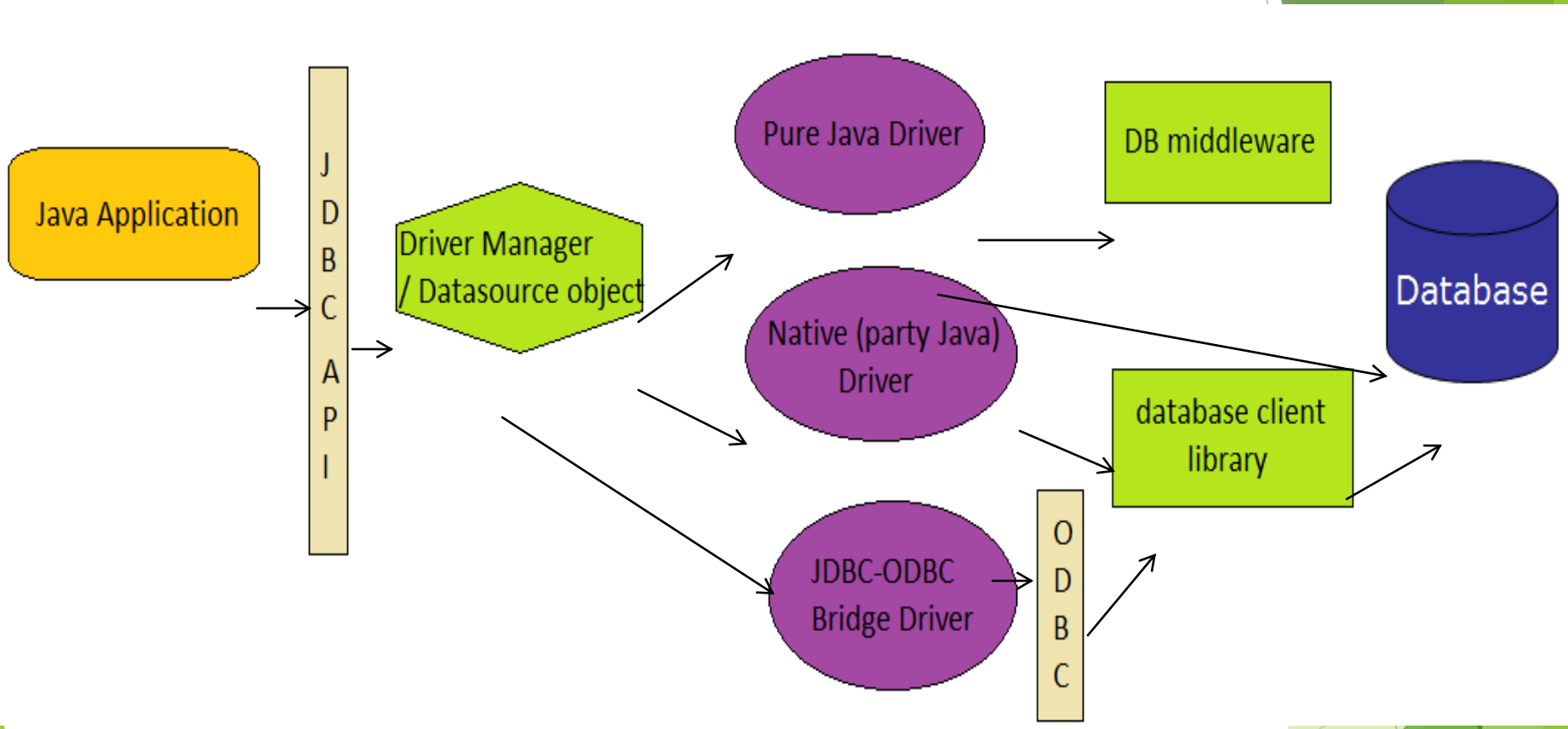
```
DriverManager.registerDriver( new <driver class name>());
```

Both of these registers the given driver with the **`DriverManager`**.. Static block of the <Driver class> calls **`DriverManager.registerDriver()`** method!

- With JDBC 4.0, this class also provides a mechanism to automatically load the database drivers (provided they are packaged in the specified way) → later

Get Connected

- Getting Connection:
  - **`static Connection getConnection(String url) throws SQLException`**
  - **`static Connection getConnection(String url, Properties info) throws SQLException`**
  - **`static Connection getConnection(String url, String user, String password) throws SQLException`**

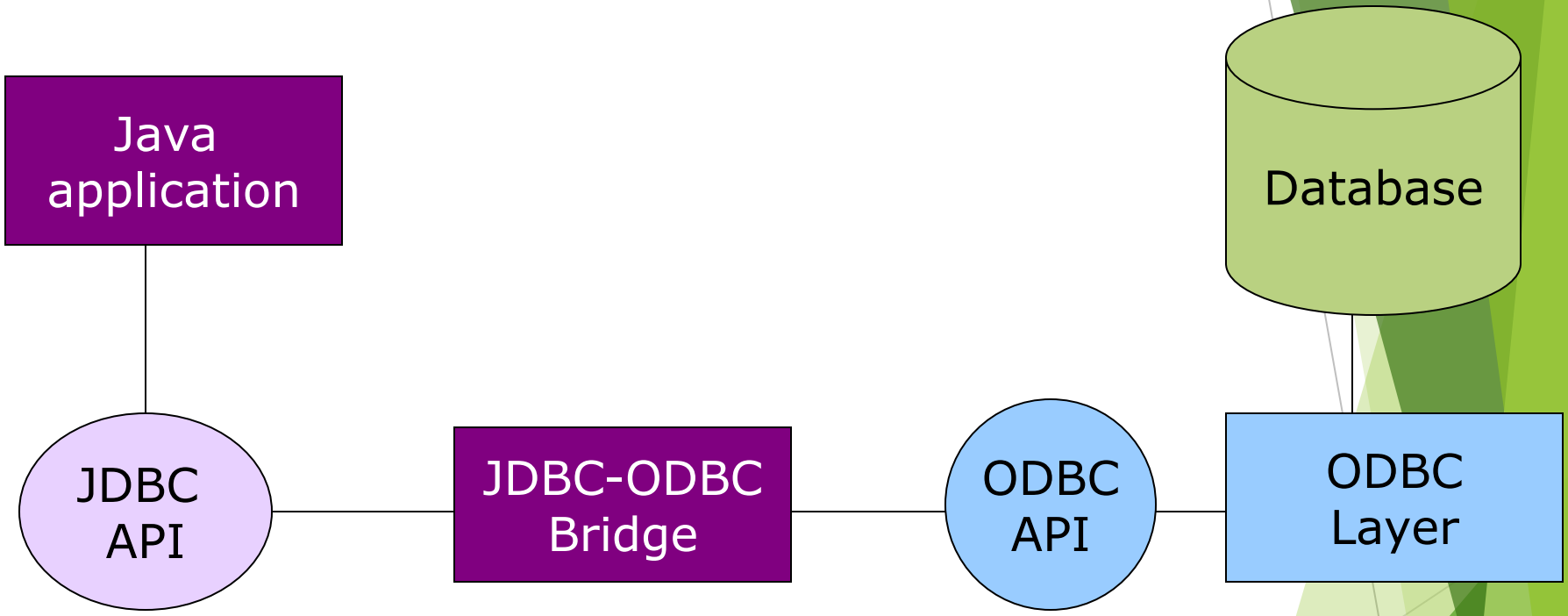


4 Ways to connect to database – through 4 types of driver

Driver Types

- JDBC driver are classes used to translate JDBC calls either to vendor-specific database calls or may be directly invoke database commands.
- Types of driver
  - ✓ Type 1- JDBC-ODBC Bridge
  - ✓ Type 2- Part Java, Part Native Driver
  - ✓ Type 3- Intermediate Database Access Server
  - ✓ Type 4- Pure Java Drivers

JDBC-ODBC Bridge



Drivers classes that implement the JDBC API as a mapping to ODBC (Open Database Connectivity) API.

ODBC Microsoft's interface for accessing data in a heterogeneous database management systems.

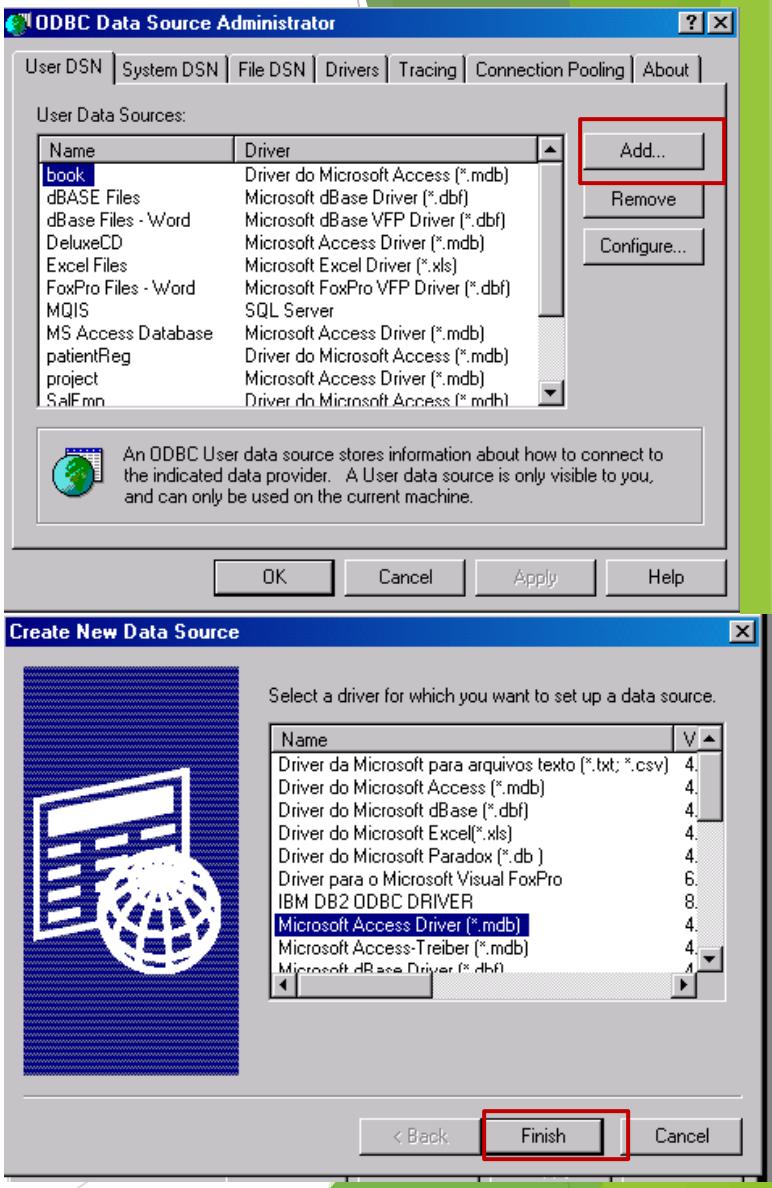
ODBC binary code and in many cases, database client code -- must be loaded on each client machine that uses a JDBC-ODBC Bridge

Example using JDBC-ODBC Bridge

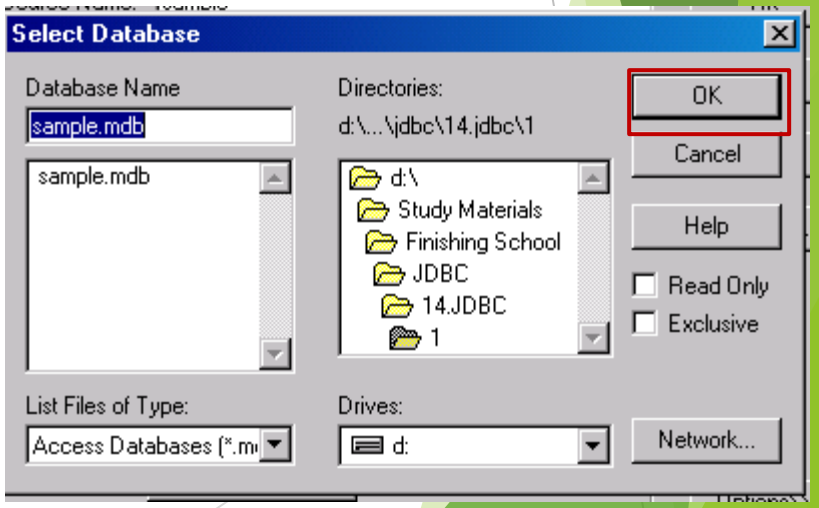
- We will write code to establish connection with JDBC-ODBC Bridge.
- Before we write code there are 2 things we need to do
  1. Create a database in MS-Access with any name.
  2. Configure the DSN for ODBC Driver as specified in the next slides

Configure the DSN for ODBC Driver for MS-Access

- In control panel locate “Data Sources (ODBC)” icon (inside Administrative Tools)
- Double-click the icon and click on “Add” button in the “User DSN” tab.
- Select “Microsoft Access Driver” and click “Finish”



- Enter DSN name as “sample” (we will use this name in the code) and click “Select”.
- Browse through and get the access file and click “OK”.
- Click “OK” on next two screens and come out of the control panel.



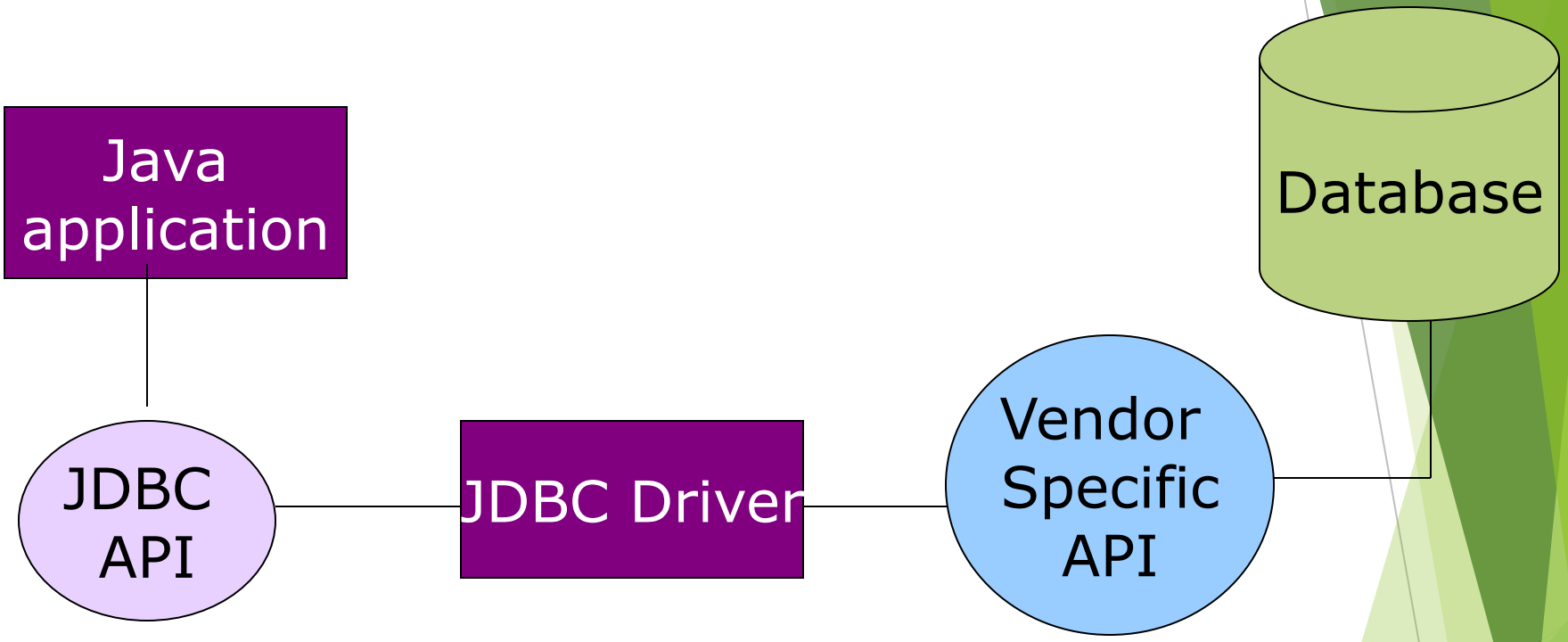
Code to get JDBC-ODBC connection

```
import java.sql.*;
public class ODBCMain{
  Connection con;
  public ODBCMain()  {
    try    {
   //sample is data source name (DSN)
      String url = "jdbc:odbc:sample";
   //Load the JDBC-ODBC bridge driver
   Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
   //Obtain the connection to the database
   con = DriverManager.getConnection(url);
 System.out.println ("JDBC-ODBC connection established");
   con.close();
}
catch(Exception e){
  e.printStackTrace();
} }
public static void main(String str[]){new ODBCMain();}}
```

Disadvantages

- ODBC uses a C. Calls from Java to native C code have a number of drawbacks in the security, robustness, and automatic portability of applications.
- Multiple layers of indirection leads to inefficiency.

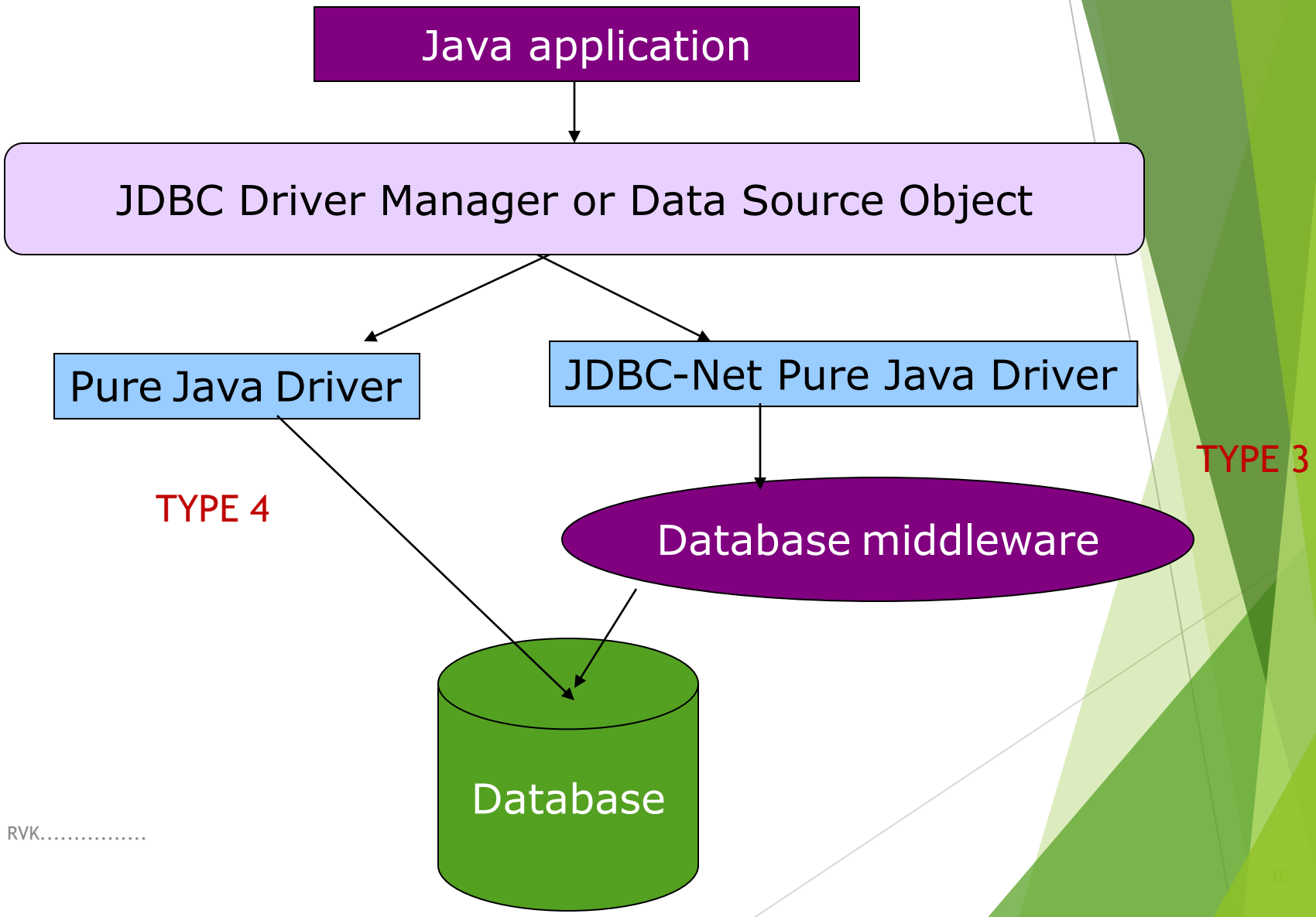
Type2- Part Java, Part Native Driver



Much like the bridge driver, this style of driver also requires that some binary code be loaded on each client machine. Therefore this also shares the same disadvantages. This driver is very rarely now a days.

Preferred Database Drivers

- Type 3 and Type 4 drivers are preferred database drivers for Java.
- Each of these drivers are used in different architectural situations.
- Type 4 Drivers
  - Used in 2-tier architecture
  - Direct to database connection from java application
  - Pure Java Driver
- Type 3
  - Three-tier
  - Connection to database happens through a middleware. The middleware provides connectivity to many different databases.
  - JDBC-Net pure Java Driver

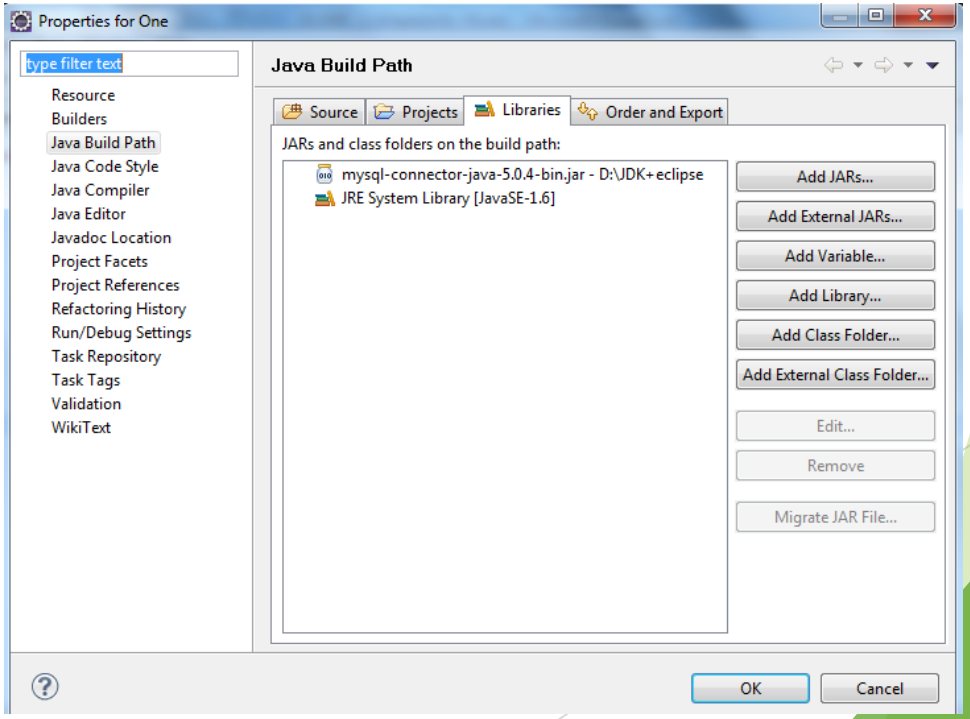

Java application
JDBC Driver Manager or Data Source Object
Pure Java Driver
JDBC-Net Pure Java Driver
TYPE 4
TYPE 3
Database middleware
Database

Connecting using type 4 driver

- There are two ways in which we can write this code to connect to a database using Type 4 driver
  1. By specifying class explicitly in the code using **`Class.forName()`** **`or DriverManager.registerDriver()`**.
  2. By using JDBC 4.0 way where we can avoid hard coding the class name in the code.
- MySQL database using type 4 driver name is **`com.mysql.jdbc.Driver`**.
- But before we write code to connect to MySQL databse
  1. we must have database. Create a database called **`test`** in MySQL.
  2. set the classpath to the jar file with respect to MySQL in the classpath. Download Connector/J driver for **`mysql-connector-java-5.0.4-bin.jar`**.

Setting classpath to MySQL driver in eclipse

- Select your project, Go to Project menu and select Properties.
- Move to “Java Build Path” on the left panel. Click on “Add External JARs”, browse and locate the jar file for MySQL.
- Click on Open. This will add the jar file to the Properties Box as shown in the diagram.



Note:

For Web Applications jdbc jar files to be saved in lib directory of the project.

Example: Connecting to MySQL - older way

```
import java.sql.*;
import java.util.Properties;

public class Connect   {
  public static void main (String[] args) {
  Connection conn = null;
  try {
   String userName = "root";
   String password = "root";
   String url = "jdbc:mysql://localhost/test";
   Class.forName ("com.mysql.jdbc.Driver");
   Properties connectionProps = new Properties();
   connectionProps.put("user", userName);
      connectionProps.put("password",password);
   conn =
   DriverManager.getConnection(url,connectionProps);
 System.out.println ("Database connection
successful");}
```

```
    catch (SQLException e){
      System.err.println ("Failed to connect to
database" +e);
       }
   finally{
                if (conn != null)     {
                try     {     conn.close ();}
                  catch (SQLException e) { }
                }
            }
        }
   }
```

Result:

Database connection successful

Example: Connecting using JDBC 4.0 way

- With JDBC 4.0 the Java service provider mechanism, applications no longer need to explicitly load JDBC drivers using `Class.forName()`.
- When the `getConnection()` is called, the `DriverManager` will attempt to locate a suitable driver from amongst specified in the classpath. (If there are more than one, the first one is used.)
- The code is fundamentally same as the previous example- only line eliminated is `Class.forName ("com.mysql.jdbc.Driver");`

```
import java.sql.*;
import java.util.Properties;
  public class Connect   {
   public static void main (String[] args)        {
   Connection conn = null;
   try {
      String userName = "root";
      String password = "root";
      String url = "jdbc:mysql://localhost/test";
```

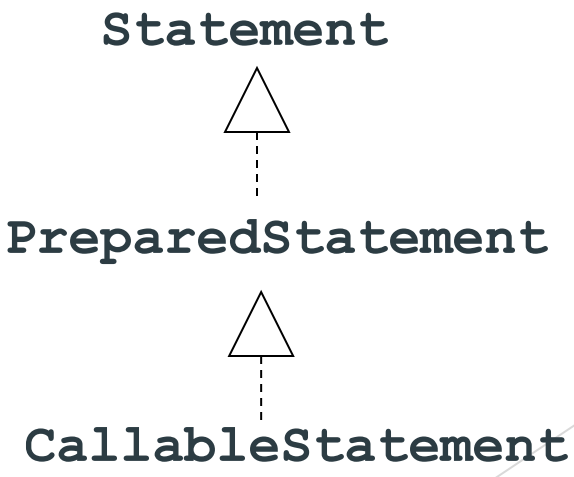
```
Properties connectionProps = new Properties();
connectionProps.put("user", userName);
connectionProps.put("password",password);
conn = DriverManager.getConnection (url,connectionProps);

System.out.println ("Database connection successful");
}
catch (SQLException e) {
System.err.println ("Failed to connect to database" +e);
}
finally
      {
       if (conn != null)     {
       try     {     conn.close ();}
             catch (SQLException e) { }
                }
            }
}    }
```

java.sql.Connection

- **`Connection`** is an interface.
- **`getConnection()`** method of **`DriverManager`** returns the object that implements this interface.
- Connection class can be used to do a variety of tasks
  - Obtain **`Statement`** object
  - To work with transactions
  - To get meta-data about the database
- For executing a static SQL statement and returning the results it produces, an object that implements **`Statement`** interface is used.

Obtaining Statement

- **Connection** class methods to obtain **Statement** and its subclasses:
  - **Statement createStatement() throws SQLException**
  - **PreparedStatement prepareStatement(String sql) throws SQLException**
  - **CallableStatement prepareCall(String sql) throws SQLException**



Statement

- The object of this type is used for executing a static SQL statement and returning the results it produces.
- The result of the query statement is returned in the form of **`ResultSet`** object .
- Only one **`ResultSet`** object per **`Statement`** object can be open at the same time.
- Therefore, reading data 2 or more **`ResultSet`** objects would require obtaining **`ResultSets`** from different Statement objects.
- On calling execute methods on **`Statement`** object, the **`ResultSet`** object obtained from this **`Statement`** object (if there are any) is automatically closed.

Statement members

- **`ResultSet executeQuery(String sql) throws SQLException`**
  - sql is typically a static SQL SELECT statement. a ResultSet object that contains the data produced by the given query; never null

- **`int executeUpdate(String sql) throws SQLException`**
  - sql is typically a SQL DML (INSERT, UPDATE, or DELETE statement) or DDL like (CREATE, DROP statements)
  - Returns the row count as number of rows affected for DML statements or 0 for SQL DDL statement.

- **`boolean execute(String sql) throws SQLException`**
  - Used to execute stored procedure
  - Used to execute sql that many return multiple result set or update counts
  - To get the results - methods **`getResultSet`** or **`getUpdateCount`** to retrieve the first result, and **`getMoreResults`** to move to any subsequent result(s).
    - **`ResultSet getResultSet() throws SQLException`**
    - **`int getUpdateCount() throws SQLException`**
    - **`boolean getMoreResults() throws SQLException`**

- Batch execution methods: Multiple updates can be sent to the database for execution as a batch.
  - **`void addBatch(String sql) throws SQLException`**

    used to add the given SQL command to be executed as batch.
  - **`int[] executeBatch() throws SQLException`**

    submits a batch of commands to the database for execution and if all commands execute successfully, returns an array of update counts

`ResultSet` methods (default)

- **`ResultSet`** is an interface representing table of data that is retrieved from a database
- It maintains a cursor initially pointing to the first row. The **`next`** method moves the cursor to the next row until returns false when there are no rows to read.
- A default **`ResultSet`** object is not updatable and has a cursor that moves forward only.

```
boolean next() throws SQLException
void close() throws SQLException
XXX getXXX(int columnIndex)
            throws SQLException
```

where XXX is any primitive type, String, java.sql.Date (subclass of **java.util.Date**), **Object**

columnIndex begins from 1

Example :Code to insert and fetch records

Assuming a table in MYSQL named **Student** with **regNo(int(11)), name(varchar(45)), degree(name(varchar(45)) and semester((int(11))**

```
import java.sql.*;
import java.util.Properties;
  public class Connect   {
   public static void main (String[] args) {
   Connection conn = null;
       try
           {
      String userName = "root";
      String password = "root";
      String url = "jdbc:mysql://localhost/test";
      Properties props = new Properties();
      props.put("user", userName);
            props.put("password",password);
            conn = DriverManager.getConnection(url,props);
      Statement s= conn.createStatement();
```

```
s.executeUpdate("INSERT INTO STUDENT VALUES
(1,'Rama','M.C.A.', 1)");
s.executeUpdate("INSERT INTO STUDENT VALUES
(2,'Sita','B.Tech', 2)");
ResultSet rs=s.executeQuery("SELECT * FROM STUDENT");
System.out.println("ID     Name     Degree
Semester");
while (rs.next() ) {
System.out.println( rs.getInt(1) +"
"+rs.getString(2)+"      "+rs.getString(3)+"
"+rs.getInt(4));  }
} catch (SQLException e){
System.err.println ("Failed to connect to database" +e);
}
finally  {
if (conn != null) {
     try    {  conn.close ();}catch (SQLException e) { }
} } }  }
```

Advanced `ResultSet`

- A default **`ResultSet`** object is not updatable and has a cursor that moves forward only.
- In order to have **`ResultSet`** which are scrollable and updatable a different **`createStatement()`** method has to be used.
- **`public Statement createStatement(int resSetType,int resSetConcurrency) throws SQLException`**
- **`resSetType :`**

  `ResultSet.TYPE_FORWARD_ONLY`

  cursor may move only forward (default)

  `ResultSet.TYPE_SCROLL_INSENSITIVE`

  scrollable but not sensitive to changes to the underlying data in the database that happens outside the purview of this object

  `ResultSet.TYPE_SCROLL_SENSITIVE`

  scrollable but not sensitive to changes to the underlying data

- **resSetConcurrency :**

  ResultSet.CONCUR_READ_ONLY

  makes the result set read only

  ResultSet.CONCUR_UPDATABLE

  makes the result set updateable. Using this object, rows can be inserted, updated and deleted in the object itself which automatically synchronizes with the database.

More `ResultSet` methods

- `void afterLast() throws SQLException`
- `void beforeFirst() throws SQLException`
- `boolean first() throws SQLException`
- `boolean last() throws SQLException`
- `boolean isAfterLast()`
- `boolean isBeforeFirst()`
- `boolean isFirst()`
- `boolean isLast()`
- `boolean rowUpdated() throws SQLException`
- `boolean rowInserted() throws SQLException`
- `boolean rowDeleted() throws SQLException`

- **`void updateXXX(int columnIndex, byte x) throws SQLException`**

Where **`XXX`** is primitives or **`String, java.sql.Date, Object`**

- **`void insertRow() throws SQLException`**
- **`void updateRow() throws SQLException`**
- **`void deleteRow() throws SQLException`**
- **`void refreshRow() throws SQLException`**
- **`void moveToInsertRow() throws SQLException`**

How these methods work is better understood by an example ahead.

Example: using advanced ResultSet

```
import java.sql.*;
import java.util.Properties;
public class AdvRS {
 public static void main (String[] args)          {
Connection conn = null;
try
    {
   String userName = "root";
   String password = "root";
   String url = "jdbc:mysql://localhost/test";
   Properties props = new Properties();
      props.put("user", userName);
      props.put("password",password);
      conn = DriverManager.getConnection (url,props);

Statement stmt =
conn.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
ResultSet.CONCUR_UPDATABLE);
```

```
ResultSet rs = stmt.executeQuery("SELECT * FROM
STUDENT");
System.out.println("Before...");
System.out.println("ID     Name     Degree
Semester");
while (rs.next() ) {
System.out.println( rs.getInt(1) +"
"+rs.getString(2)+"      "+rs.getString(3)+"
"+rs.getInt(4));
}
//inserting a new row
   rs.moveToInsertRow();
   rs.updateInt("RegNo", 3);
   rs.updateString("name", "Geeta");
   rs.updateString("degree", "B.E.");
   rs.updateInt("semester", 3);
   rs.insertRow();
```

```
//updating 2nd row - changing name to Seetha
    rs.absolute(2);
       rs.updateString(2,"Seetha");
    rs.updateRow();

 rs.beforeFirst();
System.out.println("After...");
System.out.println("ID      Name       Degree
Semester");
while (rs.next() ) {
System.out.println( rs.getInt(1) +"
"+rs.getString(2)+"        "+rs.getString(3)+"
"+rs.getInt(4));  }}
catch (SQLException e) {  System.err.println ("Failed to
connect to database" +e);
}
finally {if (conn != null)     {
try {conn.close ();} catch (SQLException e) { }          }
}      } }
```

Result of execution of the code

```
Before...
ID      Name      Degree      Semester
1       Rama       M.C.A.      1
2       Sita       B.Tech      2
After...
ID      Name      Degree      Semester
1       Rama       M.C.A.      1
2       Seetha        B.Tech      2
3       Geeta       B.E.      3
```

PreparedStatement

- Interface that inherits from **`Statement`**
- If the same sql statement is executed many times it is more efficient to use a prepared statement.
- It enables a SQL statement to contain parameters like functions. So same statement can be executed for different set of values.

Example: Inserting large objects in the database

- Example demonstrating insertion of a picture in a database. Assuming that we have table Photo(id integer(110, name varchar(45), pho blob).

```
import java.sql.*;
import java.io.*;

public class InsertPhoto{
public static void main(String[] args) {
System.out.println("Insert Image Example!");
String driverName = "com.mysql.jdbc.Driver";
 String userName = "root";
          String password = "root";
          String url = "jdbc:mysql://localhost/test";
Connection con = null;
try{
Class.forName(driverName);
con = DriverManager.getConnection(url,userName,password);
```

```
File imgfile = new File("D:\\image.jpg");
FileInputStream fin = new FileInputStream(imgfile);
PreparedStatement pre = con.prepareStatement("insert
into Photo values(?,?,?)");
pre.setInt(1,5);
pre.setString(2,"Durga");
pre.setBinaryStream(3,fin,(int)imgfile.length());
pre.executeUpdate();
System.out.println("Inserting Successfully!");
pre.close();
con.close();
}
catch (Exception e){
System.out.println(e.getMessage());}
}
}
```

Using `PreparedStatement` for batch updates

- Apart from the batch related methods added by the **`Statement`** interface, **`PreparedStatement`** adds one more
- **`void addBatch() throws SQLException`**

  is used to add batch of commands
- But since most of the batch operations are executed using same sql statement, it is usually used with **`PreparedStatement`** since it allows parameterized statements and hence they can be cached to give better performance.

Example: Batch updates

```
import java.sql.*;
public class BatchEx {
 public static void main (String[] args)  throws
SQLException     {
         Connection conn = null;
             String userName = "root";
             String password = "root";
             String url ="jdbc:mysql://localhost/test";
java.util.Calendar  c=java.util.Calendar.getInstance();
   conn =
DriverManager.getConnection(url,userName,password);
PreparedStatement stmt = conn.prepareStatement("INSERT
INTO BookIssue VALUES(?,?,?,?)");
stmt.setInt(1,15);
stmt.setInt(2,1);
c.clear();   c.set(2011,9,25);
stmt.setDate(3, new
java.sql.Date((c.getTime()).getTime()));
```

```
c.set(2011,9,27);
stmt.setDate(4,new
java.sql.Date((c.getTime()).getTime()));

stmt.addBatch();
stmt.setInt(1,16);
stmt.setInt(2,2);
c.clear();
c.set(2011,10,12);
stmt.setDate(3,new
java.sql.Date((c.getTime()).getTime()));
c.set(2011,10,27);
stmt.setDate(4,new
java.sql.Date((c.getTime()).getTime()));
stmt.addBatch();
stmt.executeBatch();
conn.close();
}}
```

CallableStatement

- Interface that inherits from **`PreparedStatement`** that is used to execute stored-procedure.
- **`In Connection.prepareCall(String`** s)is used to create a **`CallableStatement.`**
- The string parameter is of the form

```
{[?=] call <procedure-name>[<arg1>,<arg2>, ...]}
```

- The arguments IN parameter values are set using the set methods inherited from **`PreparedStatement`**.
- All **`OUT`** parameters must be registered before a stored procedure is executed.

```
void registerOutParameter(int parameterIndex, int sqlType)
throws SQLException
```

- **`java.sql.Types has all the`** parameters that can be sent as **`sqlType.`** (Please refer to <docfolder>\docs\api\java\sql\Types.html)

Java Code to call the stored procedure

```
PROCEDURE GETSTUD(IN id1 INT,OUT nm VARCHAR(45))
BEGIN
    SELECT NAME INTO nm FROM STUDENT WHERE REGNO=id1;
END
```

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

String userName = "root";
          String password = "root";
          String url = "jdbc:mysql://localhost/test";
Connection conn = null;
try{

conn =
DriverManager.getConnection(url,userName,password);
CallableStatement c=conn.prepareCall("{call
GETSTUD(?,?)}");
```

```
String name=null;
c.setInt(1,2);
c.registerOutParameter(2,java.sql.Types.VARCHAR);
c.executeUpdate();
name=c.getString(2);

System.out.println("Name retrieved: "+ name);
}
catch (SQLException e)  {
System.err.println ("Failed to connect to database" +e);
}
        finally  {
            if (conn != null)
            try    {    conn.close ();}    catch
(SQLException e) { }
            }
}
}
```

Example: Retrieving `ResultSet` from stored procedure

```
PROCEDURE AllStudents()
BEGIN
Select * from Student;
END
```

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

String userName = "root";
          String password = "root";
          String url = "jdbc:mysql://localhost/test";
Connection conn = null;
try{
conn =
DriverManager.getConnection(url,userName,password);
```

```
CallableStatement c=conn.prepareCall("{call
AllStudents()}");
boolean res = c.execute();
while (res) {
      ResultSet rs = c.getResultSet();
System.out.println("ID       Name     Degree
Semester");
while (rs.next() ) {
System.out.printf( "%2s %10s %10s
%7s\n",rs.getInt(1),rs.getString(2),rs.getString(3),rs.g
etInt(4));
}
        res = c.getMoreResults();
}}
catch (SQLException e) { System.err.println(e);}
finally  {  if (conn != null)
try {conn.close ();} catch (SQLException e) { }
}   }}
```

Transaction support

- Methods in **`Connection`** class helps to demarcate the set of statements into a transaction.
- **`public void setAutoCommit(boolean autoCommit) throws SQLException`**
- **`public void rollback()  throws SQLException`**
- **`public void commit() throws SQLException`**

Example: Transaction support

To understand transactions we will use two tables Login (id, login, password) and Student (regno, name,degree,semester). Insertion of student data will require us to insert data either into both the tables or in none of them in case there is an error. Login will be same as regno.

```
import java.sql.*;
class Trans{
public static void insert(String login, String
pass,int id, String name){
String userName = "root";
    String password = "root";
    String url = "jdbc:mysql://localhost/test";
Connection con = null;
try{
con =
DriverManager.getConnection(url,userName,password);
Statement st=con.createStatement();
con.setAutoCommit(false);
```

```
st.executeUpdate("INSERT INTO Login VALUES("+ id+
",'"+login+"','"+pass+"')");
st.executeUpdate("INSERT INTO Student VALUES("+login+",'"
+ name + "','B.E.',1)");
con.commit();
con.close();
 }  catch(Exception e)  {
       try{
 System.out.println("Rolling back Exception :" +
e.toString());
       con.rollback();
 e.printStackTrace();}
  catch(Exception e1){}     }
 }
 public static void main(String[] s){
insert("1111", "danger", 1,"Emily"); } }
```

Test the application by giving duplicate login values

a) without transaction statements

b) with transaction statements

JDBC Transaction Isolation Levels

- `Connection` interface defines a method to set isolation level
- **`void setTransactionIsolation(int level) throws SQLException`**
- Level can have following static constants define in `Connection`
  - **`TRANSACTION_READ_UNCOMMITTED`** :Allows dirty reads, non-repeatable reads, and phantom reads to occur.
  - **`TRANSACTION_READ_COMMITTED`** :Ensures only committed data can be read.
  - **`TRANSACTION_REPEATABLE_READ`**: Is close to being serializable, however, phantom reads are possible.
  - **`TRANSACTION_SERIALIZABLE`**: Dirty reads, non-repeatable reads, and phantom reads are prevented.

- http://docs.oracle.com/javase/tutorial/jdbc/basics/transactions.html

DatabaseMetaData to check Isolation Levels support

- To check if transaction isolation Levels are supported

```
DatabaseMetaData md = conn.getMetaData();
if
   md.supportsTransactionIsolationLevel(Connection.TRANS
   ACTION_REPEATABLE_READ)) {
System.out.println("TRANSACTION_REPEATABLE_READ is
   supported.");
conn.setTransactionIsolation(Connection.TRANSACTION_REA
   D_COMMITTED);}
```

MySQL supports this