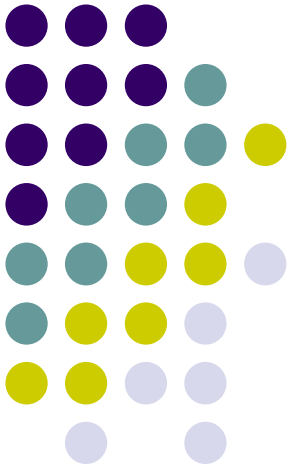
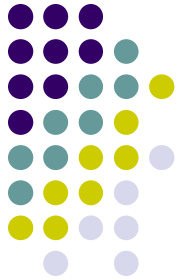
JDBC –

Java DataBase Connectivity

Accessing a Database in Java

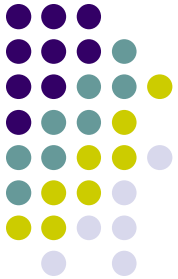


Different Applications in Java



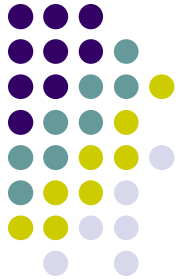
- Standalone Applications
- Web Applications
- Mobile Applications

Different Editions of Java



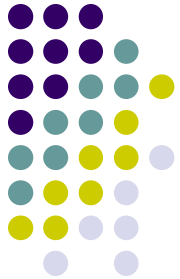
- J2SE (Java to Standard Edition)
- J2EE (Java to Enterprise Edition)
- J2ME (Java to Micro Edition)

J2EE Multi-tier Architecture



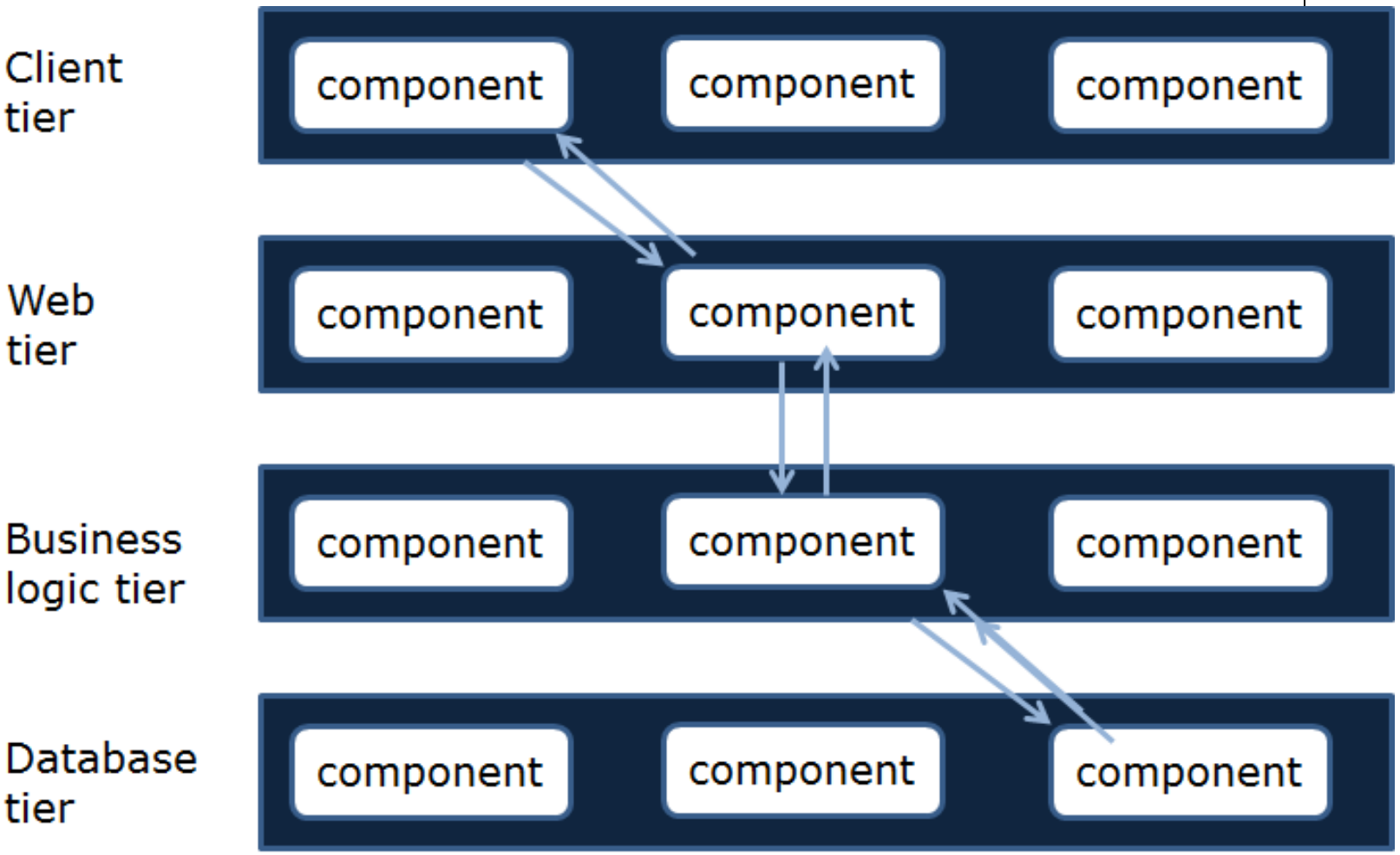
- J2EE is a multi-tier architecture (four layered architecture).
  1. **Client Tier:**
     1. Components of Client Tier will run in the client devices / containers.
     2. Client Tier components are standalone java applications, static and dynamic HTML pages, and applets.
  2. **Web Tier:**
     1. The web tier components namely JSP's and Servlets execute with the help of J2EE web server in a web container.

J2EE Multi-tier Architecture

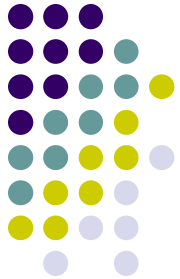


- J2EE is a multi-tier architecture (four layered architecture).
  3. **Business Logic Tier:**
     1. Enterprise Java Beans (EJB) are the business tier components that are executed within the EJB container using J2EE Application Server.
  4. **Database Tier:**
     1. In the Database tier, the application related data are stored in a database.

J2EE Multi-tier Architecture

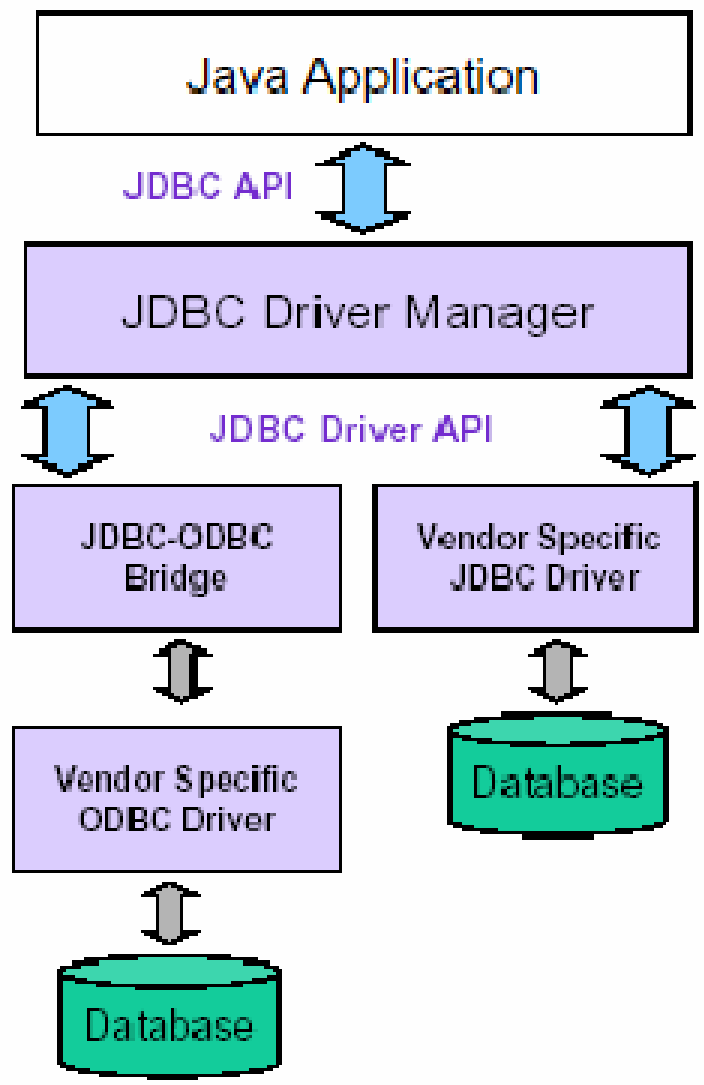


What is JDBC?

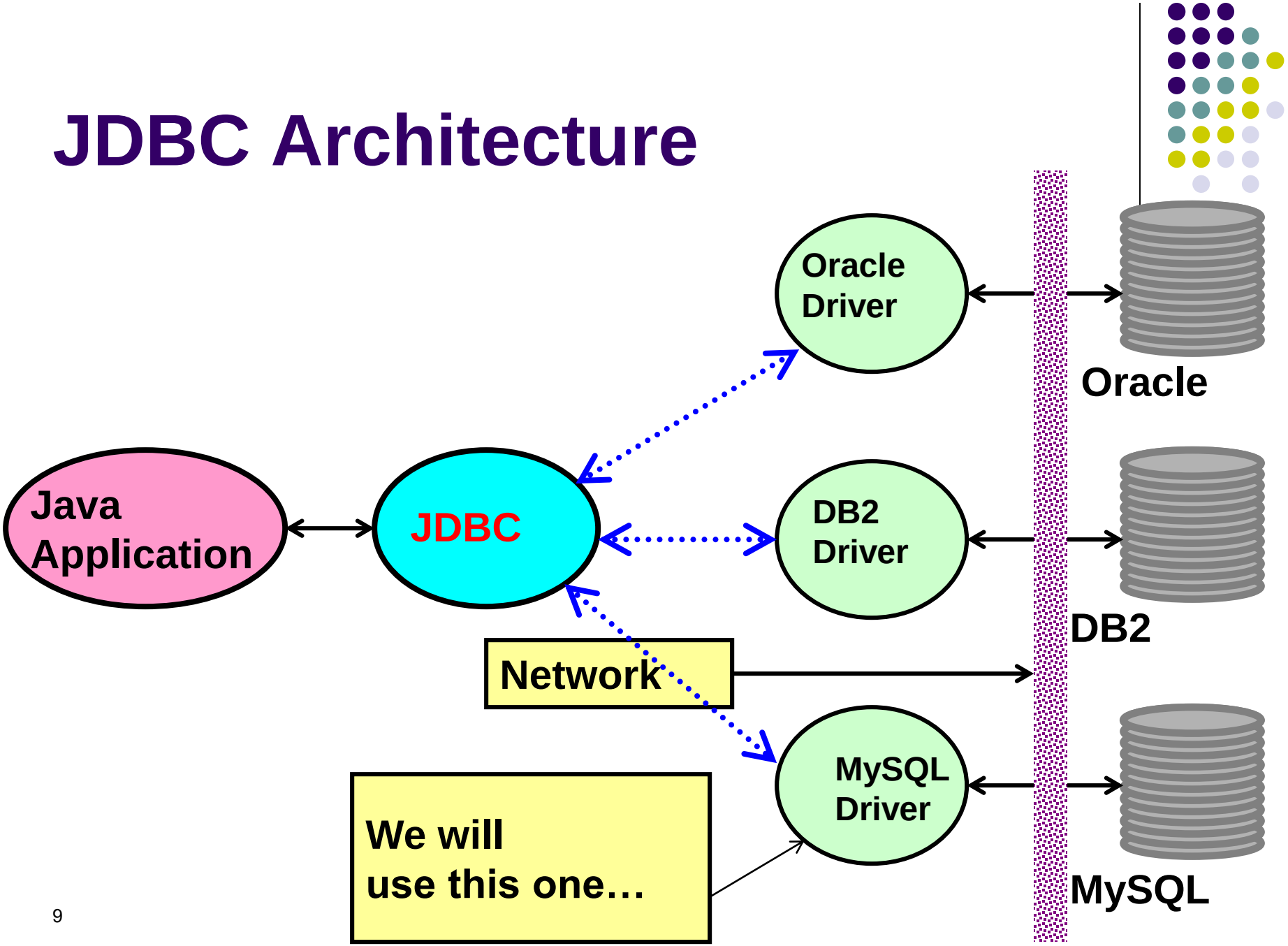


- JDBC helps us to connect to a database and execute SQL statements against a database.
- JDBC API provides set of classes and interfaces with different implementations respective to different databases.
- What's an API?
  - "An API that lets you access virtually any tabular data source from the Java programming language"
- What's a tabular data source?
  - "… access virtually any data source, from relational databases to spreadsheets and flat files."

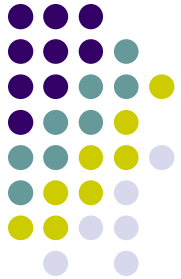
General Architecture



JDBC Architecture

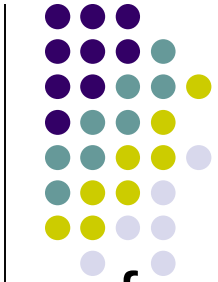


What is JDBC Driver ?



- JDBC drivers implement the defined interfaces in the JDBC API for interacting with your database server.
- For example, using JDBC drivers enable you to open database connections and to interact with it by sending SQL or database commands then receiving results with Java.
- The ***Java.sql*** package that ships with JDK contains various classes with their behaviors defined and their actual implementations are done in third-party drivers.
- Third party vendors implements the *java.sql.Driver* interface in their database driver.
- A driver is nothing but software required to connect to a database from Java program.

JDBC Drivers Types:

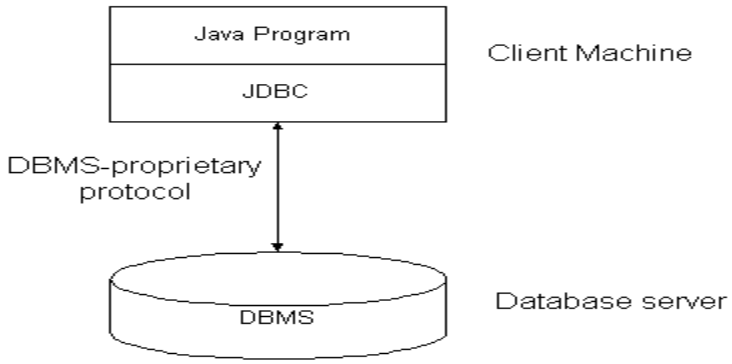


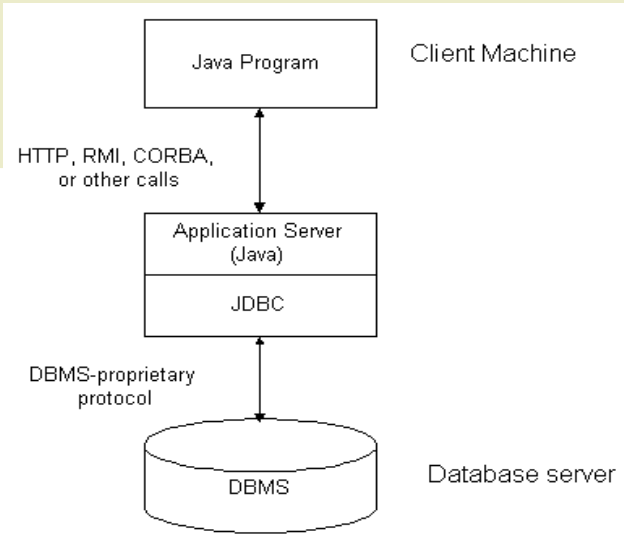
- JDBC driver implementations vary because of the wide variety of operating systems and hardware platforms in which Java operates.
- Sun has divided the implementation types into four categories
- Type 1: JDBC-ODBC Bridge
- Type 2: Native-API/partly Java driver
- Type 3: Net-protocol/all-Java driver
- Type 4: Native-protocol/all-Java driver

Two-tier and three-tier architecture

- The JDBC API supports a two-tier and a three-tier architecture for database access.

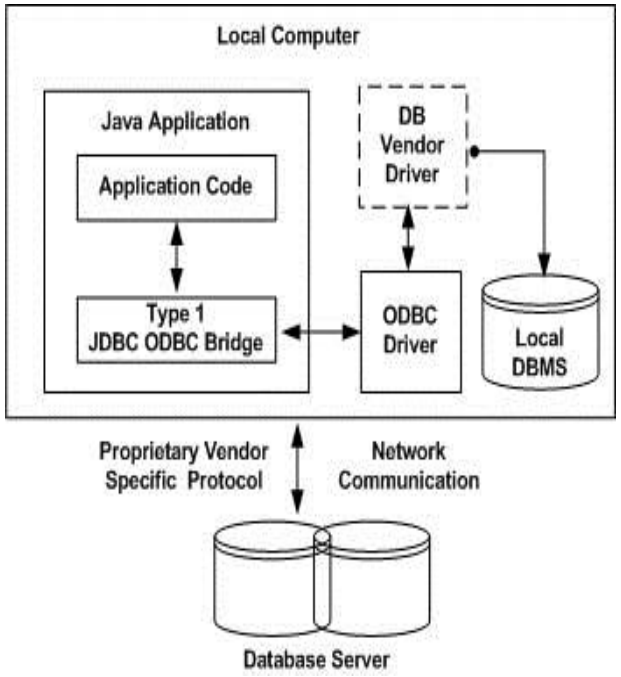
two-tier model	three-tier model
In a two-tier model, a Java application/applet communicates directly with the database, via the JDBC driver. The Java application/applet and the database can be on the same machine, or the database can be on a server and the Java application/applet can be on a client machine using any network protocol.	In a three-tier model, i.e. client, server and database. It is a Java application /applet communicates with a middle tier component that functions as an application server. The application server talks to a given database using JDBC.





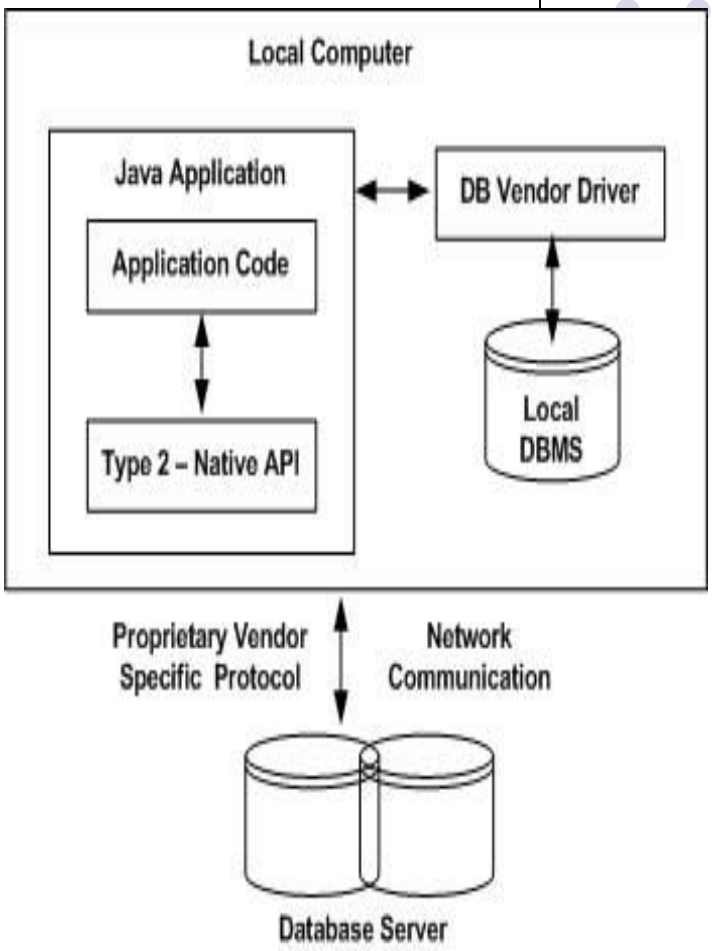
Type 1: JDBC-ODBC Bridge Driver

- In a Type 1 driver, a JDBC bridge is used to access ODBC(Object Database connectivity) drivers installed on each client machine.
- Using ODBC requires configuring on your system a Data Source Name (DSN) that represents the target database.
- When Java first came out, this was a useful driver because most databases only supported ODBC access but now this type of driver is recommended only for experimental use or when no other alternative is available.
- The JDBC-ODBC bridge that comes with JDK 1.2 is a good example of this kind of driver.



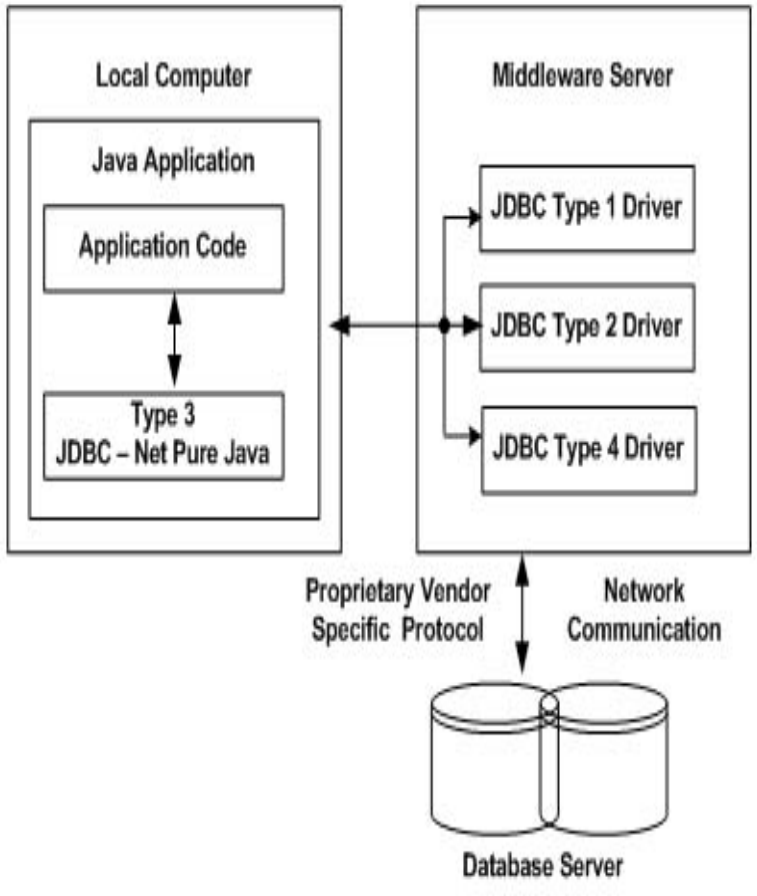
Type 2: JDBC-Native API:

- In a Type 2 driver, JDBC API calls are converted into native C/C++ API calls which are unique to the database. These drivers typically provided by the database vendors and used in the same manner as the JDBC-ODBC Bridge, the vendor-specific driver must be installed on each client machine.
- If we change the Database we have to change the native API as it is specific to a database and they are mostly obsolete now but you may realize some speed increase with a Type 2 driver, because it eliminates ODBC's overhead.
- The Oracle Call Interface (OCI) driver is an example of a Type 2 driver.



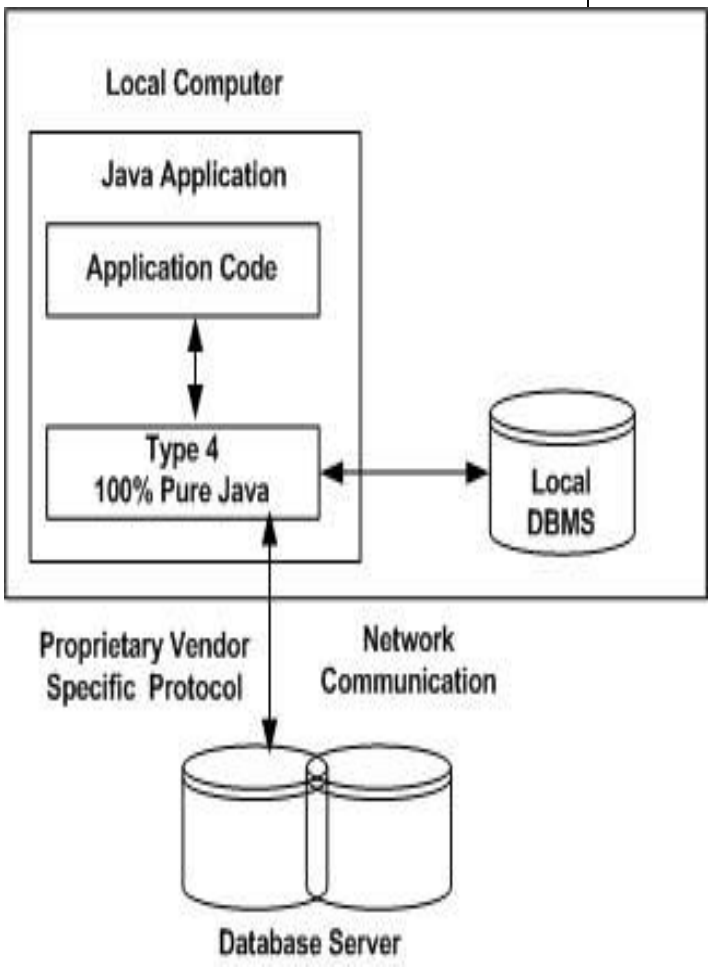
Type 3: JDBC-Net pure Java:

- In a Type 3 driver, a three-tier approach i.e. client, server and database is used.
- The JDBC clients use standard network sockets to communicate with an middleware application server.
- The socket information is then translated by the middleware application server into the call format required by the DBMS, and forwarded to the database server.
- This kind of driver is extremely flexible, since it requires no code installed on the client and a single driver can actually provide access to multiple databases.

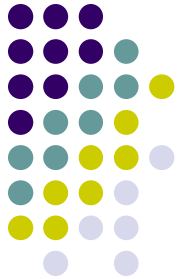


Type 4: 100% pure Java:

- In a Type 4 driver, a pure Java-based driver that communicates directly with vendor's database through socket connection.
- This is the highest performance driver available for the database and is usually provided by the vendor itself.
- This kind of driver is extremely flexible, you don't need to install special software on the client or server.
- Further, these drivers can be downloaded dynamically.
- MySQL's Connector/J driver is a Type 4 driver.
- Because of the proprietary nature of their network protocols, database vendors usually supply type 4 drivers.

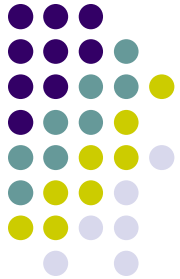


Which Driver should be used?



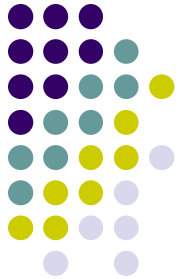
- If you are accessing one type of database, such as Oracle, Sybase, or IBM, the preferred driver type is 4.
- If your Java application is accessing multiple types of databases at the same time, type 3 is the preferred driver.
- Type 2 drivers are useful in situations where a type 3 or type 4 driver is not available yet for your database.
- The type 1 driver is not considered a deployment-level driver and is typically used for development and testing purposes only.

JDBC Driver for MySQL (Connector/J)



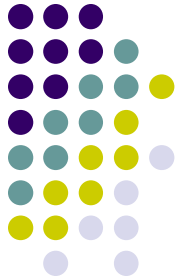
- Download Connector/J using binary distribution from : http://dev.mysql.com/downloads/connector/j/5.0.html
- To install simply unzip (or untar) and put mysql-connector-java-*[version]*-bin.jar (I have installed **mysql-connector-java-5.0.4-bin.jar**) in the class path
- For online documentation, see : http://dev.mysql.com/doc/refman/5.0/en/connector-j.html

Basic steps to use a database in Java



- 1.Establish a **connection**
- 2.Create JDBC **Statements**
- 3.Execute **SQL** Statements
- 4.GET **ResultSet**
- 5.**Close** connections

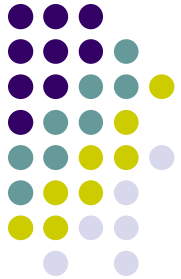
1. Establish a connection



- **import java.sql.*;**
- **Load the vendor specific driver**
  - Class.forName("com.mysql.jdbc.Driver ");
    - What do you think this statement does, and how?
    - Dynamically loads a driver class, for MYSQL database

```
try {
  Class.forName("com.mysql.jdbc.Driver");
}
catch (ClassNotFoundException e) {
  System.out.println("Couldn't load the Driver");
}
```

1. Establish a connection



- **Make the connection**
  - Connection con = DriverManager.getConnection(“jdbc:mysql://localhost:3306/ “+dbname,username, passwd);
    - Establishes connection to database by obtaining a *Connection* object

```
Connection con = null;
try {
    con = DriverManager.getConnection
    ("jdbc:mysql://localhost:3306/mydb", username, passwd);
}
catch (SQLException e) {
    System.out.println("Couldn't Obtain the connection");
}
```

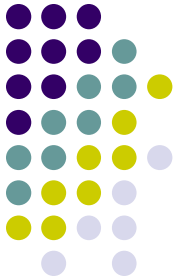
Database URL

The format of a database URL is:

```
String DB_URL = "jdbc:mysql://dbserver:3306/mydb";
```

- Port is the TCP port number where the database server is listening.
  - 3306 is the **default port** for MySQL
- Use hostname **`"localhost"`** for the local machine.

Database URL



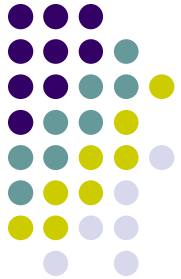
The hostname and port are optional.

For MySQL driver: defaults are localhost and port 3306

Example: These 4 URL refer to the same database

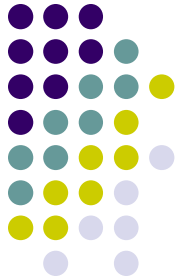
```
"jdbc:mysql://localhost:3306/world"
"jdbc:mysql://localhost/world"
"jdbc:mysql:///world"
"jdbc:mysql:/world"
```

Driver class names



- Oracle: oracle.jdbc.driver.OracleDriver
- MySQL: com.mysql.jdbc.Driver
- MS SQL Server: com.microsoft.jdbc.sqlserver.SQLServerDriver

How to Execute SQL Commands



The Statement interface defines many execute methods:

```
Resultset rs =
        statement.executeQuery("SELECT ...");
```

- use for statements that return data values (SELECT)
- Returns results as a ResultSet object

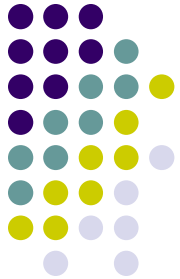
```
int count =
        statement.executeUpdate("UPDATE ...");
```

- Use for DROP TABLE, CREATE TABLE
- use for INSERT, UPDATE, and DELETE

```
boolean b =
        statement.execute("DROP TABLE test");
```

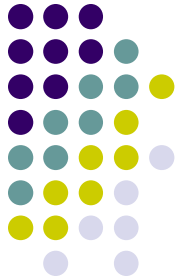
- use to execute any SQL statement(s)

2. Create JDBC statement(s)



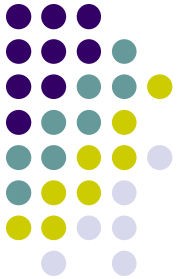
- Creating the query
  - Obtain any of the Statement object in order to execute the query.
  - Statement object can be either Statement, PreparedStatement, or CallableStatement.
  - Use createStatement() method of Connection class to create the simple Statement

2. Create JDBC statement(s) & Executing SQL Statements



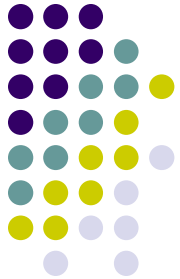
- Executing the query
  - Once the statement is ready, Execute the SQL query using executeQuery() method.
  - It returns a ResultSet object.

3. Executing SQL Statements



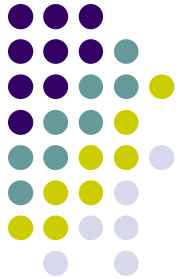
```
Statement st = null;
ResultSet rs = null;
String qry = "select * from stud";
try {
    st = con.createStatement();
    rs = st.executeQuery(qry);
}
catch (SQLException e) {
    System.out.println("Error while processing SQL
    query");
}
```

3. Executing SQL Statements



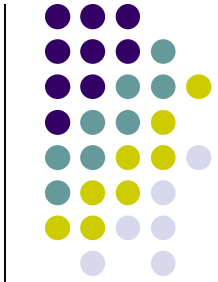
```
String createtab = "Create table stud " +
    "(USN VARCHAR(10), Name VARCHAR(32), " +
    "Marks Integer)";

stmt.executeUpdate(createtab);

String insertstud = "Insert into stud values“ +“(cs-
    01,aaa,89)";

stmt.executeUpdate(insertstud);
```

4. Get ResultSet



```
String querystud = "select * from stud";
ResultSet rs = Stmt.executeQuery(querystud);
while (rs.next()) {
    String usn = rs.getString(“USN");
    String name = rs.getString("NAME");
    int marks = rs.getInt("MARKS");
}
```

5. Close connection



- Last step is to close the database connection.
- This releases the external resources like cursor, handlers etc.

```
try {
        stmt.close();
        pstmt.close();
        rs.close();
        con.close();
}
catch (Exception e) {
        System.out.println("Error while closing the connection");
}
```

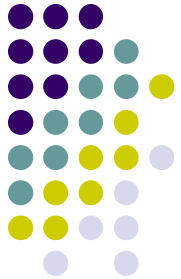
Set up a simple database program

```
import java.sql.*;

public class JdbcExample{
    public static void main(String args[]) {
        Connection con = null;
        try {
             Class.forName("com.mysql.jdbc.Driver").newInstance();
             con = DriverManager.getConnection("jdbc:mysql://localhost:3306/mydb", "root", "");

             if (!con.isClosed())
                  System.out.println("Successfully connected to MySQL");
        }
        catch(Exception e){
             System.err.println("Exception: " + e.getMessage());
        }
        finally {

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

```
//Program to display the contents of a table

import java.sql.*;

public class mysql_demo
{
        public static void main(String[] args)
        {
                System.out.println("MySQL Connect Example.");
                Connection conn = null;
                String url = "jdbc:mysql://localhost:3306/";
                String dbName = "mydb";
                String driver = "com.mysql.jdbc.Driver";
                String userName = "root";
                String password = "";
                String f1,f2;
                try
                {
                        Class.forName(driver).newInstance();
                        conn = DriverManager.getConnection(url+dbName,userName,password);
                        String query = "Select * FROM stud";
                        System.out.println("Connected to the database");
                        Statement stmt = conn.createStatement();
                        ResultSet rs = stmt.executeQuery(query);
```

```
                while (rs.next())
                {
                        f1 = rs.getString(1);
                        f2 = rs.getString(2);
                        System.out.println(f1+"  "+f2);
                } //end while

                conn.close();
                System.out.println("Disconnected from database");
        } //end try
        catch(ClassNotFoundException e)
        {
                e.printStackTrace();
        }
        catch(SQLException e)
        {
                e.printStackTrace();
        }
        catch (Exception e)
        {
                e.printStackTrace();
        }
    }
}
```

Statements in JDBC



- A Statement is an interface that represents a SQL statement.
- You execute Statement objects, and they generate ResultSet objects, which is a table of data representing a database result set.
- You need a Connection object to create a Statement object.

- Example,
- stmt = con.createStatement();

Statements in JDBC

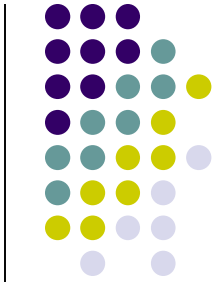
- There are three different kinds of statements:

- **Statement:**
  - Used to implement simple SQL statements with no parameters.

- **PreparedStatement: (Extends Statement)**
  - Used for precompiling SQL statements that might contain input parameters. These input parameters will be given a value in the runtime.

- **CallableStatement: (Extends PreparedStatement)**
  - Used to execute stored procedures that may contain both input and output parameters.

Statement

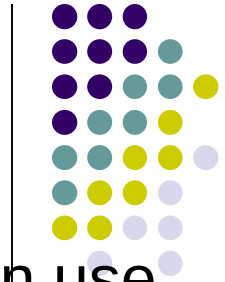


- The Statement object is used whenever a J2EE component needs to immediately execute a query without first having the query compiled.

- Before you can use a Statement object to execute a SQL statement, you need to create one using the Connection object's createStatement() method.

```
Statement st = null;
try {
    st = con.createStatement();
} catch (SQLException e) {
    e.printStackTrace();
}
```

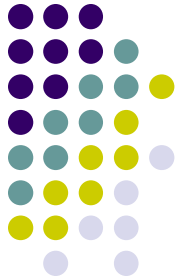
Statement



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

- **boolean execute(String SQL):**
- Returns a boolean value of true if a ResultSet object can be retrieved; otherwise, it returns false.
- Use this method to execute SQL DDL statements like Create, Alter, Drop, etc or when you need to use truly dynamic SQL.

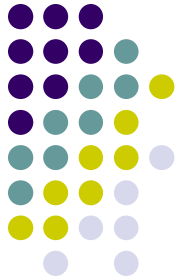
Statement



- **int executeUpdate(String SQL) :**
- Returns the numbers of rows affected by the execution of the SQL statement.
- Use this method to execute SQL statements for INSERT, UPDATE, or DELETE statement.

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

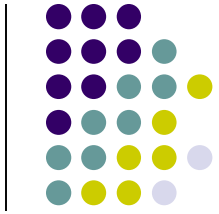
JDBC Programs



Program to read employee details from the EMPLOYEE Table.

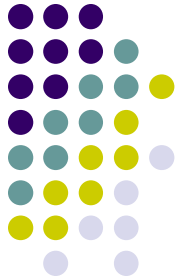
EMPLOYEE(id integer, name varchar, project varchar, salary integer)

JDBC Programs



```
public class TestDBResult {
        public static void main(String a[]){
                try {
                        Class.forName("com.mysql.jdbc.Driver");
                        Connection con = DriverManager.getConnection
("jdbc:mysql://localhost:3306/emp","user","password");

                        Statement stmt = con.createStatement();
                        ResultSet rs = stmt.executeQuery("select * from EMPLOYEE");

                        while(rs.next()){
                                System.out.println(rs.getInt("id"));
                                System.out.println(rs.getString("name"));
                                System.out.println(rs.getString("project"));
                                System.out.println(rs.getInt("salary"));
                        }
                        rs.close();
                        con.close();
                }
                catch (ClassNotFoundException e) {
                        e.printStackTrace();
                }
                catch (SQLException e) {
                        e.printStackTrace();
                }
        }
}
```

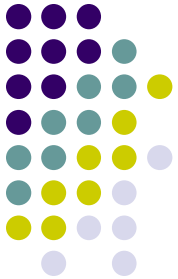
JDBC Programs



Program to update an employee details in the EMPLOYEE Table. Update the salary to 45000 of an employee with id = 10.

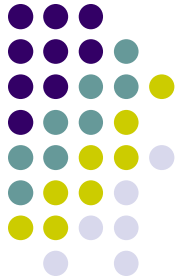
EMPLOYEE(id integer, name varchar, project varchar, salary integer)

JDBC Programs



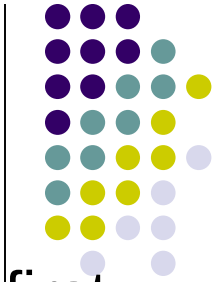
```
public class TestDBUpdate {
        public static void main(String[] args) {
                Connection con = null;
                Statement stmt = null;
                try {
                        Class.forName("com.mysql.jdbc.Driver");
                        Connection con = DriverManager.getConnection
                                ("jdbc:mysql://localhost:3306/emp","user","password");

                        Statement stmt = con.createStatement();
                        String query = "update table EMPLOYEE set salary=45000 where id=10";

                        //count will give you how many records got updated
                        int count = stmt.executeUpdate(query);
                        System.out.println("Updated queries: "+count);
```

JDBC Programs



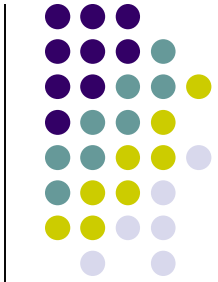
```
                ResultSet rs = stmt.executeQuery("select * from EMPLOYEE where id=10");
                while(rs.next()){
                        System.out.println(rs.getInt("id"));
                        System.out.println(rs.getString("name"));
                        System.out.println(rs.getString("project"));
                        System.out.println(rs.getInt("salary"));
                }
        }
        catch (ClassNotFoundException e) {
                e.printStackTrace();
        }
        catch (SQLException e) {
                e.printStackTrace();
        }
        finally{
                try{
                        if(stmt != null)
                                stmt.close();
                        if(con != null)
                                con.close();
                }
                catch(Exception ex){}
        }
    }
}
```

PreparedStatement



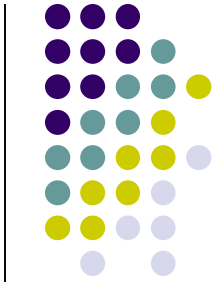
- The PreparedStatement is used to compile the query first before executing it.
- The PreparedStatement interface extends the Statement interface.
- This statement gives you the flexibility of supplying input arguments dynamically.
- All parameters (arguments) are represented by the ? symbol, which is known as the place holder.

PreparedStatement



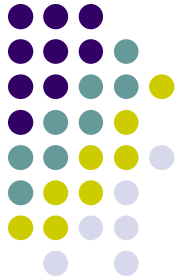
- You must supply values for every parameter before executing the SQL statement.
- The setXXX() methods bind values to the parameters, where XXX represents the Java data type of the value.
- Each parameter marker is referred to by its ordinal position. The first marker represents position 1, the next position 2, and so forth.
- All of the Statement object's methods for interacting with the database: execute(), executeQuery(), and executeUpdate() also work with the PreparedStatement object.

PreparedStatement



```
PreparedStatement ps = null;
try {
        String qry = "update stud set name=? where usn=?";
        ps = con.prepareStatement(qry);
        ps.setString(1, "ABCD");
        ps.setString(2, "1MS12CS001");
        ps.execute();
}
catch (SQLException e) {
        e.printStackTrace();
}
finally {
        try {
                ps.close();
        }
        catch (Exception e) {
                e.printStackTrace();
        }
}
```

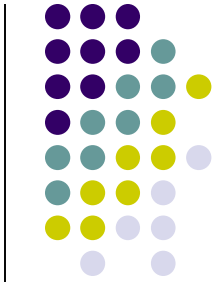
JDBC Programs



- Insert a new row into the EMPLOYEE Table using Prepared Statements.

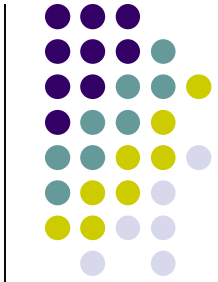
- EMPLOYEE(id integer, name varchar, project varchar, salary integer)

JDBC Programs



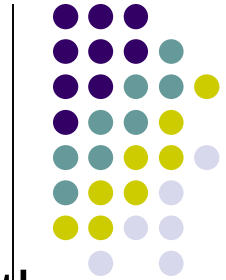
```
public class TestDBInsert {
        public static void main(String[] args) {
                Connection con = null;
                PreparedStatement prSt = null;
                try {
                        Class.forName("com.mysql.jdbc.Driver");
                        Connection con = DriverManager.getConnection
                                ("jdbc:mysql://localhost:3306/emp","user","password");

                        String query="insert into emp(id,name,project,salary) values(?,?,?,?)";
                        prSt = con.prepareStatement(query);
                        prSt.setInt(1, 11);
                        prSt.setString(2, "PQR");
                        prSt.setString(3, "Automation Testing");
                        prSt.setInt(4, 50000);

                        //count will give you how many records got updated
                        int count = prSt.executeUpdate();
```

JDBC Programs



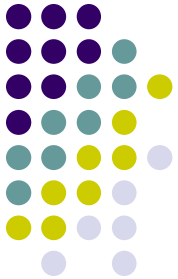
```
                //Run the same query with different values
                prSt.setInt(1, 12);
                prSt.setString(2, "XYZ");
                prSt.setString(3, "Manual Testing");
                prSt.setInt(4, 40000);
                count = prSt.executeUpdate();
            }
            catch (ClassNotFoundException e) {
                e.printStackTrace();
            }
            catch (SQLException e) {
                e.printStackTrace();
            }
            finally{
                try{
                    if(stmt != null)
                        stmt.close();
                    if(con != null)
                        con.close();
                }
                catch(Exception ex){}
            }
        }
}
```

CallableStatement



- CallableStatement object is used to execute a call to the stored procedure from within a J2EE object.
- The stored procedure can be written in PL/SQL, Transact-SQL, C, or another programming language.
- The stored procedure is a block of code and is identified by a unique name.

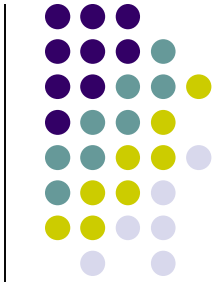
CallableStatement



Example:

```
CREATE OR REPLACE PROCEDURE getStudName
(STUD_ID IN VARCHAR, STUDENT_NAME OUT
   VARCHAR) AS
BEGIN
   SELECT name INTO STUDENT_NAME
   FROM Student
   WHERE ID = STUD_ID;
END;
```

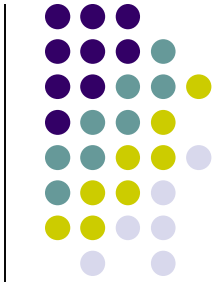
CallableStatement



- Three types of parameters exist:
  1. **IN**
  2. **OUT**
  3. **INOUT**

- **IN:**
- A parameter whose value is unknown when the SQL statement is created.
- You bind values to IN parameters with the setXXX() methods.

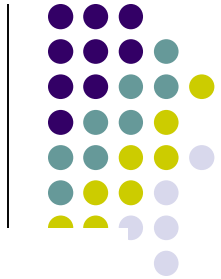
CallableStatement



- **OUT:**
- A parameter whose value is supplied by the SQL statement it returns.
- You should register this parameter using registerOutParameter() method.
- You retrieve values from the OUT parameters with the getXXX() methods.

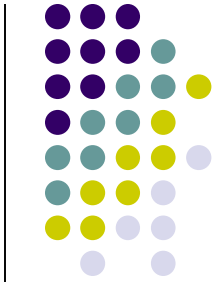
- **INOUT:**
- A parameter that provides both input and output values.
- You bind variables with the setXXX() methods and retrieve values with the getXXX() methods.

CallableStatement



```
CallableStatement cs = null;

try {

String qry = "{CALL getStudName (?,?) }";

cs = con.prepareCall(qry);
cs.setString(1, "1MS12CS001");
cs.registerOutParameter(2, Types.VARCHAR);
cs.execute();

String name = cs.getString(2);
System.out.println("Student name is .. "+name);

}
catch (Exception e) {
        e.printStackTrace();
}
finally {
        try {
                cs.close();
        }
        catch (Exception e) {
                e.printStackTrace();
        }
}
```

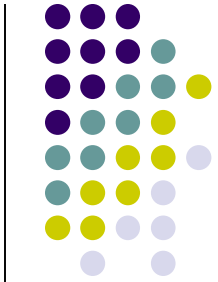
JDBC Programs



- Retrieve the Project Name of an Employee with id = 12 from the EMPLOYEE Table using Callable Statements.

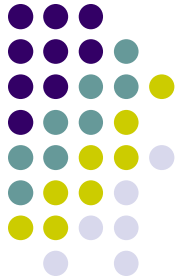
- EMPLOYEE(id integer, name varchar, project varchar, salary integer)

JDBC Programs



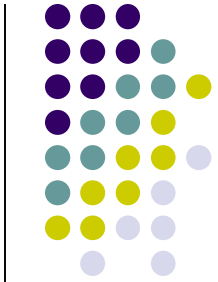
```
public class TestCallable {
        public static void main(String[] args) {
                Connection con = null;
                CallableStatement callSt = null;
                try {
                        Class.forName("com.mysql.jdbc.Driver");
                        Connection con = DriverManager.getConnection
                                ("jdbc:mysql://localhost:3306/emp","user","password");

                        callSt = con.prepareCall("{call getProjectName(?,?)}");
                        callSt.setInt(1,12);
                        callSt.registerOutParameter(2, Types.VARCHAR);
                        callSt.execute();

                        String projectName = callSt.getString(2);
                        System.out.println("Project : " + projectName);
                }
```

JDBC Programs



```
                catch (ClassNotFoundException e) {
                        e.printStackTrace();
                }
                catch (SQLException e) {
                        e.printStackTrace();
                }
                finally{
                        try{
                                if(stmt != null)
                                        stmt.close();
                                if(con != null)
                                        con.close();
                        }
                        catch(Exception ex){}
                }
        }
}
```

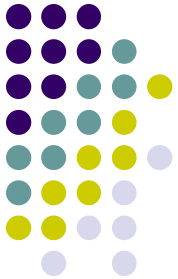
JDBC Programs INOUT Parameter



- Retrieve the Marks of a Student with id = 123 from the Student Table using Callable Statements.

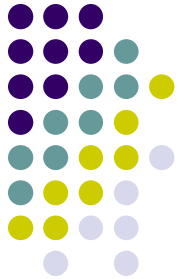
- Student(id integer, marks integer)

INOUT Parameter



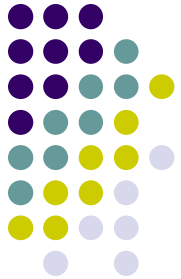
```
public class Main {
	public static void main(String[] args) {
		CallableStatement callSt = null;
		try {
			Class.forName("com.mysql.jdbc.Driver");
			Connection con = DriverManager.getConnection
			("jdbc:mysql://localhost:3306/Student","user","password");
			callSt = con.prepareCall("{call getStudentMark(?)}");
			callSt.setInt(1,123);
			callSt.registerOutParameter(1, Types.INTEGER);
			callSt.execute();
			int marks = callSt.getInt(1);
			System.out.println("Marks  : " + marks);
		}
```

INOUT Parameter



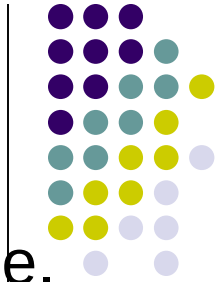
```
                catch (Exception e) {
                        e.printStackTrace();
                }
                finally{
                        try{
                                if(stmt != null)
                                        stmt.close();
                                if(con != null)
                                        con.close();
                        }
                        catch(Exception ex){}
                }
        }
}
```

ResultSet in JDBC



- A ResultSet object maintains a virtual cursor that points to a row in the result set.
- The "result set" refers to the virtual table of row and column data contained in a ResultSet object.
- The virtual cursor is initially positioned above the first row of data when the ResultSet is returned by executeQuery() method.

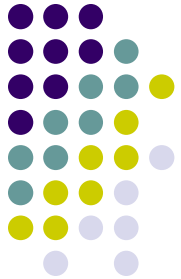
ResultSet in JDBC



- There are various methods in the ResultSet interface.
- These methods can be grouped into three categories as follows

1. **Get Methods:** Used to view the data in the columns of the current row being pointed by the virtual cursor.
2. **Navigational Methods:** Used to move the virtual cursor around the virtual table.
3. **Update methods:** Used to update the data in the columns of the current row.

Reading the ResultSet using Get methods

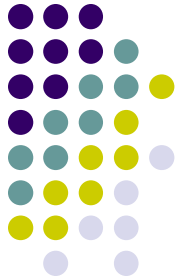


- There is a get method in ResultSet interface for each of the possible data types of the form getXXX() where XXX is the data type of the column, and each get method has two versions:

- One that takes in a column name.
- Example: String name = **rs.getString("name");**

- One that takes in a column index.
- Example: String name = **rs.getString(1);**

ResultSet Methods for Getting Data



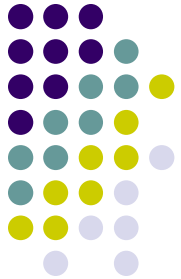
ResultSet "get" methods return column data:

getLong(3) : get by column index (most efficient)

getLong("population") : get by field name (safest)

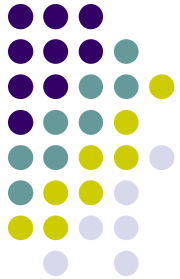
```
getInt( ), getLong( )     - get Integer field value
getFloat( ), getDouble() - get floating pt. value
getString( )  - get Char or Varchar field value
getDate( )    - get Date or Timestamp field value
getBoolean( ) - get a Bit field value
getBytes( )   - get Binary data
getBigDecimal( ) - get Decimal field as BigDecimal
getBlob( )    - get Binary Large Object
getObject( )  - get any field value
```

Scrolling a ResultSet



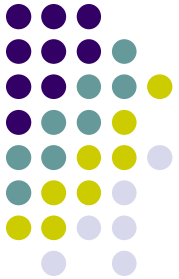
- There are several methods in the ResultSet interface that involve moving the cursor.
- **public void beforeFirst() throws SQLException**
  - Moves the cursor to the front of this ResultSet object, just before the first row.
  - This method has no effect if the result set contains no rows.
- **public void afterLast() throws SQLException**
  - Moves the cursor to the end of this ResultSet object, just after the last row.
  - This method has no effect if the result set contains no rows.

Scrolling a ResultSet



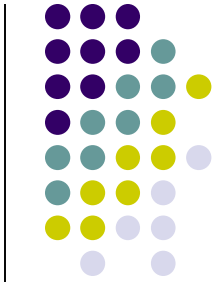
- **public boolean first() throws SQLException**
  - Moves the cursor to the first row in this ResultSet object.
- **public void last() throws SQLException**
  - Moves the cursor to the last row in this ResultSet object.
- **public boolean absolute(int row) throws SQLException**
  - Moves the cursor to the given row number in this ResultSet object.
- **public boolean next() throws SQLException**
  - Moves the cursor forward one row from its current position.

Scrolling a ResultSet



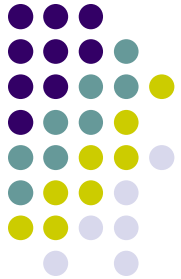
- **public boolean relative(int row) throws SQLException**
  - Moves the cursor a relative number of rows, either positive or negative.
- **public boolean previous() throws SQLException**
  - Moves the cursor to the previous row in this ResultSet object.
- **public int getRow() throws SQLException**
  - Retrieves the current row number.

Scrollable ResultSet



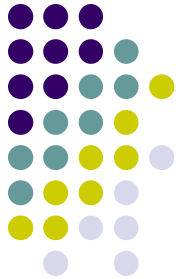
- **Types of ResultSet:**
  - **ResultSet.TYPE_FORWARD_ONLY**
    - The cursor can only move forward in the result set.
    - This is the default.
  - **ResultSet.TYPE_SCROLL_INSENSITIVE**
    - The cursor can scroll forwards and backwards.
    - The result set is not sensitive to changes made by others to the database that occur after the result set was created.

Scrollable ResultSet



- **Types of ResultSet:**
  - **ResultSet.TYPE_SCROLL_SENSITIVE**
    - The cursor can scroll forwards and backwards.
    - The result set is sensitive to changes made by others to the database that occur after the result set was created.

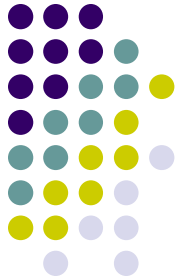
JDBC 2 – Scrollable Result Set



- JDBC 2.0 includes scrollable result sets. Additional methods included are : ‘first’, ‘last’, ‘previous’, and other methods

```
Statement stmt =
con.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
                    ResultSet.CONCUR_READ_ONLY);

String query = “select name from stud where type=‘not sleeping’ “;

ResultSet rs = stmt.executeQuery( query );
    rs.previous();  / / go back in the RS (not possible in JDBC 1…)
    rs.relative(-5); / / go 5 records back
    rs.relative(7); / / go 7 records forward
    rs.absolute(100); / / go to 100th record
…
```

Concurrency of ResultSet

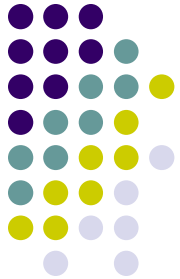


- Rows contained in ResultSet's virtual table can either be read only or be updatable.

- **ResultSet.CONCUR_READ_ONLY**
  - Creates a read-only result set.
  - This is the default.

- **ResultSet.CONCUR_UPDATABLE**
  - Creates an updateable result set.

Concurrency of ResultSet



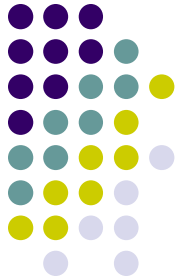
Example 1:

```
Statement st = null;
st=con.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
    ResultSet.CONCUR_UPDATABLE);
```

Example 2:

```
Statement st = null;
st=con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
    ResultSet.CONCUR_UPDATABLE)
```

Updating a ResultSet

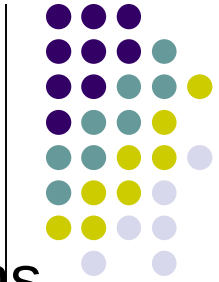


- The ResultSet interface contains a collection of update methods for updating the data of a result set.
- As with the get methods, there are two updateXXX() methods for each data type, where XXX is the data type of the column:

- One that takes in a column name.
- Example: **rs.updateString(“name", “XYZ");**

- One that takes in a column index.
- Example: **rs.updateString(1, “XYZ");**

Updating a ResultSet



- Updating a row in the result set changes the columns of the current row in the ResultSet object (virtual table), but not in the underlying database.

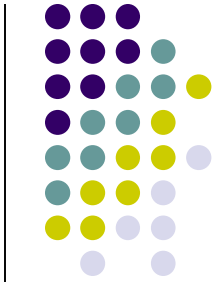
- To update your changes to the row in the database, you need to invoke one of the following methods.

- **public void updateRow()**

- **public void deleteRow()**

- **public void insertRow()**

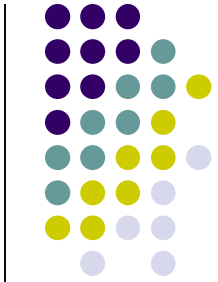
Updating a ResultSet



Updating the ResultSet

```
rs = st.executeQuery(qry);
while (rs.next())
{
    if (rs.getString("name").equals(“XYZ"))
    {
        rs.updateString("name", “ABCD");
        rs.updateRow();
    }
}
```

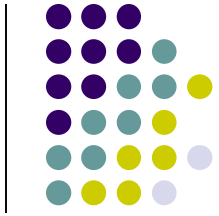
Updating a ResultSet



Deleting a Row

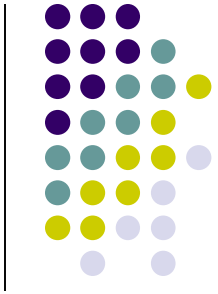
```
rs = st.executeQuery(qry);
while (rs.next())
{
    if (rs.getString("name").equals(“PQR"))
    {
        rs.deleteRow();
    }
}
```

Updating a ResultSet



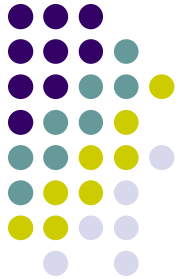
```
public class TestUpdatableResultSet {
        public static void main(String a[]){
                Connection con = null;
                Statement st = null;
                ResultSet rs = null;
                try {
                        Class.forName("com.mysql.jdbc.Driver");
                        con = DriverManager.getConnection
                                ("jdbc:mysql://localhost:3306/emp","user","password");
                        st = con.createStatement
                                (ResultSet.TYPE_SCROLL_SENSITIVE,ResultSet.CONCUR_UPDATABLE);

                        rs = st.executeQuery("select id, name, salary from EMPLOYEE");
                        while(rs.next()){
                                if(rs.getInt(1) == 12){
                                        rs.updateInt(3, 60000);
                                        rs.updateRow();
                                        System.out.println("Record updated!!!");
                                }
                        }
                }
```

Updating a ResultSet



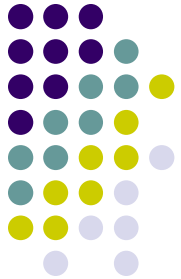
```
                catch (ClassNotFoundException e) {
                        e.printStackTrace();
                }
                catch (SQLException e) {
                        e.printStackTrace();
                }
                finally{
                        try{
                                if(rs != null)
                                        rs.close();
                                if(st != null)
                                        st.close();
                                if(con != null)
                                        con.close();
                        }
                        catch(Exception ex){
                        }
                }
        }
}
```

Meta Data



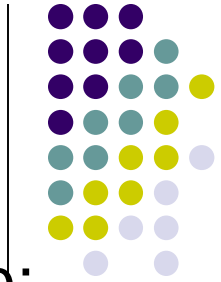
- There are two types of metadata that can be retrieved from the DBMS.
- **DatabaseMetaData** which is the data about the database
- **ResultSetMetaData** which is the data about the Result Set

DatabaseMetaData



- Meta data is the data about the data.
- A J2EE component can access the metadata by using the DatabaseMetaData interface.
- It is used to retrieve information about databases, tables, column, indexes, etc.
- getMetaData() method of Connection object is used to retrieve the metadata about the database.

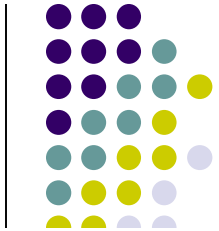
DatabaseMetaData



- Different methods in DatabaseMetaData interface:

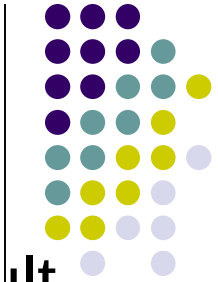
Methods	Information
getDatabaseProductName()	Returns the product name of the DBMS
getUserName()	Returns the username
getURL()	Returns the URL for the database
getSchemas()	Returns all the schema names available in this database
getPrimaryKeys()	Returns primary keys
getProcedures()	Returns stored procedure names
getTables()	Returns names of tables in the database

DatabaseMetaData



```
public class TestDBMetadata {
        public static void main(String a[]){
                try {
                        Class.forName("com.mysql.jdbc.Driver");
                        Connection con = DriverManager.getConnection
                                ("jdbc:mysql://localhost:3306/emp","user","password");

                        DatabaseMetaData dm = con.getMetaData();
                        System.out.println(dm.getDriverVersion());
                        System.out.println(dm.getDriverName());
                        System.out.println(dm.getDatabaseProductName());
                        System.out.println(dm.getDatabaseProductVersion());
                }
                catch (SQLException e) {
                        e.printStackTrace();
                }
                finally{
                        try {
                                if(con != null)
                                        con.close();
                        }
                        catch (SQLException e) {
                                e.printStackTrace();
                        }
                }
        }
}
```

ResultSetMetaData



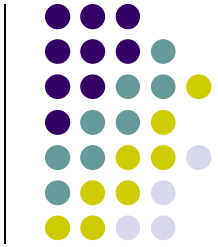
- ResultSetMetaData interface describes the Result set (virtual database).
- getMetaData() method of the ResultSet object can be used to retrieve the Result set metadata.
- Commonly used methods in ResultSetMetaData interface

Methods	Information
getColumnCount()	Returns the number of columns obtained in the ResultSet
getColumnName(int number)	Returns the name of the column specified by the column number
getColumnType(int number)	Returns the data type of the column specified by the column number

ResultSetMetaData

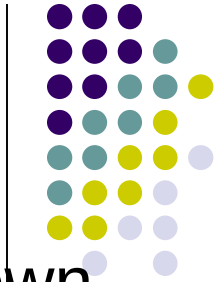
```
try {
        Class.forName("com.mysql.jdbc.Driver");
        Connection con = DriverManager.getConnection
                ("jdbc:mysql://localhost:3306/emp","user","password");

        st = con.createStatement();
        rs = st.executeQuery("select * from EMPLOYEE");
        ResultSetMetaData rsmd = rs.getMetaData();
        int columnCount = rsmd.getColumnCount();

        for(int i=0;i<=columnCount;i++){
                System.out.println(rsmd.getColumnName(i));
                System.out.println(rsmd.getColumnType(i));
        }
}
```

Data Types



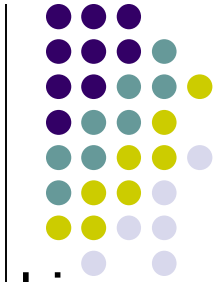
SQL	JDBC/Java	setXXX	getXXX	updateXXX
VARCHAR	java.lang.String	setString	getString	updateString
CHAR	java.lang.String	setString	getString	updateString
BIT	boolean	setBoolean	getBoolean	updateBoolean
NUMERIC	java.math.BigDecimal	setBigDecimal	getBigDecimal	updateBigDecimal
TINYINT	byte	setByte	getByte	updateByte
SMALLINT	short	setShort	getShort	updateShort
INTEGER	int	setInt	getInt	updateInt
BIGINT	long	setLong	getLong	updateLong
REAL	float	setFloat	getFloat	updateFloat
FLOAT	float	setFloat	getFloat	updateFloat
DOUBLE	double	setDouble	getDouble	updateDouble
VARBINARY	byte[ ]	setBytes	getBytes	updateBytes
BINARY	byte[]	setBytes	getBytes	updateBytes
DATE	java.sql.Date	setDate	getDate	updateDate
TIME	java.sql.Time	setTime	getTime	updateTime

Handling Errors with Exceptions



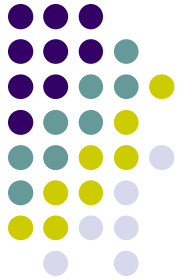
- There are three kinds of exceptions that are thrown by JDBC methods.
- SQLException
- SQLWarning
- DataTruncation

Handling Errors with Exceptions



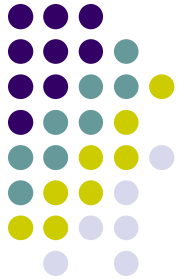
- **SQLException:**
  - It commonly reflects a syntax error in the query and is thrown by many of the methods.
- **SQLWarning:**
  - It throws warnings received by the connection from the DBMS.
  - The getWarning() method of the Connection object retrieves the warning and the getNextWarning() method retrieves the subsequent warnings.
- **DataTruncation:**
  - It is thrown whenever a data is lost due to the truncation of the data value.

Handling Errors with Exceptions



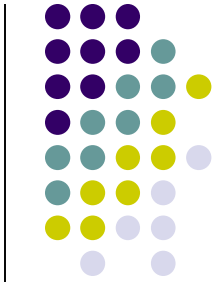
- Programs should recover and leave the database in a consistent state.
- If a statement in the try block throws an exception or warning, it can be caught in one of the corresponding catch statements
- How might a finally {…} block be helpful here?
- E.g., you could rollback your transaction in a catch { …} block or close database connection and free database related resources in finally {…} block

Database Transaction



- A Database Transaction consists of a set of SQL statements, each of which must be successfully completed for the transaction to be completed.
- If one fails, SQL statements that executed successfully up to that point in the transaction must be rolled back.
- A database transaction, by definition, must be atomic, consistent, isolated and durable (ACID).

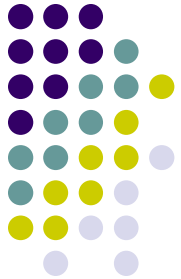
Database Transaction



- Atomicity: Atomicity requires that each transaction is "all or nothing": if one part of the transaction fails, the entire transaction fails, and the database state is left unchanged.

- Consistency: The consistency property ensures that any transaction will bring the database from one valid state to another.

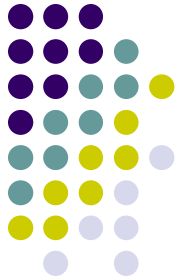
Database Transaction



- Isolation: The isolation property ensures that the concurrent execution of transactions results in a system state that would be obtained if transactions were executed serially, i.e. one after the other.

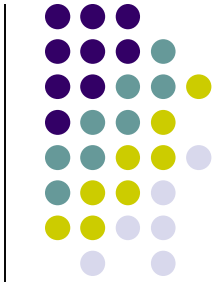
- Durability: Durability means that once a transaction has been committed, it will remain so, even in the event of power loss, crashes, or errors. In a relational database, for instance, once a group of SQL statements execute, the results need to be stored permanently.

commit() and rollback() methods



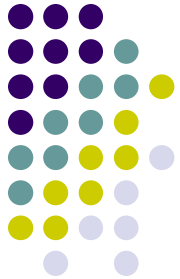
- A database transaction isn't completed until the J2EE component calls the commit() method of the Connection object.
- All SQL statements executed prior to the call to the commit() method can be rolled back.
- However, once the commit() method is called, none of the SQL statements can be rolled back.

commit() and rollback() methods



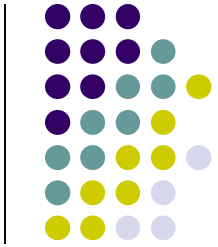
- The commit() method must be called regardless if the SQL statement is part of a transaction or not.
- Till now, we never issued a commit() on the database but still we were able to save the data in the database.
- This is because of the auto commit feature of the database connection.

commit() and rollback() methods



```
public class TestDBTransaction {
        public static void main(String[] args) {
                try {
                        Class.forName("com.mysql.jdbc.Driver");
                        Connection con = DriverManager.getConnection
                                ("jdbc:mysql://localhost:3306/emp","user","password");
                        con.setAutoCommit(false);

                        Statement stmt1 = con.createStatement();
                        Statement stmt2 = con.createStatement();
                        String query1 = "update table EMPLOYEE set salary=45000 where id=10";
                        String query2 = "update table EMPLOYEE set salary=50000 where id=11";

                        //count will give you how many records got updated
                        int count = stmt1.executeUpdate(query1);
                        System.out.println("Updated queries: "+count);
                        count = stmt1.executeUpdate(query2);
                        System.out.println("Updated queries: "+count);

                        con.commit();
```

commit() and rollback() methods

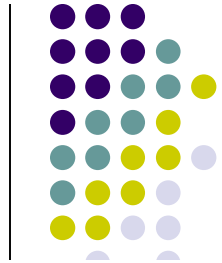


```
                ResultSet rs = stmt.executeQuery("select * from EMPLOYEE");
                while(rs.next()){
                        System.out.println(rs.getInt("id"));
                        System.out.println(rs.getString("name"));
                        System.out.println(rs.getString("project"));
                        System.out.println(rs.getInt("salary"));
                }
        }
        catch (Exception e) {
                System.out.println("Transaction Failed");
                try {
                        con.rollback();
                }
                catch (SQLException e1) {
                        e1.printStackTrace();
                }
                e.printStackTrace();
        }
        finally{
                try{
                        stmt1.close();
                        stmt2.close();
                        con.close();
                }catch(Exception ex){}
        }
    }
}
```

Using Savepoints

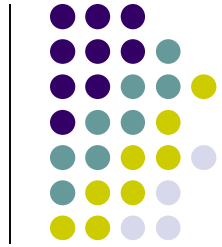
- A transaction may consist of many tasks, some of which don't need to be rolled back when the transaction fails.
- The J2EE component can control the number of tasks that are rolled back by using savepoints.
- A savepoint is a virtual marker that defines the task at which the rollback stops.
- When you set a savepoint you define a logical rollback point within a transaction.
- If an error occurs past a savepoint, you can use the rollback method to undo either all the changes or only the changes made after the savepoint.

Using Savepoints



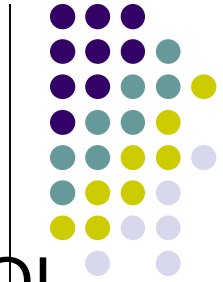
```
public class TestSavePoints {
        public static void main(String[] args) {
                try {
                        Class.forName("com.mysql.jdbc.Driver");
                        Connection con = DriverManager.getConnection
                                ("jdbc:mysql://localhost:3306/emp","user","password");
                        con.setAutoCommit(false);

                        Statement stmt1 = con.createStatement();
                        Statement stmt2 = con.createStatement();
                        String query1 = "update table EMPLOYEE set salary=45000 where id=10";
                        String query2 = "update table EMPLOYEE set salary=50000 where id=11";

                        //count will give you how many records got updated
                        int count = stmt1.executeUpdate(query1);
                        System.out.println("Updated queries: "+count);

                        Savepoint sp = con.setSavepoint();

                        count = stmt1.executeUpdate(query2);
                        System.out.println("Updated queries: "+count);
```

Using Savepoints



```
                con.rollback(sp1);
                con.releaseSavepoint(sp1);

                con.commit();

                ResultSet rs = stmt.executeQuery("select * from EMPLOYEE");
                while(rs.next()){
                        System.out.println(rs.getInt("id"));
                        System.out.println(rs.getString("name"));
                        System.out.println(rs.getString("project"));
                        System.out.println(rs.getInt("salary"));
                }
        }catch (Exception e) {
                System.out.println("Transaction Failed");
                try {
                        con.rollback();
                }
                catch (SQLException e1) {
                        e1.printStackTrace();
                }
                e.printStackTrace();
        }finally{
                try{
                        stmt1.close();
                        stmt2.close();
                        con.close();
                }catch(Exception ex){}
}       }       }
```

Batching SQL statements

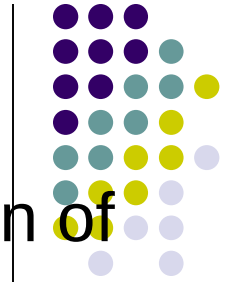


- Batch Processing allows you to group related SQL statements into a batch and submit them with one call to the database.

- When you send several SQL statements to the database at once, you reduce the amount of communication overhead, thereby improving performance.

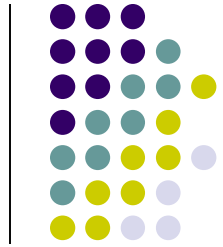
- The addBatch() method of Statement, PreparedStatement, and CallableStatement is used to add individual statements to the batch.

Batching SQL statements



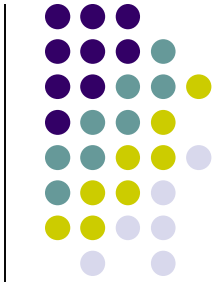
- The executeBatch() is used to start the execution of all the statements grouped together.
- The executeBatch() returns an array of integers, and each element of the array represents the update count for the respective update statement.
- A value of -1 indicates the failure of update statement.
- You can remove the statements with the clearBatch() method.

Batching SQL statements



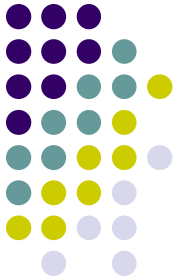
```
public class TestBatchStmnts {
        public static void main(String[] args) {
                try {
                        Class.forName("com.mysql.jdbc.Driver");
                        Connection con = DriverManager.getConnection
                                ("jdbc:mysql://localhost:3306/emp","user","password");
                        con.setAutoCommit(false);

                        Statement stmt1 = con.createStatement();
                        String query1 = "update table EMPLOYEE set salary=45000 where id=12";
                        String query2 = "update table EMPLOYEE set salary=50000 where id=13";

                        stmt1.addBatch(qry1);
                        stmt1.addBatch(qry2);
                        stmt1.executeBatch();
                        stmt1.clearBatch();

                        con.commit();
```

Batching SQL statements



```
                }
                catch (Exception e) {
                        System.out.println("Transaction Failed");
                        try {
                                con.rollback();
                        }
                        catch (SQLException e1) {
                                e1.printStackTrace();
                        }
                        e.printStackTrace();
                }
                finally{
                        try{
                                stmt1.close();
                                con.close();
                        }catch(Exception ex){}
                }
        }
}
```

Creating Table with Prepared Statement



```
public class CreateTableExample {
 private static final String CREATE_TABLE_SQL="CREATE TABLE employee ("
                    + "EMP_ID int(11) NOT NULL,"
                    + "NAME VARCHAR(45) NOT NULL,"
                    + "DOB DATE NOT NULL,"
                    + "EMAIL VARCHAR(45) NOT NULL,"
                    + "DEPT varchar(45) NOT NULL,"
                    + "PRIMARY KEY (EMP_ID))";
 public static void main(String[] args) {
   String jdbcUrl = "jdbc:mysql://localhost:3306/MYDB";
   String username = "root";
   String password = "admin";

   Connection conn = null;
   PreparedStatement stmt = null;
```

Creating Table with Prepared Statement

```
  try {

    conn = DriverManager.getConnection(jdbcUrl, username, password);
    stmt = conn.prepareStatement(CREATE_TABLE_SQL);
    stmt.executeUpdate();

    System.out.println("Table created");

  } catch (SQLException e) {
    e.printStackTrace();
  } finally {
    try {
      if (stmt != null) {
        stmt.close();
      }
      if (conn != null) {
        conn.close();
      }
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

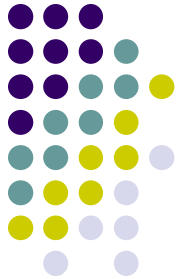
Callable with Batch statements

```
public class MyBatchCallableStatement {
    public static void main(String a[]){
        Connection con = null;
        CallableStatement callSt = null;
        try {
            //Connection codes

            callSt = con.prepareCall("{call myprocedure(?,?)}");
            callSt.setInt(1,200);
            callSt.setDouble(2, 3000);
            callSt.addBatch();
            callSt.setInt(1,130);
            callSt.setDouble(2, 2000);
            callSt.addBatch();
            int[] updateCounts = callSt.executeBatch();
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        } catch (SQLException e) {
            e.printStackTrace();
        } finally{
            try{
                if(callSt != null) callSt.close();
                if(con != null) con.close();
            } catch(Exception ex){}
        }
    }
}
```

Prepared Statement with Rollback

```
public class JDBCSavepointExample {
  public static void main(String[] args) {
     // Connection codes
       conn.setAutoCommit(false);
       String INSERT_SQL = "INSERT INTO employee "+ "(EMP_ID, NAME, DOB) VALUES (?,?,?)";
        try () {
           // Insert 1st record
           insertStmt.setInt(1, 1);
           insertStmt.setString(2, "Michael");
           insertStmt.setDate(3, new Date(dateFormat.parse("1995-07-01").getTime()));
           insertStmt.executeUpdate();
           Savepoint savepoint = conn.setSavepoint();
           insertStmt.setInt(1, 2);
           insertStmt.setString(2, "Manish");
           insertStmt.setDate(3, new Date(dateFormat.parse("1992-01-21").getTime()));
           insertStmt.executeUpdate();
           conn.rollback(savepoint);
           conn.commit();
           System.out.println("Transaction is commited successfully.");
        } catch (SQLException | ParseException e) {
           e.printStackTrace();
           if (conn != null) {
              try {
                 // Roll back transaction
                 System.out.println("Transaction is being rolled back.");
                 conn.rollback();
              } catch (Exception ex) {
                 ex.printStackTrace();
              }
           }
        }
  }
```

Creating mysql database, table and inserting data



- sql>show databases;
- sql>create database mydb;
- sql>use mydb;
- sql>create table admin (username varchar(15), password varchar(15),type varchar(5));
- sql>show tables;
- sql>insert into admin values (‘aa1’,’Abc@123’,’user1’);
- sql>commit;
- sql>select * from admin;