Java Database Connectivity (JDBC)

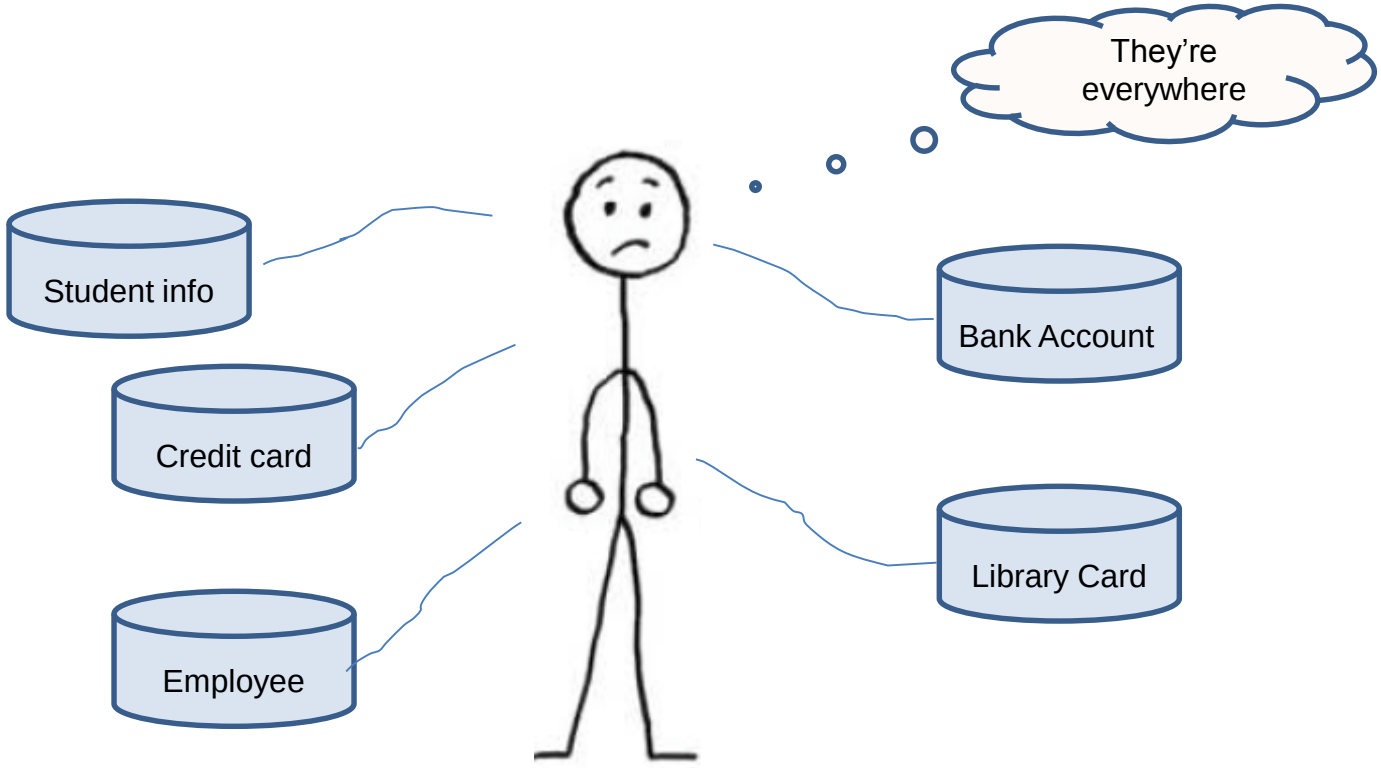
Campus Mind 2011

Objectives

- Overview of JDBC technology
- JDBC drivers
- Basic steps in using JDBC
- Using Statement, PreparedStatement and CallableStatement
- Retrieving data from a ResultSet
- Handling SQL exceptions
- Submitting multiple statements as a transaction
- Good JDBC programming practices.

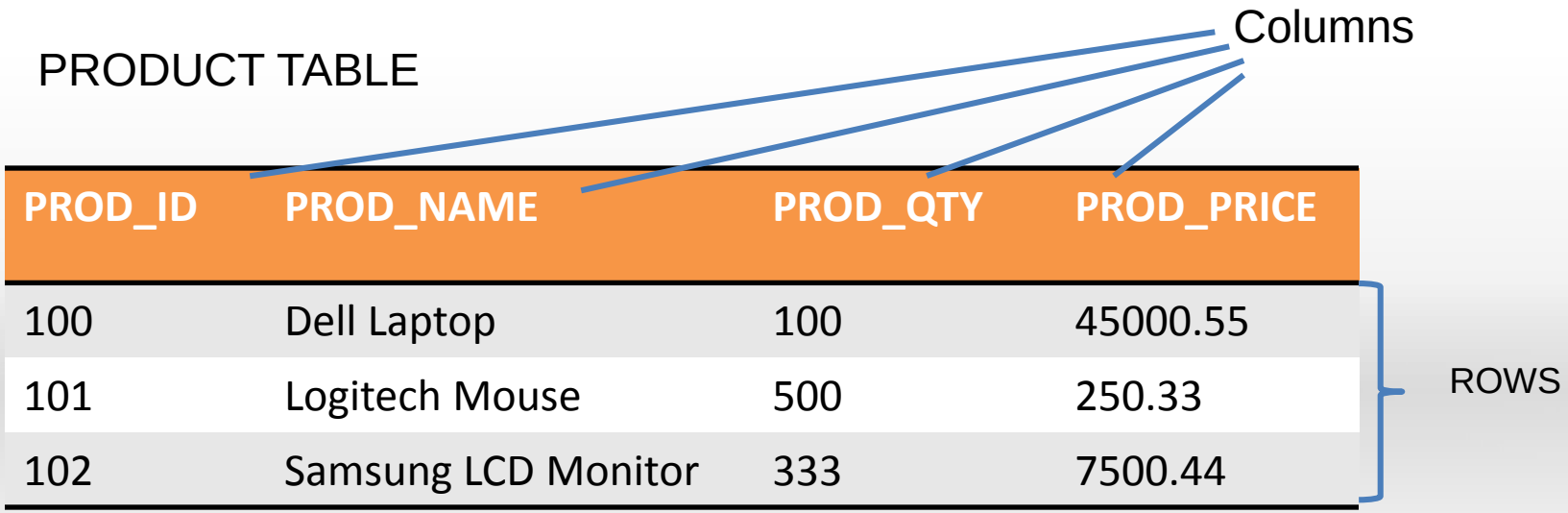
What is a database?

- A database is a container that holds tables and other SQL structures related to that table.



What is a table?

- A table is the structure inside your database that contains data, organized in columns and rows.
- Columns are like your instance variables, correspond to attributes of the class.
- Row contains all the information about one object in your table.



JDBC

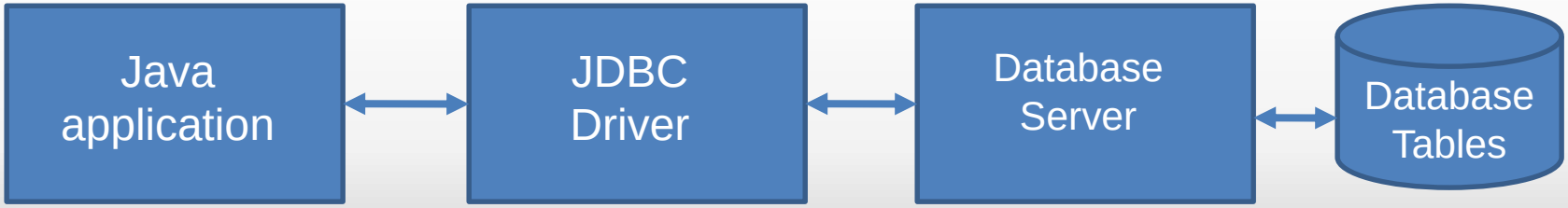
- **JDBC (Java Database Connectivity) provides a standard library for accessing relational databases.**
  - **API standardizes**
    - **Way to establish connection to database.**
    - Approach to instate queries.
  - API does not standardize SQL syntax
- JDBC classes are in java.sql package.

JDBC

- JDBC standardizes
  - Mechanism for connecting to the Database
  - Syntax for sending SQL queries
  - The data structure of query result (table)
- JDBC does not standardize
  - SQL syntax (JDBC is not embedded SQL)
  - Dialects
  - Extensions like OODBMS

JDBC Drivers

- JDBC Drivers are required to access a database from Java application.
- Different databases require different drivers.
- Drivers may be supplied by database vendors or a third party.
- When Java program issues SQL commands, the driver forwards them to the database.

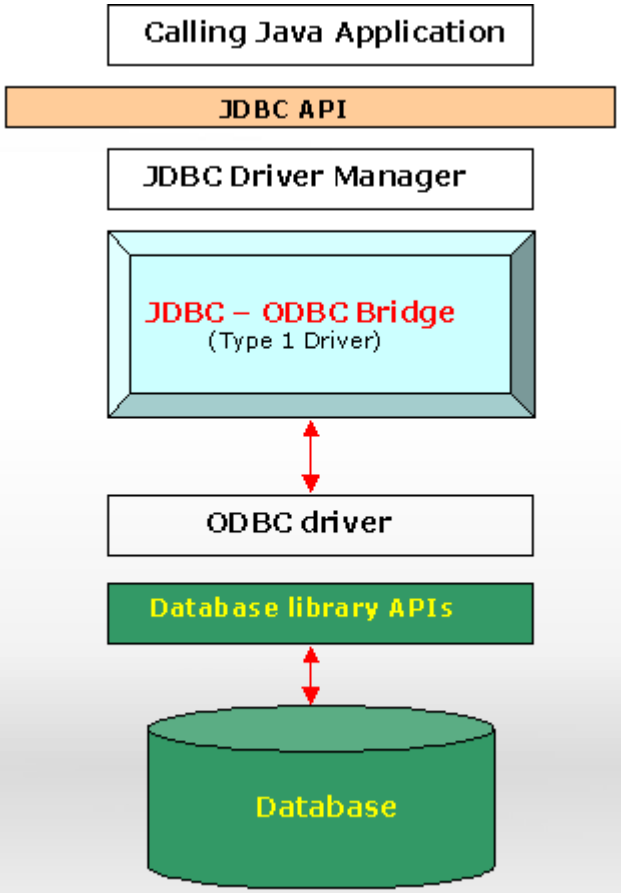


JDBC Driver types

- **JDBC drivers** are divided into four types or levels:
  - **Type 1 Driver - JDBC-ODBC bridge**
  - **Type 2 Driver - Native-API Driver specification**
  - **Type 3 Driver - Network-Protocol Driver**
  - **Type 4 Driver - Native-Protocol Driver**

Type 1 Driver - JDBC-ODBC bridge

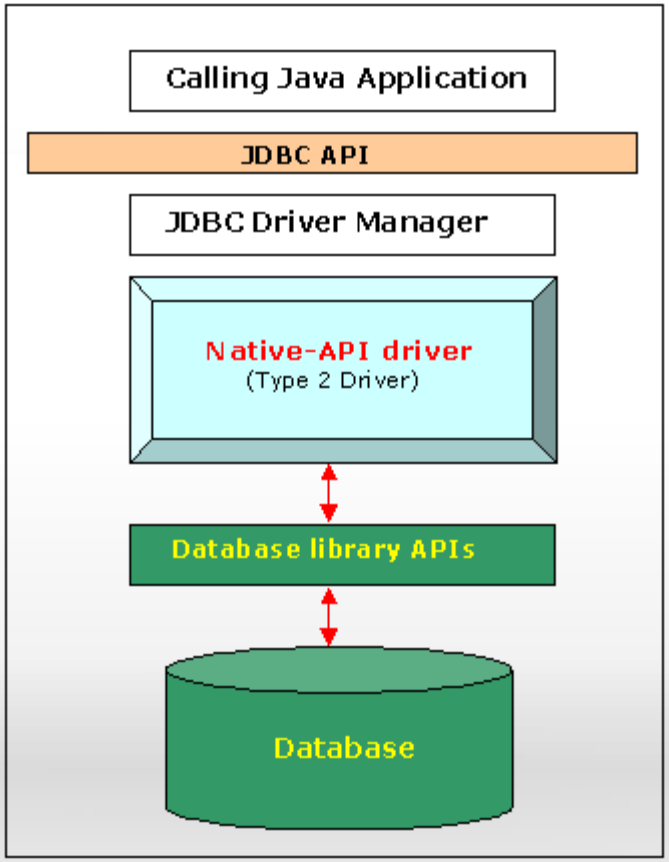
- The driver converts JDBC method calls into ODBC (Open Database Connectivity) function calls.
- Sun (Oracle) provides a JDBC-ODBC Bridge driver -> “sun.jdbc.odbc.JdbcOdbcDriver”.
- **Advantages**
  - Easy to connect.
- **Disadvantages**
  - Performance overhead since every call has to go through the jdbc -odbc bridge and to the ODBC driver, then to the native db connectivity interface.
  - The ODBC driver needs to be installed on the client machine.
  - Compared to other driver types it's slow.



Type 2 Driver - Native-API Driver specification

- Type 2 Drivers uses drivers provided by database vendors. These drivers are known as client-side libraries of the database.
- The Driver is compiled to be used with the particular operating system.

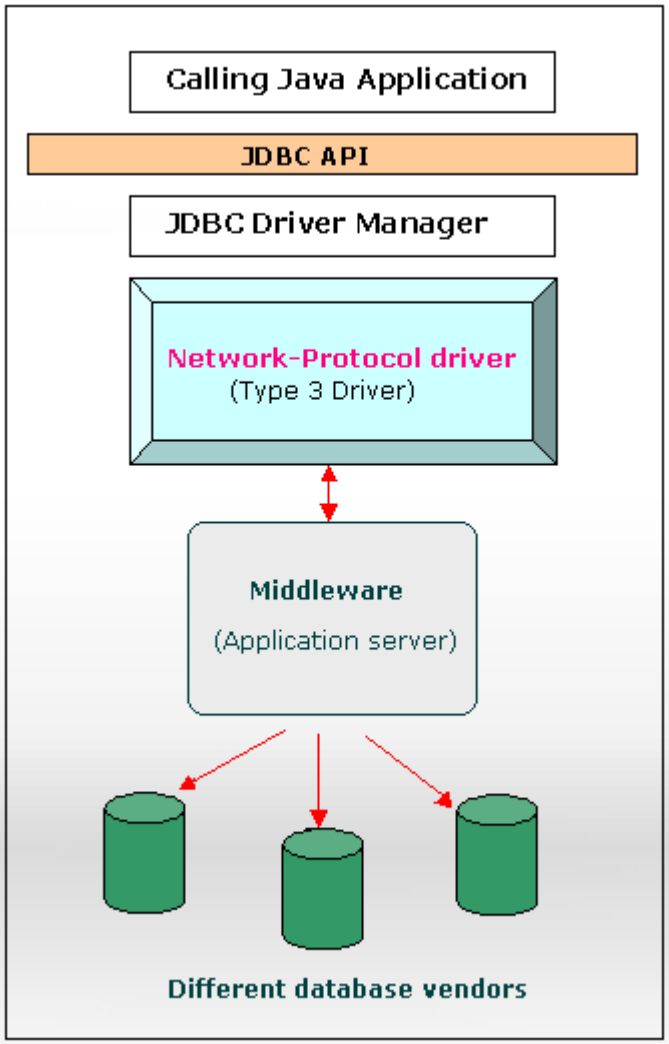
- Disadvantages
  - Not all the databases have a client side library
  - This driver is platform dependent



Type 3 Driver - Network-Protocol Driver

- Type 3 driver makes use of a middle tier between the calling programs and the database. The middle-tier converts JDBC calls into vendor-specific database protocol.

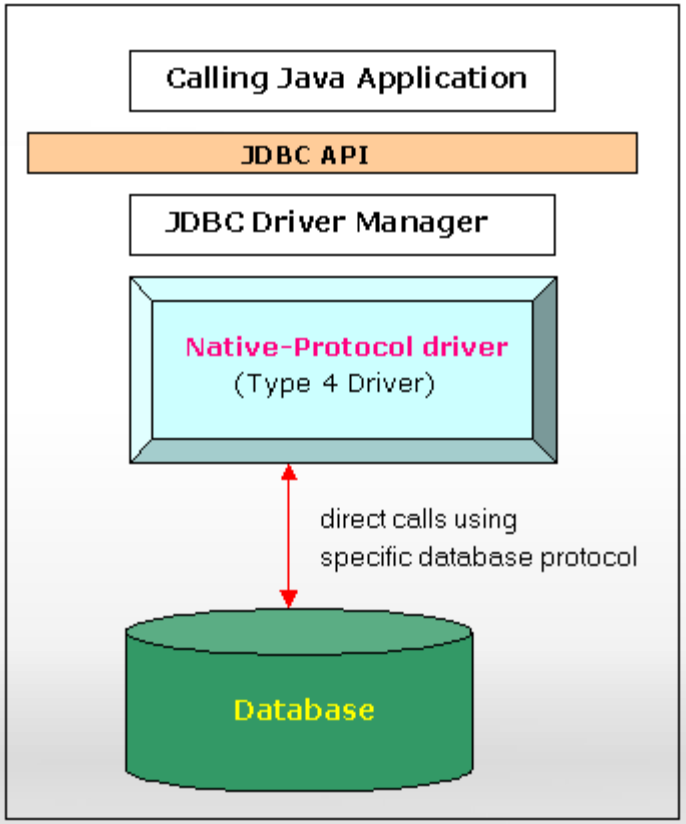
- Advantages
  - No need for Database vendor libraries in client machine.
- Disadvantages
  - An extra layer added between client and database.



Type 4 Driver - Native-Protocol Driver

- The Type 4 driver written completely in Java, is also known as Pure Java Driver.
- This database driver implementation that converts JDBC calls directly into a vendor-specific database protocol. Therefore it is called a THIN driver.

- Advantages:
  - Type 4 drivers are Platform independent



Steps in using JDBC

- Step 1 : Load the Driver
  - Add the Database specific jar file to classpath and load the driver class

```
/*
 * Load the driver classes
 */

 try {
      // for mySQL
      Class.forName("com.mysql.jdbc.Driver");
      // for Oracle
      Class.forName("oracle.jdbc.OracleDriver");
      //for JDBC-ODBC
      Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
} catch (ClassNotFoundException e) {
      /*
       * If driver class not found in classpath
       * or the not spelled properly,
       * ClassNotFoundException is thrown
       */
     e.printStackTrace();
}
```

Step 2: Establish a database connection.

- The getConnection( ) method of DriverManager attempts to select an appropriate driver from the set of registered JDBC drivers and attempts to establish a connection to the given database URL

```
try {
    /* java.sql.Connection is an interface
     * DriverManager's getConnection() returns an instance
     * of mySQL Connection
     */
    Connection con =
            DriverManager.getConnection("jdbc:mysql://localhost:3306/javaDB",
                "root","root");

    /* DriverManager's getConnection() returns an instance
     * of Oracle Connection
     */
    con = DriverManager.getConnection("jdbc:oracle:thin:@localhost:javaDB",
            "scott", "tiger");

} catch (SQLException e) { //if a database access error occurs
    e.printStackTrace();
}
```

Step 3: Send SQL statements

- The java.sql package provides 3 interfaces for sending SQL statements

1. Statement
   - Represents the base statements interface.
   - In terms of efficiency, it is suitable to use *Statement* only when we know that we will not need to execute the SQL query multiple times

2. PreparedStatement
   - Also know as precomplied or parameterized statements are more efficient for multiple executions.
   - Supports only “IN” parameters.

3. CallableStatement.
   - This interface is used for executing the SQL stored procedures
   - Supports “IN”, “OUT” and “INOUT” parameters.

java.sql.Statement interface

- Some important methods:

int **executeUpdate**(String sql)

Executes the given SQL statement, which may be an INSERT, UPDATE, or DELETE statement.

Returns the row count.

int **executeUpdate**(String sql, int autoGeneratedKeys)

Similar to previous method but signals the driver that the auto-generated keys produced by this Statement object should be made available for retrieval.

Returns the row count.

ResultSet **executeQuery**(String sql)

Executes the given SQL statement, which returns a single ResultSet object

ResultSet **getGeneratedKeys**()

Retrieves any auto-generated keys created as a result of executing this Statement object.

Code Snippet: using Statement interface

```
String sql = "insert into PRODUCT" +
          "(PRODUCT_ID,PRODUCT_NAME,PRODUCT_QTY,PRODUCT_PRICE) " +
          " values (122,'Microsoft Mouse',555,3333.33)";

/*
 * java.sql.Statement is an interface
 * Statement object is returned by the Connection object's
 * createStatement().
 * It returns instance of com.mysql.jdbc.StatementImpl
 * if connection is to mySQL,
 * similarly it returns instance of oracle.driver.OracleStatement
 * if connecction is established to Oracle
 */
Statement stmt = con.createStatement();

int rowCount = stmt.executeUpdate(sql);

System.out.println("Rows affected :" + rowCount);
```

Example

- Code Example:
  - Refer: StatementExample.java
    - Illustrates basic steps required to interact with database
    - Illustrates how to use executeUpdate() of java.sql.Statement.
  - Table used for this example is PRODUCT
  - SQL Script:

```
create table PRODUCT
   (PRODUCT_ID int PRIMARY_KEY AUTO INCREMENT,
    PRODUCT_NAME varchar(50) unique not null,
    PRODUCT_QTY int,
    PRODUCT_PRICE double);
```

java.sql.PreparedStatement interface

- PreparedStatement extends java.sql.Statement interface.
- A SQL statement is precompiled and stored in a PreparedStatement object. This object can then be used to efficiently execute this statement multiple times.
- PreparedStatement takes “IN” parameter.
- In the following example of setting a parameter, con represents an active connection:

```
PreparedStatement pstmt =
  con.prepareStatement
      ("UPDATE EMPLOYEES SET SALARY = ? WHERE ID = ?");
pstmt.setBigDecimal(1, 153833.00);
pstmt.setInt(2, 100);
```

Code Snippet: java.sql.PreparedStatement interface

```
String insertProductSQL = "insert into PRODUCT" +
          "(PRODUCT_ID,PRODUCT_NAME,PRODUCT_PRICE) " +
          " values (?,?,?)";
// reader from keyboard
BufferedReader reader =
     new BufferedReader(new InputStreamReader(System.in));
// create prepared statement for given sql.
PreparedStatement psmt = con.prepareStatement(insertProductSQL);
/*
 * insert three records of product
 */
for (int i = 0; i < 3; i++) {
     System.out.println("Enter Product ID :");
     int productId = Integer.parseInt(reader.readLine());
     System.out.println("Enter Product Name :");
     String productName =  reader.readLine() ;
     System.out.println("Enter Product Price :");
     double productPrice = Double.parseDouble(reader.readLine());

     psmt.setInt(1, productId);  // set IN parameters 1--> PRODUCT_ID
     psmt.setString(2, productName); // 2 --> PRODUCT_NAME
     psmt.setDouble(3, productPrice); // 3 --> PRODUCT_PRICE
     psmt.executeUpdate(); // No need to send SQL here
}
```

Example

- Code Example:
  - Refer: PreparedStatementExample.java
    - Illustrates how to use PreparedStatement

Table used for this example is PRODUCT

- SQL Script:

```
create table PRODUCT
   (PRODUCT_ID int PRIMARY_KEY AUTO INCREMENT,
    PRODUCT_NAME varchar(50) unique not null,
    PRODUCT_QTY int,
    PRODUCT_PRICE double);
```

Stored Procedures.

- Stored Procedure: A **stored procedure** is a subroutine available to applications accessing a relational database system.
  - Uses are :
    - Extensive or complex processing that requires the execution of several SQL statements
    - centralize logic for different applications.
- Example of how to create a stored procedure in mySQL.

```
mysql> delimiter $$
mysql> create procedure INSERT_PRODUCT
        (IN pid INT, IN pname VARCHAR(50), IN price DOUBLE)
        BEGIN
                insert into PRODUCT (PRODUCT_ID,PRODUCT_NAME,PRODUCT_PRICE)
                values (pid,pname,price);
         END;
        $$
        Query OK, 0 rows affected (0.00 sec)

mysql> delimiter ;
```

java.sql.Callable interface

- This interface is used to execute SQL stored procedures.
- Usage Syntax:
  - {?= call <procedure-name>[<arg1>,<arg2>, ...]}
  - {call <procedure-name>[<arg1>,<arg2>, ...]}
- IN parameter values are set using the set methods inherited from PreparedStatement.
- OUT parameters must be registered prior to executing the stored procedure; their values are retrieved after execution via the get methods

Code Snippet: using CallableStatement

```
/*
 create procedure INSERT_PRODUCT
    (IN pid INT, IN pname VARCHAR(50), IN price DOUBLE)
    BEGIN
        insert into PRODUCT (PRODUCT_ID,PRODUCT_NAME,PRODUCT_PRICE)
        values (pid,pname,price);
    END;
 */
String insertProcedure = "{call INSERT_PRODUCT(?,?,?)}" ;
/*
 * create an instance of CallableStatement
 */
CallableStatement cstmt = con.prepareCall(insertProcedure);

// set the IN parameters
cstmt.setInt(1, 600);    // set IN parameters 1--> PRODUCT_ID
cstmt.setString(2, "NetGear");  // 2 --> PRODUCT_NAME
cstmt.setDouble(3, 2500.00); // 3 --> PRODUCT_PRICE
cstmt.execute();
```

Code Snippet: using CallableStatement OUT parameter

```
/*
   create procedure GET_SUM_PRICE
    (OUT total INT)
    BEGIN
        select sum(PRODUCT_PRICE) into total from PRODUCT;
    END;
 */
String getSumProcedure = "{call GET_SUM_PRICE(?)}";
CallableStatement cstmt = con.prepareCall(getSumProcedure);
// Register OUT Parameter
cstmt.registerOutParameter(1, java.sql.Types.BIGINT);
cstmt.execute();
// get the value for OUT parameter
long totalPrice = cstmt.getLong(1);
System.out.println("Total of Product price : " + totalPrice);
```

Example

- Code Example:
  - Refer: CallableStatementExample.java
    - Illustrates how to call stored procedures and to set “IN” parameters , register “OUT” parameters

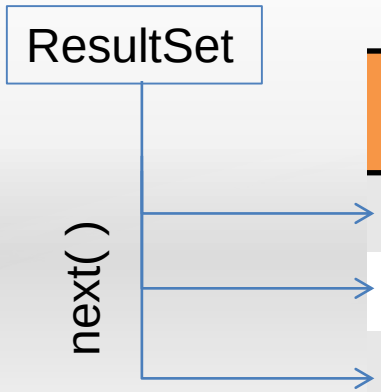
Create the following stored procedures required by this example to check “IN” and “OUT” parameters .

```
create procedure INSERT_PRODUCT
     (IN pid INT, IN pname VARCHAR(50), IN price DOUBLE)
    BEGIN
          insert into PRODUCT (PRODUCT_ID,PRODUCT_NAME,PRODUCT_PRICE)
             values (pid,pname,price);
     END;

create procedure GET_SUM_PRICE (OUT total INT)
     BEGIN
         select sum(PRODUCT_PRICE) into total from PRODUCT;
   END;
```

ResultSet

- ResultSet is a table of data representing a database result set, which is usually generated by executing a statement that queries the database.
- A ResultSet object maintains a cursor pointing to its current row of data. Initially the cursor is positioned before the first row.
- The boolean next( ) method moves the cursor to the next row, and because it returns false when there are no more rows in ResultSet Object.



PROD_ID	PROD_NAME	PROD_QTY	PROD_PRICE
100	Dell Laptop	100	45000.55
101	Logitech Mouse	500	250.33
102	Samsung LCD Monitor	333	7500.44

ResultSet methods

Some important **getXXX**( ) methods or ResultSet

SQL data type	Java Type	Get Method
CHAR	String	getString( )
VARCHAR	String	getString( )
BIT	Boolean	getBoolean( )
TINYINT	Byte	getByte()
INTEGER	Integer	getInt()
DOUBLE	Double	getDouble()
DATE	java.sql.Date	getDate()

Code Snippet : using ResultSet

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

  String sql =
       "select PRODUCT_ID,PRODUCT_NAME,PRODUCT_QTY, PRODUCT_PRICE" +
       " from PRODUCT";
  /*
   * get a cursor to the database result set retreived
   * using a Statement object.
   */
  ResultSet productResultSet = stmt.executeQuery(sql);

  while(productResultSet.next()) {
       int productId = productResultSet.getInt("PRODUCT_ID");
       String productName = productResultSet.getString("PRODUCT_NAME");
       int productQty = productResultSet.getInt("PRODUCT_QTY");
       double productPrice = productResultSet.getDouble("PRODUCT_PRICE");

       System.out.println(productId +","
                          + productName + ","
                          + productQty + ","
                          + productPrice);

  }
```

Example

- Code Example:
  - Refer: ResultSetExample.java
    - Illustrates how to traverse through the retrieved results from database

Retrieving Automatically Generated Keys

- JDBC 3.0's auto generated keys feature provides a way to retrieve values from columns that are part of an index or have a default value assigned.

- mySQL supports the auto increment feature, which allows users to create columns in tables for which the database system automatically assigns increasing integer values.

```
create table PRODUCT
    ( PRODUCT_ID int PRIMARY KEY AUTO_INCREMENT,
    PRODUCT_NAME varchar(50) unique not null,
    PRODUCT_QTY int,
    PRODUCT_PRICE double );
```

- *Statement.getGeneratedKeys( )* can be called to retrieve the value of such a column.

Code Snippet: Auto generated keys.

```
String insertProductSQL = "insert into PRODUCT"
        + "(PRODUCT_NAME,PRODUCT_PRICE) values (?,?)";

/*
 * The second parameter to Statement is to make all generated keys
 * retrievable
 */
PreparedStatement pstmt = con.prepareStatement(insertProductSQL,
        Statement.RETURN_GENERATED_KEYS);

pstmt.setString(1, "Timex Watch"); // 1 --> PRODUCT_NAME
pstmt.setDouble(2, 7500.00); // 2 --> PRODUCT_PRICE
pstmt.executeUpdate();

ResultSet rs = pstmt.getGeneratedKeys();
if (rs.next()) {
    int productId = rs.getInt(1); // get AUTO INCREMENT column
                                  // "PRODUCT_ID"
    System.out.println("Product with ID: " + productId + " added");
}
```

Transaction

- A transaction is a set of one or more statements that is executed as a unit, so either all of the statements are executed, or none of the statements is executed.

- Example: Assume you need to transfer funds between two accounts 230 and 500, it involve two updates
  - Withdraw
    - Update Account set balance = balance – amount where acc_no = 230.
  - Deposit
    - Update Account set balance = balance + amount where acc_no = 500.
  - Both these statements has to execute or none.

JDBC Transaction

- Disable Auto-Commit mode.
  - con.setAutoCommit(false);
- Commit transaction.
  - After the auto-commit mode is disabled, no SQL statements are committed until you call the method commit() explicitly.
- Rollback transaction.
  - calling the method rollback() terminates a transaction and returns any values that were modified to their previous values.

Code Snippet: JDBC transaction

```
// Disable auto commit
con.setAutoCommit(false);
try {
    // set of SQL statements


    // If all the SQL statements are executed commit
    con.commit();
} catch (SQLException e) {
    // if any SQLException occurs rollback all
    con.rollback();
    // tell the user about failure
    throw new DaoException("Transaction failed ", e);
}
```

Example :Complete DAO

- Code Example: ShoppingExample.zip
  - Code contains complete java project
  - Illustrates database application using DAO pattern.
  - Steps
    - Create database
    - Run sql's provided in shopDB.sql
    - Modify "database.properties" as applicable
    - Execute Client.java
  - Things to observe
    - Modularity
    - Exception funneling
    - Program to interface
    - Association between entity classes
    - Transaction
      - Inserting Order and all line items should be a single transaction
    - Using java.sql.Date
  - Add reporting functionality to this application
    - List Orders placed between two dates.
      - It should also display item details

Core JDBC Best Practices.

- Always close Statements, ResultSet, and Connection:
  - This practice involves always closing JDBC objects in a **finally** block to avoid resource limitations found in many databases.
- Consolidate formation of SQL statement strings:
  - This practice involves placing the SQL statement text in a field that is declared static final to reduce string processing as well as make SQL statements easy to identify and read.
- Use Date, Time, and Timestamp objects for temporal fields (avoid using strings):
  - This practice eliminates conversion overhead in the database and often the application.
- Always specify a column list with a select statement (avoid "select *") and with an insert statement:
  - This practice insulates your code against tuning activities of database administrators.

Other JDBC Best practices

- ***Query Fetch Size for Large Result Sets***
  - The fetch size is the number of rows physically retrieved from the database at one time by the JDBC driver as you scroll through a query ResultSet with next().
  - For example, you set the query fetch size to 100. When you retrieve the first row, the JDBC driver retrieves the first 100 rows (or all of them if fewer than 100 rows satisfy the query).

    When you retrieve the second row, the JDBC driver merely returns the row from local memory - it doesn't have to retrieve that row from the database.

    This feature improves performance by reducing the number of calls to the database.
  - Performance increases vary depending upon the speed of the network and also the fetch size should be based on your JVM heap memory setting

Other JDBC Best practices.

- Connection Pooling
  - Connection pooling provides a way for database connections to be established separately before your application needs them. They are used and reused throughout the lifetime of application.
  - It eliminates the latency involved in creating a connection and closing those resources.
- Batch Updates
  - Consider an baking application, if you are transferring funds between two accounts it involves. Two updates on accounts and one insert into transaction table. All these statements can be treated as a batch and submitted to database for processing as a batch.
  - Refer : http://goo.gl/nCLsQ
- Stored Procedures
  - Check how many network transmissions will be saved by calling a stored procedure.
  - Stored procedures are not portable. Also not all databases support stored procedures.

Quiz!

Some quiz questions to reinforce the classroom learning

Quiz questions

- How do you load database drivers?
- Which of the following supports IN parameters?
  - Connection
  - Statement
  - PreparedStatement
  - CallableStatement.
- What is the return type of executeUpdate( ) and what does it represent?
- Which method of Statement should be used to invoke “SELECT” SQL statements?

Quiz questions

- When do you use CallableStatement?
- Do we need to call commit after executing “INSERT”, “DELETE” or “UPDATE” sql statements from java.
- State True or False:
  - ResultSet can be used to read information from a single table only.

JDBC Metadata

DatabaseMetaData can be

- Used to Discover database schema and catalog information.
- Discover database users, tables, views, and stored procedures.
- Understand and analyze the result sets returned by SQL queries.
- Find out the table, view, or column privileges.
- Determine the signature of a specific stored procedure in the database.
- Identify the primary/foreign keys for a given table.

ResultSetMetadata can be used to

- Get information about types and properties of columns in a ResultSet object.

Some methods of DatabaseMetaData.

Method Name	***Description***
getSchemas()	*Schemas provide a logical classification of database objects (tables, views, aliases, stored procedures, user-defined types, and triggers) in an RDBMS.*
getTables(catalog, schema, tableNames, columnNames)	*Returns table names for all tables matching tableNames and all columns matching columnNames.*
getColumns(catalog, schema, tableNames,columnNames)	*Returns table column names for all tables matching tableNames and all columns matching columnNames.*
getURL()	*Gets the name of the URL you are connected to.*
getDriverName()	*Gets the name of the database driver you are connected to.*

Refer : http://goo.gl/Cz5F9

Some important methods of ResultSetMetaData

Method Name	Description
String getColumnLabel(int column)	Gets the designated column's suggested title for use in printouts and displays
String getColumnName(int column)	Get the designated column's name.
int getColumnType(int column)	Retrieves the designated column's SQL type.
boolean isAutoIncrement(int column)	Indicates whether the designated column is automatically numbered, thus read-only
int getColumnCount()	Returns the number of columns in this ResultSet object.

Refer: http://goo.gl/BbsN1

Explore More!!

Never let your curiosity die!

Advanced Topic

- Using JDBC Updateable ResultSet
  - http://goo.gl/PFeET
- How to traverse back word's using ResultSet
  - http://goo.gl/PFeET
- RowSet
  - How to view results in disconnected mode
  - http://goo.gl/23b3P
- Explore how to connect to database using JDBC:ODBC driver.
  - Check if it supports concurrent access.

References

Contains the reference that will supplement the self learning and will be needed for completing the assignments & practice questions

References

- JDBC Tutorial
  - http://java.sun.com/developer/Books/JDBCTutorial/
- **Core J2EE Patterns - Data Access Object**
  - http://java.sun.com/blueprints/corej2eepatterns/Patterns/DataAccessObject.html
- **Isolation Levels and Concurrency**
  - http://db.apache.org/derby/manuals/develop/develop71.html
- JDBC Code Examples
  - http://goo.gl/n70cI