Unit 5

Database Connectivity with JAVA 5hrs

5.1 JDBC Architecture

Database Connectivity (JDBC):

A database is an organized collection of data. There are many different strategies for organizing data to facilitate easy access and manipulation. A database management system (DBMS) provides mechanisms for storing, organizing, retrieving and modifying data from any users. DBMS allows for the access and storage of data without concern for the internal representation of data.

Java Database Connectivity (JDBC) is an API (Application Programming Interface) for java that allows the Java programmer to access the database. The JDBC API consists of a number of classes and interfaces, written in java programming language, which provides a number of methods for updating and querying a data in a database. It is a relational database-oriented driver. It makes a bridge between java application and database. Java application utilizes the features of database using JDBC.

JDBC helps to write Java applications that manage these three programming activities:

- Connect to a data source, like a database.
- Send queries and update statements to the database.
- Retrieve and process the results received from the database in answer to our query.

JDBC Components:

The JDBC API:

The JDBC API provides programmatic access to relational data from the Java programming language. Using JDBC API, front end java applications can execute query and fetch data from connected database. JDBC API can also connect with multiple applications with same database or same application with multiple databases which can resides in different computers (distributed environment). The JDBC API is part of the Java platform, which includes the Java Standard Edition (Java SE) and the Java Enterprise Edition (Java EE). The JDBC 4.0 API is divided into two packages: java.sql and javax.sql. Both packages are included in the Java SE and Java EE platforms.

JDBC Driver Manager:

The JDBC DriverManager class defines objects which can connect Java applications to a JDBC driver. Driver Manager manages all the JDBC Driver loaded in client's system. Driver Manager loads the most appropriate driver among all the Drivers for creating a connection.

JDBC Test Suite:

The JDBC driver test suite helps us to determine that JDBC drivers will run our program. JDBC Test Suite is used to check compatibility of a JDBC driver with J2EE platform. It also check whether a Driver follow all the standard and requirements of J2EE Environment.

JDBC-ODBC Bridge:

The Java Software Bridge provides JDBC access via ODBC Bridge (or drivers). JDBC Bridge is used to access ODBC drivers installed on each client machine. The JDBC Driver contacts the ODBC Driver for connecting to the database. When Java first came out, ODBC 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 ODBC Driver is already installed or come as default driver in windows. In Windows „Datasource" name can be created using control panel >administrative tools>Data Sources (ODBC). After creating 'data source', connectivity of 'data source' to the 'database' can be checked. Using this „data source" , we can connect JDBC to ODBC.

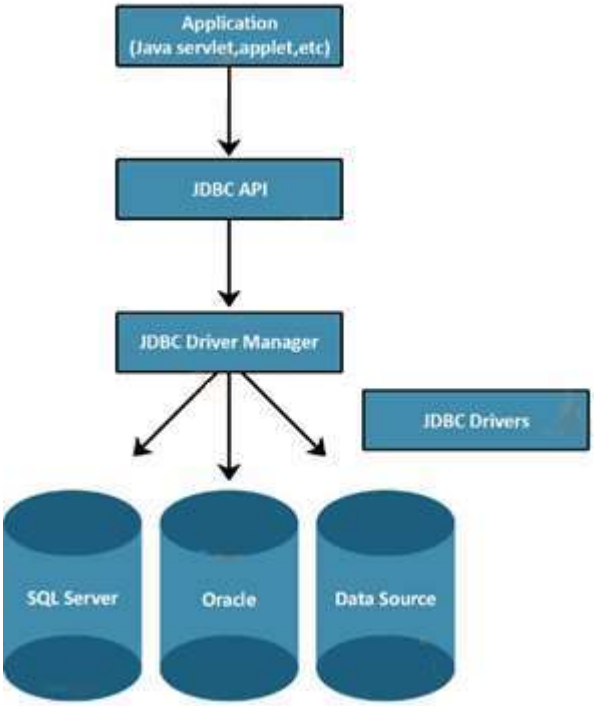


Figure 1 Fig: JDBC Architecture

5.2 JDBC Driver Types and Configuration

JDBC Driver Types

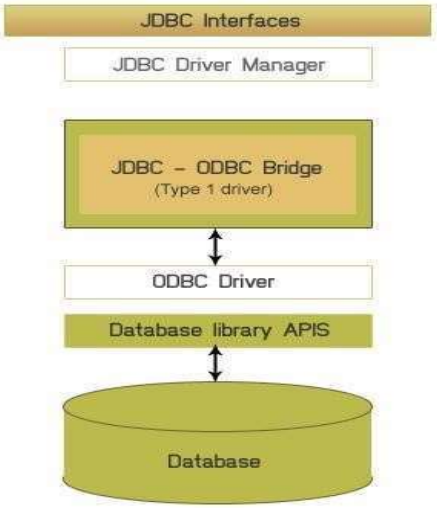
JDBC drivers are classified into the following four categories:

JDBC-ODBC BRIDGE DRIVER (TYPE 1)

Features

- Convert the query of JDBC Driver into the ODBC query, which in return pass the data.
- The bridge requires deployment and proper configuration of an ODBC driver.
- JDBC-ODBC is native code not written in java.
- Nowadays in most cases it is only being used for the educational purposes.

- The connection occurs as follows -- Client -> JDBC Driver -> ODBC Driver -> Database.



Pros

- A type-1 driver is easy to install and handle.

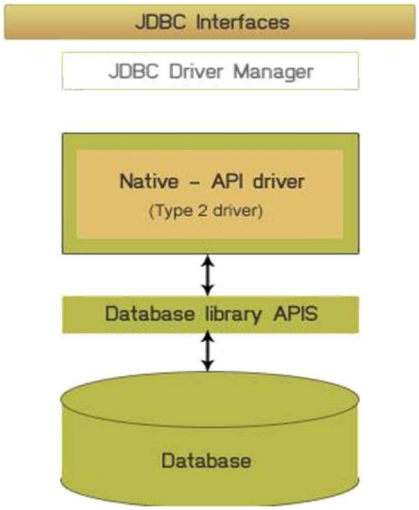
Cons

- Extra channels in between database and application made performance overhead.
- Needs to be installed on client machine.
- Not suitable for applet, due to the installation at clients end.

Native-API (Type-2)

Driver Features

The type 2 driver need libraries installed at client site. For example, we need "mysqlconnector.jar" to be copied in library of java kit. It is not written in java entirely because the non-java interfaces have the direct access to database. It is written partly in Java and partly in native code. When we use such a driver, we must install some platform-specific code in addition to a Java library. These drivers are 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.



Pros

- Type 2 driver has additional functionality and better performance than Type 1.
- Has faster performance than type 1,3 and 4, since it has separate code for native APIS.

Cons

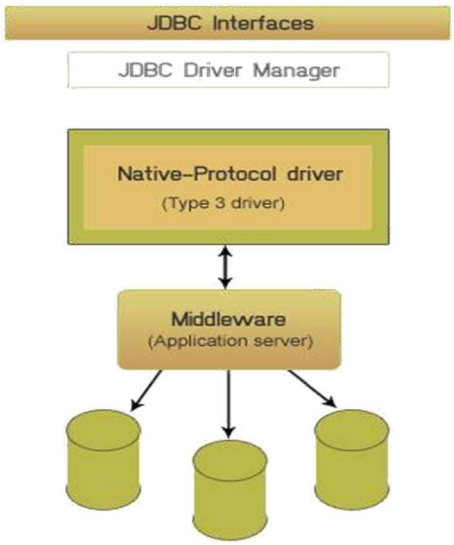
- Library needs to be installed on the client machine.
- Due to the client-side software demand, it can't be used for web-based application.
- Platform dependent.
- It doesn't support "Applets".

Network-Protocol (Type 3) driver

Features

- It is also known as JDBC-Net pure Java.
- It has 3-tier architecture.
- It can interact with multiple database of different environment.
- The JDBC Client driver written in java communicates with a middleware-net-server using a database independent protocol, and then this net server translates this request into database

  commands for that database.
- The connection occurs as follows--Client -> JDBC Driver -> Middleware-Net Server -> Any Database.



Pros

- The client driver to middleware communication is database independent.
- Can be used in internet since there is no client side software needed.
- 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.

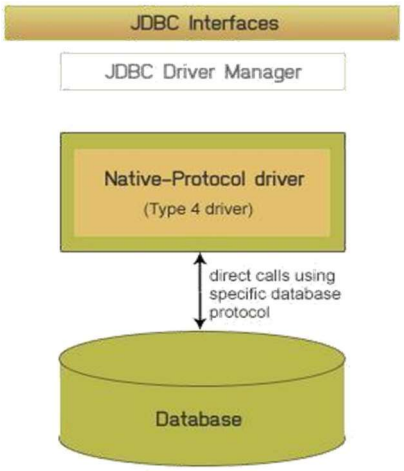
Cons

- Needs specific coding for different database at middleware.
- Due to extra layer in middle can result in time-delay.

Native Protocol (Type 4) Driver

Features

Also known as Direct to Database Pure Java Driver .It is entirely in java. It interacts directly with database generally through socket connection. It is platform independent. It directly converts driver calls into database protocol calls. MySQL's Connector/J driver is an example of Type 4 driver.



Pros

- Improved performance because no intermediate translator like JDBC or middleware server.
- All the connection is managed by JVM, so debugging is easier.

5.3 Managing Connections and Statements

Creating JDBC Application:

There are following six steps involved in building a JDBC application:

- **Import the packages:** We should include the packages containing the JDBC classes needed for database programming. We use, ***import java.sql.*.***

- **Register (Load) the JDBC driver:** We should initialize a driver so that we can open a communications channel with the database. We can load the driver class by calling Class.forName() with the Driver class name as an argument. Once loaded, the Driver class creates an instance of itself.
  Syntax: Class.forName (String ClassName)
  Example: `Class.forName("com.mysql.cj.jdbc.Driver");`

- **Creating a jdbc Connection:** We can use the DriverManager.getConnection() method to create a Connection object, which represents a physical connection with the database. The JDBC DriverManager class defines objects which can connect Java applications to a JDBC driver. DriverManager class manages the JDBC drivers that are installed on the system and getConnection() method is used to establish a connection to a database. This method uses a jdbc url and username, password (not compulsory in every DBMS) to establish a connection to the database and returns a connection object.

Syntax: Connection con=DriverManager.getConnection(url,username,password);

```
Connection cn =
 DriverManager.getConnection("jdbc:mysql://localhost:3306/mysql","root","");
```

- **Creating a jdbc Statement object:** Once a connection is obtained we can interact with the database. To execute SQL statements, we need to instantiate a Statement object from our connection object by using the createStatement() method.
  Syntax: Statement statement = con.createStatement();
  A statement object is used to send and execute SQL statements to a database.

- **Executing a SQL statement with the Statement object, and returning a jdbc ResultSet:** Object of type Statement is used for building and submitting an SQL statement to the database. The Statement class has three methods for executing statements i.e. executeQuery(), executeUpdate(), and execute(). For a SELECT statement, executeQuery() method is used . For statements that create or modify tables, executeUpdate() method is used. execute() executes an SQL statement that is written as String object.

  ResultSet provides access to a table of data generated by executing a Statement. The table rows are retrieved in sequence. A ResultSet maintains a cursor pointing to its current row of data. The next() method is used to successively step through the rows of the tabular results.

- **Clean up the environment:** We should explicitly close all database resources after the task is complete.
  conn.close();

Sample Code:

```
//STEP 1. Import required packages
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

import javax.swing.JOptionPane;
public class FirstExample {
     static final String JDBC_DRIVER = "com.mysql.cj.jdbc.Driver";
     static final String DB_URL = "jdbc:mysql://localhost:3306/bca_db";
     private static final String USER_NAME = "root";
     private static final String PASSWORD = "";

     public static void main(String[] args) {
          Connection conn = null;
          Statement stmt = null;
          try{
               //STEP 2: Register JDBC driver
               Class.forName(JDBC_DRIVER);
```

```
                //STEP 3: Open a connection
                System.out.println("Connecting to database...");
                conn =
DriverManager.getConnection(DB_URL,USER_NAME,PASSWORD);

                //STEP 4: create a statement object
                System.out.println("Creating statement...");
                stmt = conn.createStatement();

                //STEP 5:Execute a SQL statement with the Statement
object, and returning a jdbc resultSet
                String sql = "SELECT id, first, last, age FROM Employees";
                ResultSet rs = stmt.executeQuery(sql);

                //Extract data from result set
                while(rs.next()){
                      //Retrieve by column name
                      int id = rs.getInt("id");
                      int age = rs.getInt("age");
                      String first = rs.getString("first");
                      String last = rs.getString("last");

                      //Display values
                      System.out.print("ID: " + id);
                      System.out.print(", Age: " + age);
                      System.out.print(", First: " + first);
                      System.out.println(", Last: " + last);
                }

                //STEP 6: Clean-up environment
                rs.close();
                stmt.close();
                conn.close();
           }catch(SQLException se){
                      JOptionPane.showMessageDialog(null, "SQL exception
caught");
           }catch(Exception e){
                      JOptionPane.showMessageDialog(null, "exception caught");
           }
      }//end main
}//end FirstExample
```

OUTPUT:

```
Connecting to database...
Creating statement...
ID: 100, Age: 18, First: Sunil, Last: Bist
```

ID: 101, Age: 25, First: Ravi, Last: Khadka
ID: 102, Age: 30, First: Pusp, Last: Joshi
ID: 103, Age: 28, First: Harendra, Last: Shah

The Statement:

The Statement object is used for general-purpose access to our database. It represents the base statements interface. It is suitable to use the Statement only when we know that we will not need to execute the SQL query multiple times. The Statement interface cannot accept parameters. In contrast to PreparedStatment, the Statement does not offer support for the parameterized SQL queries. Before we can use a Statement object to execute a SQL statement, we need to create the object using the Connection object's createStatement( ) method. Statements would be suitable for the execution of the DDL statements such as CREATE, ALTER, DROP

5.4 Result Sets and Exception Handling

ResultSet

A ResultSet object 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 can be created through any object that implements the Statement interface, including PreparedStatement, CallableStatement, and RowSet.

We access the data in a ResultSet object through a cursor. This cursor is a pointer that points to one row of data in the ResultSet. Initially, the cursor is positioned before the first row. The method **ResultSet.next()** moves the cursor to the next row. This method returns false if the cursor is positioned after the last row. This method repeatedly calls the ResultSet.next() method with a while loop to iterate through all the data in the ResultSet.

ResultSet Interface

The ResultSet interface provides methods for retrieving and manipulating the results of executed queries, and ResultSet objects can have different functionality and characteristics. These characteristics are **type, concurrency, and cursor holdability.**

Types of ResultSet

The type of a ResultSet object determines the level of its functionality in two areas: the ways in which the cursor can be manipulated, and how concurrent changes made to the underlying data source are reflected by the ResultSet object.
There are generally three different ResultSet types:

- **TYPE_FORWARD_ONLY:** The result set cannot be scrolled; its cursor moves forward only, from before the first row to after the last row. The rows contained in the result set depend on how the underlying database generates the results. That is, it contains the rows that satisfy the query at either the time the query is executed or as the rows are retrieved.

- **TYPE_SCROLL_INSENSITIVE:** The result set can be scrolled; its cursor can move both forward and backward relative to the current position, and it can move to an absolute position. The result set is insensitive to changes made to the underlying data source while it is open. It contains the rows that satisfy the query at either the time the query is executed or as the rows are retrieved.

- **TYPE_SCROLL_SENSITIVE:** The result set can be scrolled; its cursor can move both forward and backward relative to the current position, and it can move to an absolute position. The result set reflects changes made to the underlying data source while the result set remains open.

The default ResultSet type is TYPE_FORWARD_ONLY.

Note: Not all databases and JDBC drivers support all ResultSet types. The method DatabaseMetaData supports ResultSetType returns true if the specified ResultSet type is supported and false otherwise.

ResultSetMetadata

The metadata describes the ResultSets contents. Programs can use metadata programmatically to obtain information about the ResultSets column names and types. ResultSetMetaData method getColumnCount is used to retrieve the number of columns in the ResultSet.

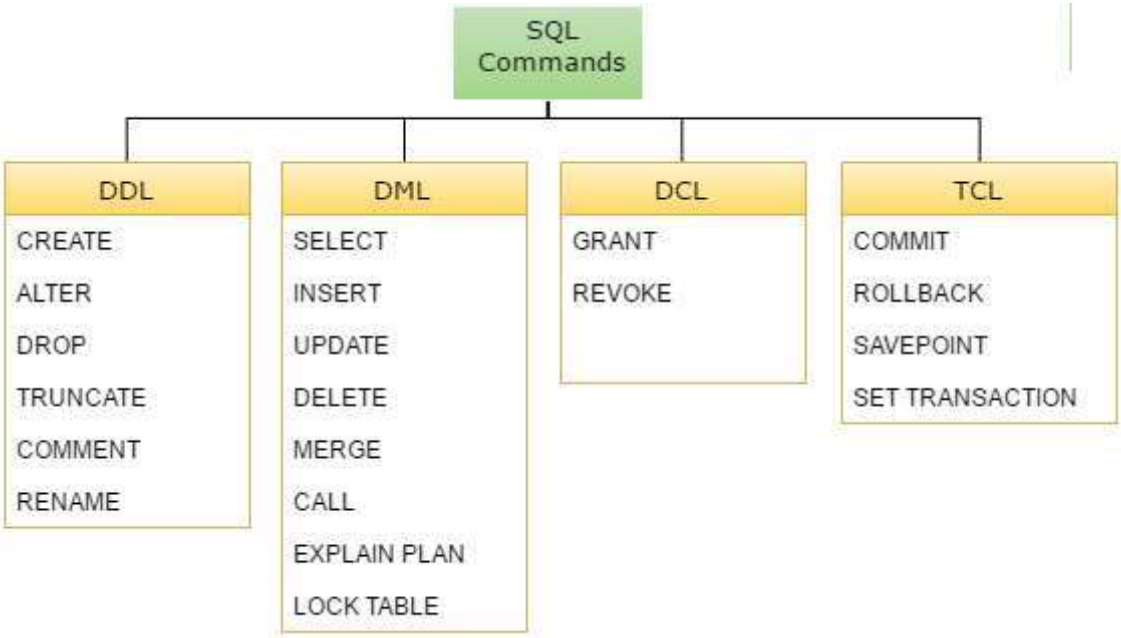
Exception Handling

All the database related SQLException and other exception use try…catch block as shown in the code below:

```
try{
      //code goes here related DDL and DML operation


}catch(ClassNotFoundException e){
      System.out.println("Driver not found:"+e);
}
catch(SQLException e) {
      System.out.println("SQL Exception occurred:"+e);
} catch(Exception e) {
      System.out.println("Exception occurred: "+e);
}
```

5.5 DDL and DML Operations



DDL Commands

CREATE: - Create database or its objects (table, index, function, views, store procedure, and triggers)

Syntax:

=>CREATE DATABASE databasename;

=>CREATE TABLE table_name (column1 data_type, column2 data_type, ...);

For example,

=>CREATE DATABASE be_computer;

=>CREATE TABLE tbl_students(id integer primary key auto_increment, first_name varchar(20) not null, last_name varchar(20));

DROP: - Delete Database or objects from the database

Syntax:

=> DROP DATABASE be_computer;

=> DROP TABLE table_name;

For example,

=> DROP DATABASE be_computer;

=> DROP TABLE tbl_students;

ALTER: - Alter the structure of the database objects

Syntax:

=>ALTER TABLE table_name ADD COLUMN column_name data_type;

For example,

=>ALTER TABLE tbl_students ADD COLUMN email varchar(50) unique;

TRUNCATE: - Remove all records from a table, including all spaces allocated for

the records are removed
Syntax:
=>TRUNCATE TABLE table_name;

For example,
=> TRUNCATE TABLE tbl_students;
RENAME: - Rename an object existing in the database
Syntax:
=> RENAME TABLE old_table_name TO new_table_name;

For example,
=> RENAME TABLE tbl_students TO students;

DQL Commands
SELECT: - It is used to retrieve data from the database
Syntax:
=> SELECT * FROM table_name;
OR
=> SELECT column1, column2, ...FROM table_name WHERE condition;

For example,
=> SELECT * FROM students;
=> SELECT * FROM students WHERE id=101
INSERT: - Insert data into a table
Syntax:
=>INSERT INTO table_name (column1, column2, ...) VALUES (value1, value2, ...);

For example,
=>INSERT INTO students(first_name, last_name, email) VALUES
('sunil','Bist','sunil@nast.edu.np');

UPDATE:- Update existing data within a table
Syntax:
=>UPDATE table_name SET column1 = value1, column2 = value2 WHERE
condition;

For example,
=>UPDATE students SET first_name='SUNIL', last_name='BIST'
WHERE email='sunil@nast.edu.np'

DELETE: - Delete records from a database table
Syntax:
=>DELETE FROM table_name WHERE condition;

For example,
=> DELETE FROM students; //delete all records
=>DELETE FROM students WHERE id=‘2’ //delete record having id=‘2’

DDL Commands
=> CREATE TABLE:-
CREATE TABLE employees (employee_id INT PRIMARY KEY, first_name VARCHAR(50), last_name VARCHAR(50));

=> ALTER TABLE:-
ALTER TABLE employees ADD COLUMN phone VARCHAR(20);
=> DROP TABLE:-
DROP TABLE employees;

DML Commands
INSERT:-
INSERT INTO employees (first_name, last_name, email) VALUES (‘Sunil', ‘Bist', ‘info@sunilbist.com’);

UPDATE:-
UPDATE employees SET email = ‘sunil@nast.edu.np' WHERE first_name = ‘Sunil' AND last_name = ‘Bist';
DELETE:-
DELETE FROM employees WHERE employee_id = 101;
DQL and Other Commands
SELECT:-
SELECT * FROM employees;
WHERE:-
SELECT * FROM employees WHERE department = ‘BCA';
ORDER BY:
SELECT * FROM employees ORDER BY hire_date DESC;
JOIN:-
SELECT e.first_name, e.last_name, d.department_name FROM employees e JOIN departments d ON e.department_id = d.department_id;

CRUD Operations in Database:

The operations such as insertion, creation, deletion, updating and selection of the tabular data are known as CRUD operation in database. Following are the CRUD operations with the appropriate examples;

INSERT OPERATION:

Example 1: Program that insert data in the student table through java

```
import java.sql.*;
class TestJdbc{
	static final String JDBC_DRIVER = "com.mysql.cj.jdbc.Driver";
	static final String DB_URL = "jdbc:mysql://localhost:3306/bca_db";
	private static final String USER_NAME = "root";
	private static final String PASSWORD = "";
	public static void main(String arg[]){
		try{
			Class.forName(JDBC_DRIVER);
			Connection cn =
DriverManager.getConnection(DB_URL,USER_NAME,PASSWORD);
			String str = "insert into student values(12,'Ram',
'dhangadi')";
			Statement stat = cn.createStatement();
			stat.executeUpdate(str);
			cn.close();
		}catch(ClassNotFoundException e){
			System.out.println("Cannot insert into the table");
		}
		catch(SQLException e) {

		}
	}
}
```

Example2: Java program that is used to insert multiple records in a table Customers

```
import java.sql.*;
class JdbcInsertMultipleRows {
	static final String JDBC_DRIVER = "com.mysql.cj.jdbc.Driver";
	static final String DB_URL = "jdbc:mysql://localhost:3306/bca_db";
	private static final String USER_NAME = "root";
	private static final String PASSWORD = "";
	public static void main(String arg[]){
		try{
			Class.forName(JDBC_DRIVER);
			Connection cn =
DriverManager.getConnection(DB_URL,USER_NAME,PASSWORD);
			Statement st = cn.createStatement();
			st.executeUpdate("INSERT INTO Customers " + "VALUES (1001,
'Ram','Kathmandu', 2001)");
			st.executeUpdate("INSERT INTO Customers " + "VALUES
(1002,'Shyam','Pokhara', 2004)");
			st.executeUpdate("INSERT INTO Customers " + "VALUES (1003,
'Hari','Nepalgung', 2003)");
			st.executeUpdate("INSERT INTO Customers " +
"VALUES(1004,'Krishna','Dhangadhi',2001)");
			cn.close();
```

```
            }
            catch (Exception e)
            {
                  System.out.println("Cannot insert into the table"+e);
            }

      }
}
```

SELECT OPERATION

Example: Program that is used to select multiple rows from the table of SQL database

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.Statement;
class QueryDemo {
      static final String JDBC_DRIVER = "com.mysql.cj.jdbc.Driver";
      static final String DB_URL = "jdbc:mysql://localhost:3306/bca_db";
      private static final String USER_NAME = "root";
      private static final String PASSWORD = "";
      public static void main(String arg[]){
            try{
                  Class.forName(JDBC_DRIVER);
                  Connection cn =
DriverManager.getConnection(DB_URL,USER_NAME,PASSWORD);
                  Statement st = cn.createStatement();

                  ResultSet rs;
                  rs = st.executeQuery("SELECT name FROM student WHERE id =
1001");
                  while ( rs.next() ) {
                        String lastName = rs.getString("name");
                        System.out.println(lastName);
                  }
                  cn.close();
            } catch (Exception e) {
                  System.out.println("Cannot select from the table"+e);

            }
      }
}
```

UPDATE OPERATION:

Example: Program that is used to modify the record in the table of SQL database

```
import java.sql.Connection;
```

```
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.Statement;
class UpdateQuery {
	static final String JDBC_DRIVER = "com.mysql.cj.jdbc.Driver";
	static final String DB_URL = "jdbc:mysql://localhost:3306/bca_db";
	private static final String USER_NAME = "root";
	private static final String PASSWORD = "";
	public static void main(String arg[]){
		try{
			Class.forName(JDBC_DRIVER);
			Connection cn =
DriverManager.getConnection(DB_URL,USER_NAME,PASSWORD);
			Statement st = cn.createStatement();

			ResultSet rs;
			st.executeUpdate("UPDATE student SET name = 'Krishna'
WHERE id = 1001");
			rs = st.executeQuery("SELECT name from student where id =
1001");
			while ( rs.next() )
			{
				String lastName = rs.getString("name");
System.out.println(lastName);
			}
			cn.close();
		} catch (Exception e) {
			System.out.println("Cannot update the table"+e);
		}
	}
}
```

DELETE OPERATION

Example: Program that is used to delete record from the table of SQL database

```
import java.sql.*;
class DeleteQuery {
	static final String JDBC_DRIVER = "com.mysql.cj.jdbc.Driver";
	static final String DB_URL = "jdbc:mysql://localhost:3306/bca_db";
	private static final String USER_NAME = "root";
	private static final String PASSWORD = "";
	public static void main(String arg[]){
		try{
			Class.forName(JDBC_DRIVER);
			Connection cn =
DriverManager.getConnection(DB_URL,USER_NAME,PASSWORD);
```

```
                Statement st = cn.createStatement();
                st.executeUpdate("DELETE from student WHERE id = 1001");
                cn.close();
            } catch (Exception e) {
                System.out.println("Cannot delete the data"+e);

            }
      }
}
```

CREATE OPERATION

Example: Program that is used to create table in the given database

```
import java.sql.*;
class CreateTableQuery {
      static final String JDBC_DRIVER = "com.mysql.cj.jdbc.Driver";
      static final String DB_URL = "jdbc:mysql://localhost:3306/bca_db";
      private static final String USER_NAME = "root";
      private static final String PASSWORD = "";
      public static void main(String arg[]){
            try{
                Class.forName(JDBC_DRIVER);
                Connection cn =
DriverManager.getConnection(DB_URL,USER_NAME,PASSWORD);
                Statement st = cn.createStatement();

                String sql = "create table college(cname varchar(255), address
varchar(255))";
                st.executeUpdate(sql);
                System.out.println("Table is created in the given database");
                st.close();
            } catch (Exception e) {
                System.out.println("Cannot create the table"+e);

            }
      }
}
```

5.6 SQL Injection and Prepared Statements

SQL Injection

- SQL injection is a code injection technique that might destroy your database.
- SQL injection is one of the most common web hacking techniques.
- SQL injection is the placement of malicious code in SQL statements, via web page input.

SQL in Web Pages

SQL injection usually occurs when you ask a user for input, like their username/userid, and instead of a name/id, the user gives you an SQL statement that you will unknowingly run on your database.

Look at the following example which creates a SELECT statement by adding a variable (txtUserId) to a select string. The variable is fetched from user input (getRequestString):

Example

txtUserId = getRequestString("UserId");
txtSQL = "SELECT * FROM Users WHERE UserId = " + txtUserId;

PreparedStatements

The PreparedStatement is derived from the more general class, Statement. The PreparedStatement interface extends the Statement interface which gives us added functionality over a generic Statement object. This statement gives us the flexibility of supplying arguments dynamically. It is more efficient to use the PreparedStatement because the SQL statement that is sent gets pre-compiled in the DBMS. We can also use PreparedStatement to safely provide values to the SQL parameters, through a range of setter methods (i.e. setInt (int, int), setString (int, String), etc.). We can execute the same query repeatedly with different parameter values. The PreparedStatement interface accepts input parameters at runtime i.e. the PreparedStatement object only uses the IN parameter. Just as we close a Statement object, we should also close the PreparedStatement object.

Example:

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;

public class PrepDemo {
	static final String JDBC_DRIVER = "com.mysql.cj.jdbc.Driver";
	static final String DB_URL = "jdbc:mysql://localhost:3306/bca_db";
	private static final String USER_NAME = "root";
	private static final String PASSWORD = "";
	public static void main(String arg[]){
		try{
			Class.forName(JDBC_DRIVER);
			Connection cn =
DriverManager.getConnection(DB_URL,USER_NAME,PASSWORD);
			PreparedStatement pstmt = cn.prepareStatement("INSERT INTO
authors VALUES(?, ?, ?)");
			pstmt.setInt(1,19);
			pstmt.setString(2, "Sunil");
			pstmt.setString(3,"Bist");

			pstmt.execute();
```

```
            }catch (Exception e ) { e.printStackTrace();
            } // end catch

      }
}
```

The three question marks (?) in the the preceding SQL statement‟s last line are placeholders for values that will be passed as part of the query to the database. Before executing a PreparedStatement, the program must specify the parameter values by using the Prepared- Statement interface‟s set methods. For the preceding query, parameters are int and strings that can be set with Prepared- Statement method setInt and setString. Method setInt's and setString‟s first argument represents the parameter number being set, and the second argument is that parameter‟s value. Parameter numbers are counted from 1, starting with the first question mark (?).

Interface PreparedStatement provides set methods for each supported SQL type. It‟s important to use the set method that is appropriate for the parameter‟s SQL type in the database—SQLExceptions occur when a program attempts to convert a parameter value to an incorrect type.

5.7 Row Sets and Transactions

RowSet

Transactions

If your JDBC Connection is in auto-commit mode, which it is by default, then every SQL statement is committed to the database upon its completion.

That may be fine for simple applications, but there are three reasons why you may want to turn off the auto-commit and manage your own transactions:

- To increase performance.
- To maintain the integrity of business processes.
- To use distributed transactions.

Transactions enable you to control if, and when, changes are applied to the database. It treats a single SQL statement or a group of SQL statements as one logical unit, and if any statement fails, the whole transaction fails.

To enable manual- transaction support instead of the auto-commit mode that the JDBC driver uses by default, use the Connection object's setAutoCommit() method. If you pass a boolean false to setAutoCommit( ), you turn off auto-commit. You can pass a boolean true to turn it back on again.

For example, if you have a Connection object named conn, code the following to turn off auto-commit

conn.setAutoCommit(false);

Commit & Rollback

Once you are done with your changes and you want to commit the changes then call commit() method on connection object as follows −

conn.commit( );

Otherwise, to roll back updates to the database made using the Connection named conn, use the following code −

conn.rollback( );

The following example illustrates the use of a commit and rollback object −

```
try{
           //Assume a valid connection object conn
           conn.setAutoCommit(false);
           Statement stmt = conn.createStatement();

           String SQL = "INSERT INTO Employees VALUES (106, 20, 'sunil',
'bist')";
           stmt.executeUpdate(SQL);
           //Submit a malformed SQL statement that breaks
           String SQL = "INSERTED IN Employees  VALUES (107, 22, 'ravi',
'khadka')";
           stmt.executeUpdate(SQL);
           // If there is no error.
           conn.commit();
      }catch(SQLException se){
           // If there is any error.
           conn.rollback();
      }
```

5.8 SQL Escapes

https://docs.oracle.com/cd/E13157_01/wlevs/docs30/jdbc_drivers/sqlescape.html

The escape syntax gives you the flexibility to use database specific features unavailable to you by using standard JDBC methods and properties.

The general SQL escape syntax format is as follow:

{keyword 'parameters'}

Following are various escape syntaxes in JDBC:

d, t, ts Keywords: They help identify date, time, and timestamp literals. As you know, no two DBMSs represent time and date the same way. This escape syntax tells the driver to render the date or time in the target database's format

{d 'yyyy-mm-dd'}

Where yyyy = year, mm = month; dd = date. Using this syntax {d '2009-09-03'} is March 9, 2009.

Example

```
//Create a Statement object
stmt = conn.createStatement();
//Insert data ==> ID, First Name, Last Name, DOB
String sql="INSERT INTO STUDENTS VALUES" + "(100,'Zara','Ali', {d '2001-12-16'})";
stmt.executeUpdate(sql);
```

escape Keyword

This keyword identifies the escape character used in LIKE clauses. Useful when using the SQL wildcard %, which matches zero or more characters. For example −

```
String sql = "SELECT symbol FROM MathSymbols WHERE symbol LIKE '\%' {escape '\'}";
stmt.execute(sql);
```

If you use the backslash character (\) as the escape character, you also have to use two backslash characters in your Java String literal, because the backslash is also a Java escape character.

fn Keyword

This keyword represents scalar functions used in a DBMS. For example, you can use SQL function *length* to get the length of a string −

{fn length('Hello World')}

This returns 11, the length of the character string 'Hello World'. call Keyword

This keyword is used to call the stored procedures. For example, for a stored procedure requiring an IN parameter, use the following syntax −

{call my_procedure(?)};

For a stored procedure requiring an IN parameter and returning an OUT parameter, use the following syntax −

{? = call my_procedure(?)};

oj Keyword

This keyword is used to signify outer joins. The syntax is as follows −

{oj outer-join}

Where outer-join = table {LEFT|RIGHT|FULL} OUTERJOIN {table | outer-join} on search-condition.

```
String sql = "SELECT Employees FROM {oj ThisTable RIGHT OUTER JOIN ThatTable on id
= '100'}";
stmt.execute(sql);
```