MODULE V
JAVA DATABASE CONNECTIVITY (JDBC)

STRUCTURE

LEARNING OBJECTIVES

- To understand database connectivity in Java
- To learn how to create and execute JDBC statements.

5.1 INTRODUCTION

JDBC stands for Java Database Connectivity. JDBC is a Java API to connect and execute the query with the database. It is a part of JavaSE (Java Standard Edition). JDBC API uses JDBC drivers to connect with the database. There are four types of JDBC drivers:

- JDBC-ODBC Bridge Driver,
- Native Driver,
- Network Protocol Driver, and
- Thin Driver

JDBC stands for Java Database Connectivity, which is a standard Java API for database-independent connectivity between the Java programming language and a wide range of databases.
The JDBC library includes APIs for each of the tasks mentioned below that are commonly associated with database usage.

- Making a connection to a database.
- Creating SQL or MySQL statements.
- Executing SQL or MySQL queries in the database.
- Viewing & Modifying the resulting records.

Fundamentally, JDBC is a specification that provides a complete set of interfaces that allows for portable access to an underlying database. Java can be used to write different types of executables, such as −

- Java Applications
- Java Applets
- Java Servlets
- Java ServerPages (JSPs)
- Enterprise JavaBeans (EJBs).

All of these different executables are able to use a JDBC driver to access a database, and take advantage of the stored data.JDBC provides the same capabilities as ODBC, allowing Java programs to contain database-independent code.

Why Should We Use JDBC

Before JDBC, ODBC API was the database API to connect and execute the query with the database. But, ODBC API uses ODBC driver which is written in C language (i.e. platform dependent and unsecured). That is why Java has defined its own API (JDBC API) that uses JDBC drivers (written in Java language).We can use JDBC API to handle database using Java program and can perform the following activities:

1. Connect to the database
2. Execute queries and update statements to the database
3. Retrieve the result received from the database.

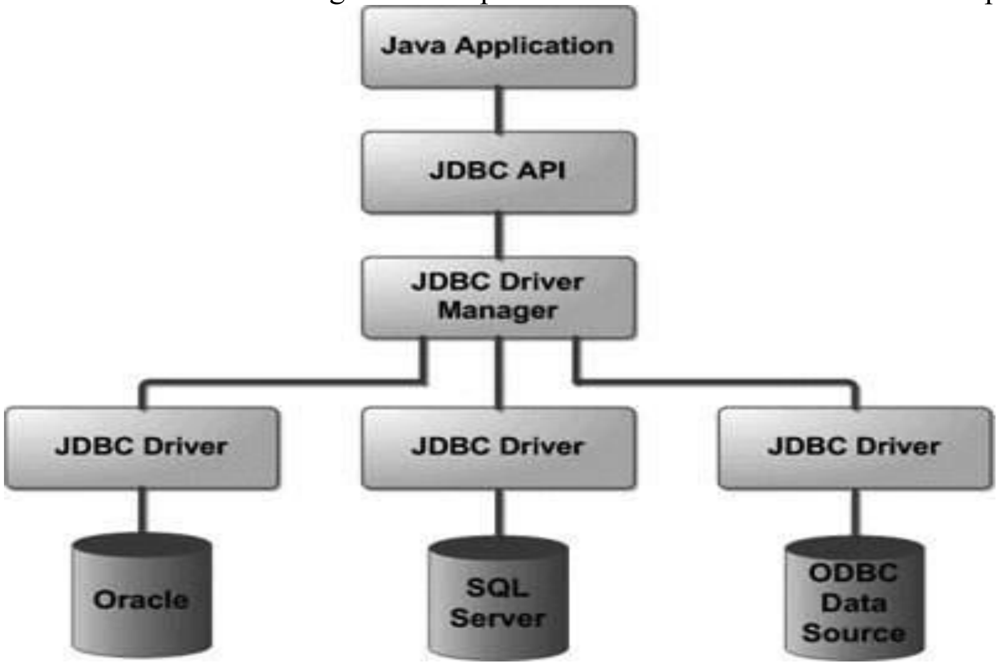
5.2 DATABASE CONNECTIVITY: JDBC ARCHITECTURE, JDBC DRIVERS

JDBC Architecture

The JDBC API supports both two-tier and three-tier processing models for database access but in general, JDBC Architecture consists of two layers −

- JDBC API: This provides the application-to-JDBC Manager connection.
- JDBC Driver API: This supports the JDBC Manager-to-Driver Connection.

The JDBC API uses a driver manager and database-specific drivers to provide transparent connectivity to heterogeneous databases. The JDBC driver manager ensures that the correct driver is used to access each data source. The driver manager is capable of supporting multiple concurrent drivers connected to multiple heterogeneous databases. Following is the architectural diagram, which shows the location of the driver manager with respect to the JDBC drivers and the Java application −



Common JDBC Components

The JDBC API provides the following interfaces and classes −

- **Driver Manager**: This class manages a list of database drivers. Matches connection requests from the java application with the proper database driver using communication sub protocol.

The first driver that recognizes a certain subprotocol under JDBC will be used to establish a database Connection.

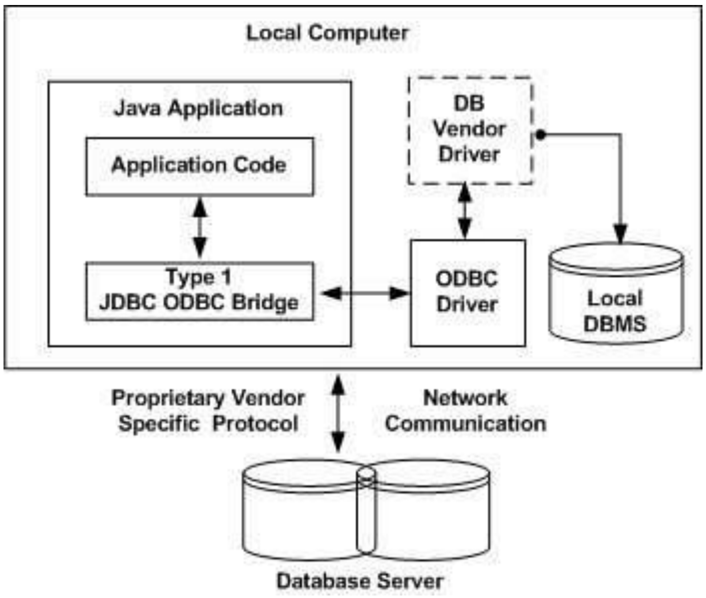
- **Driver:** This interface handles the communications with the database server. You will interact directly with Driver objects very rarely. Instead, you use DriverManager objects, which manages objects of this type. It also abstracts the details associated with working with Driver objects.
- **Connection:** This interface with all methods for contacting a database. The connection object represents communication context, i.e., all communication with database is through connection object only.
- **Statement:** You use objects created from this interface to submit the SQL statements to the database. Some derived interfaces accept parameters in addition to executing stored procedures.
- **ResultSet:** These objects hold data retrieved from a database after you execute an SQL query using Statement objects. It acts as an iterator to allow you to move through its data.
- **SQLException:** This class handles any errors that occur in a database application.

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, Types 1, 2, 3, and 4, which is explained below −

TYPE 1: JDBC-ODBC BRIDGE DRIVER

In a Type 1 driver, a JDBC bridge is used to access ODBC 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.

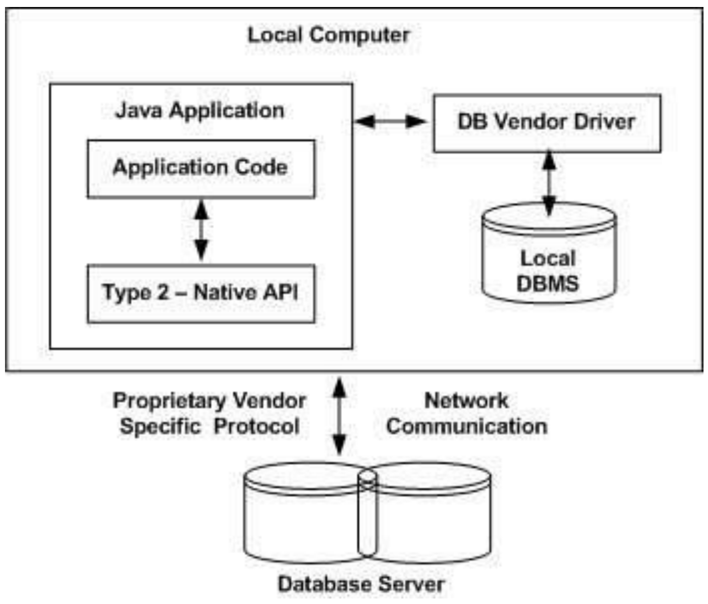


The JDBC-ODBC Bridge that comes with JDK 1.2 is a good example of this kind of driver.

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 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.

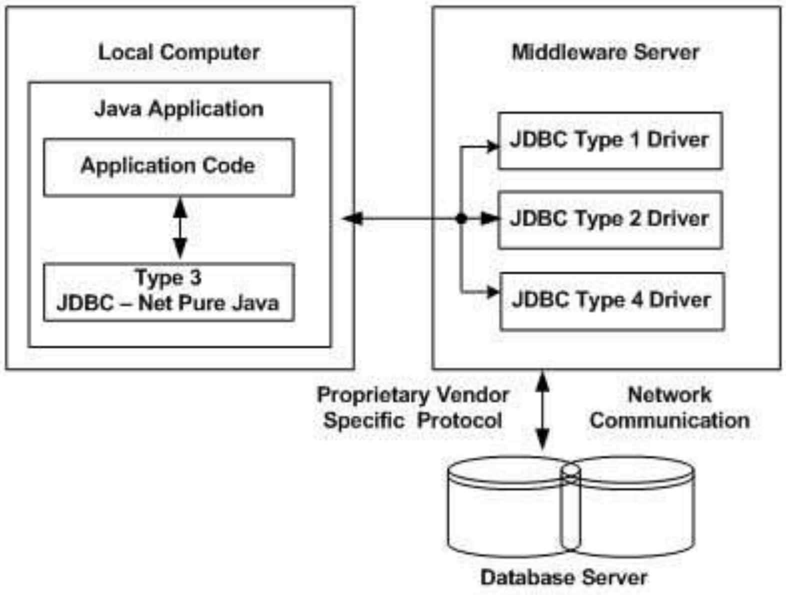
If we change the Database, we have to change the native API, as it is specific to a database and they are mostly obsolete now, but you may realize some speed increase with a Type 2 driver, because it eliminates ODBC's overhead.



The Oracle Call Interface (OCI) driver is an example of a Type 2 driver.

Type 3: JDBC-Net pure Java

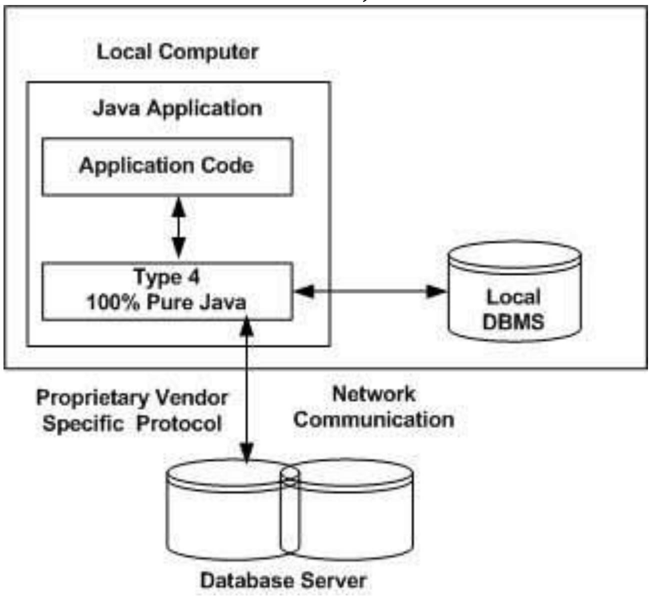
In a Type 3 driver, a three-tier approach is used to access databases. The JDBC clients use standard network sockets to communicate with a middleware application server. The socket information is then translated by the middleware application server into the call format required by the DBMS, and forwarded 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.



You can think of the application server as a JDBC "proxy," meaning that it makes calls for the client application. As a result, you need some knowledge of the application server's configuration in order to effectively use this driver type.

Your application server might use a Type 1, 2, or 4 driver to communicate with the database, understanding the nuances will prove helpful.

Type 4: 100% Pure Java
In a Type 4 driver, a pure Java-based driver communicates directly with the 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.

5.3 THE JDBC API: LOADING A DRIVER, CONNECTING TO A DATABASE, CREATING AND EXECUTING JDBC STATEMENTS

The JDBC API- is a Java API for accessing virtually any kind of tabular data. (As a point of interest, JDBC is the trademarked name and is not an acronym; nevertheless, JDBC is often thought of as standing for "Java Database Connectivity.") The JDBC API consists of a set of classes and interfaces written in the Java programming language that provide a standard API for tool/database developers and makes it possible to write industrial strength database applications using an all-Java API.The JDBC API makes it easy to send SQL statements to relational database systems and supports all dialects of SQL. But the JDBC 2.0 API goes beyond SQL, also making it possible to interact with other kinds of data sources, such as files containing tabular data.

The value of the JDBC API is that an application can access virtually any data source and run on any platform with a Java Virtual Machine. In other words, with the JDBC API, it isn't necessary to write one program to access a Sybase database, another program to access an Oracle database, another program to access an IBM DB2 database, and so on. One can write a single program using the JDBC API, and the program will be able to send SQL or other statements to the appropriate data source. And, with an application written in the Java programming language, one doesn't have to worry about writing different applications to run on different platforms. The combination of the Java platform and the JDBC API lets a programmer write once and run anywhere.

The Java programming language, being robust, secure, easy to use, easy to understand, and automatically downloadable on a network, is an excellent language basis for database applications. What is needed is a way for Java applications to talk to a variety of different data sources. JDBC is the mechanism for doing this.

The JDBC 2.0 API extends what can be done with the Java platform. For example, the JDBC API makes it possible to publish a web page containing an applet that uses information obtained from a remote data source. Or an enterprise can use the JDBC API to connect all its employees (even if they are using a conglomeration of Windows, Macintosh, and UNIX machines) to one or more internal databases via an intranet. With more and more programmers using the Java programming language, the need for easy and universal data access from Java is continuing to grow.MIS managers like the combination of the Java platform and JDBC technology because it makes disseminating information easy and economical. Businesses can continue to use their installed databases and access information easily even if it is stored on different database management systems or other data sources.

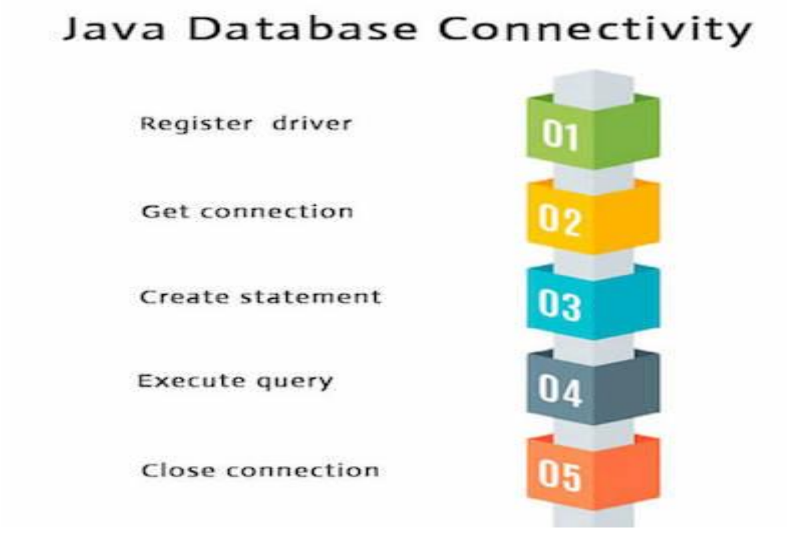
Development time for new applications is short. Installation and version control are greatly simplified. A programmer can write an application or an update once, put it on the server, and everybody has access to the latest version. And for businesses selling information services, the combination of the Java and JDBC technologies offers a better way of getting out information updates to external customers.

Loading A Driver- To begin with, you first need load the driver or register it before using it in the program . Registration is to be done once in your program. You can register a driver in one of two ways mentioned below :

- Class.forName() : Here we load the driver's class file into memory at the runtime. No need of using new or creation of object .The following example uses Class.forName() to load the Oracle driver –
  Class.forName("oracle.jdbc.driver.OracleDriver");
- DriverManager.registerDriver(): DriverManager is a Java inbuilt class with a static member register. Here we call the constructor of the driver class at compile time . The following example uses DriverManager.registerDriver()to register the Oracle driver –
  DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver())

There are 5 steps to connect any java application with the database using JDBC. These steps are as follows:

- Register the Driver class
- Create connection
- Create statement
- Execute queries
- Close connection



connecting to a database-After loading the driver, establish connections using :
Connection con = DriverManager.getConnection(url,user,password)

user – username from which your sql command prompt can be accessed.
password – password from which your sql command prompt can be accessed.
con: is a reference to Connection interface.
url : Uniform Resource Locator. It can be created as follows:
String url = “ jdbc:oracle:thin:@localhost:1521:xe”
Where oracle is the database used, thin is the driver used , @localhost is the IP Address where database is stored, 1521 is the port number and xe is the service provider. All 3 parameters above are of String type and are to be declared by programmer before calling the function. Use of this can be referred from final code.

Creating and executing JDBC statements-

Create a statement-Once a connection is established you can interact with the database.
The JDBCStatement, CallableStatement, and Prepared Statement interfaces define the methods that enable you to send SQL commands and receive data from your database.
Use of JDBC Statement is as follows:
Statement st = con.createStatement();
Here, con is a reference to Connection interface used in previous step .

Execute the query-Now comes the most important part i.e executing the query. Query here is an SQL
Query . Now we know we can have multiple types of queries. Some of them are as follows:
Query for updating / inserting table in a database.

Query for retrieving data .
The executeQuery() method of Statement interface is used to execute queries of retrieving values from the database. This method returns the object of ResultSet that can be used to get all the records of a table. The executeUpdate(sql query) method ofStatement interface is used to execute queries of updating/inserting .

5.4 HANDLING SQL EXCEPTIONS, ACCESSING RESULT SETS: TYPES OF RESULT SETS, METHODS OF RESULT SET INTERFACE. AN EXAMPLE JDBC APPLICATION TO QUERY A DATABASE

Handling SQL exceptions- When JDBC encounters an error during an interaction with a data source,it throws an instance of SQLException as opposed to Exception. (A data source in this context represents the database to which a Connection object is connected.) The SQLException instance contains the following information that can help you determine the cause of the error:A description of the error. Retrieve the String object that contains this description by calling the method SQLException.getMessage.

A SQL State code. These codes and their respective meanings have been standardized by ISO/ANSI and Open Group (X/Open), although some codes have been reserved for database vendors to define for themselves. This String object consists of five alphanumeric characters. Retrieve this code by calling the method SQL Exception.getSQLState.
An error code. This is an integer value identifying the error that caused the SQL Exception instance to be thrown. Its value and meaning are implementation-specific and might be the actual error code returned by the underlying data source. Retrieve the error by calling the method SQLException.getErrorCode

A SQL Exception instance might have a causal relationship, which consists of one or more Throwable objects that caused the SQLException instance to be thrown. To navigate this chain of causes, recursively call the method SQL Exception.getCause until a null value is returned.
A reference to any chained exceptions. If more than one error occurs, the exceptions are referenced through this chain. Retrieve these exceptions by calling the method SQL Exception.getNextException on the exception that was thrown.

Accessing result sets
The SQL statements that read data from a database query, return the data in a result set. The SELECT statement is the standard way to select rows from a database and view them in a result set. The java.sql.ResultSet interface represents the result set of a database query.
A ResultSet object maintains a cursor that points to the current row in the result set. The term "result set" refers to the row and column data contained in a ResultSet object.

The methods of the ResultSet interface can be broken down into three categories −
Navigational methods: Used to move the cursor around.
Get methods: Used to view the data in the columns of the current row being pointed by the cursor.
Update methods: Used to update the data in the columns of the current row. The updates can then be updated in the underlying database as well.
The cursor is movable based on the properties of the ResultSet. These properties are designated when the corresponding Statement that generates the ResultSet is created.
JDBC provides the following connection methods to create statements with desired ResultSet

–

createStatement(int RSType, int RSConcurrency);
prepareStatement(String SQL, int RSType, int RSConcurrency);
prepareCall(String sql, int RSType, int RSConcurrency);
The first argument indicates the type of a ResultSet object and the second argument is one of two ResultSet constants for specifying whether a result set is read-only or updatable.

Type of ResultSet
The possible RSType are given below. If you do not specify any ResultSet type, you will automatically get one that is TYPE_FORWARD_ONLY.

Type	Description
ResultSet.TYPE_FORWARD_ONLY	The cursor can only move forward in the result set.
ResultSet.TYPE_SCROLL_INSENSITIVE	The cursor can scroll forward and backward, and the result set is not sensitive to changes made by others to the database that occur after the result set was created.
ResultSet.TYPE_SCROLL_SENSITIVE.	The cursor can scroll forward and backward, and the result set is sensitive to changes made by others to the database that occur after the result set was created.

Concurrency of ResultSet
The possible RSConcurrency are given below. If you do not specify any Concurrency type, you will automatically get one that is CONCUR_READ_ONLY.

Concurrency	Description
ResultSet.CONCUR_READ_ONLY	Creates a read-only result set. This is the default
ResultSet.CONCUR_UPDATABLE	Creates an updateable result set.

All our examples written so far can be written as follows, which initializes a Statement object to create a forward-only, read only ResultSet object −

```
try {
   Statement stmt = conn.createStatement(
                           ResultSet.TYPE_FORWARD_ONLY,
                           ResultSet.CONCUR_READ_ONLY);
}
catch(Exception ex) {
   ....
}
finally {
   ....
}
```

Methods of result set interface

- The result of the query after execution of database statement is returned as table of data according to rows and columns. This data is accessed using the ResultSet interface.
- A default ResultSet object is not updatable and the cursor moves only in forward direction.

Creating ResultSet Interface
To execute a Statement or PreparedStatement, we create ResultSet object.

Example
Statement stmt = connection.createStatement();
ResultSet result = stmt.executeQuery("select * from Students");
Or
String sql = "select * from Students";
PreparedStatementstmt = con.prepareStatement(sql);
ResultSet result = stmt.executeQuery();

ResultSet Interface Methods

Methods	Description
public boolean absolute(int row)	Moves the cursor to the specified row in the ResultSet object.
public void beforeFirst( )	It moves the cursor just before the first row i.e. front of the ResultSet.
public void afterLast()	Moves the cursor to the end of the ResultSet object, just after the last row.
public boolean first()	Moves the cursor to first value of ResultSet object.
public boolean last( )	Moves the cursor to the last row of the ResultSet object.
public boolean previous ( )	Just moves the cursor to the previous row in the ResultSet object.
public boolean next( )	It moves the curser forward one row from its current position.
public intgetInt(intcolIndex)	It retrieves the value of the column in current row as int in given ResultSet object.
public String getString(intcolIndex)	It retrieves the value of the column in current row as int in given ResultSet object.
public void relative(int rows)	It moves the cursor to a relative number of rows.

Types of ResultSet Interface:

1. ResultSet.TYPE_FORWARD_ONLY
The ResultSet can only be navigating forward.
2. ResultSet.TYPE_SCROLL_INSENSITIVE
The ResultSet can be navigated both in forward and backward direction. It can also jump from current position to another position. The ResultSet is not sensitive to change made by others.
3. ResultSet.TYPE_SCROLL_SENSITIVE
The ResultSet can be navigated in both forward and backward direction. It can also jump from

current position to another position. The ResultSet is sensitive to change made by others to the database.

Example : Program to illustrate ResultSet interface with scrollable

```
import java.sql.*;
classResultSetTest
{
    public static void main(String args[])
    {
        Connection con = null;
        Statement stmt = null;
        ResultSetrs = null;
        try
        {
            Class.forName("oracle.jdbc.driver.OracleDriver");
            con = DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:XE", "scott",
"tiger");
            stmt = con.createStatement (ResultSet.TYPE_SCROLL_SENSITIVE,
ResultSet.CONCUR_UPDATABLE);
            rs = stmt.executeQuery("Select * from Student");
            rs.absolute(5);     //Accessing 5th row
           System.out.println(rs.getInt(1)+" "+rs.getString(2)+" "+rs.getString(3));
           rs.close();
           stmt.close();
           con.close();
        }
        catch(Exception e)
        {
           e.printStackTrace();
        }
    }
}
```

An example JDBC application to query a database

```
package org.mysqltutorial;
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

/**
 *
 * @author mysqltutorial.org
 */
public class Main {

   public static void main(String[] args) {
       //
       String sql = "SELECT first_name, last_name, email " +
                "FROM candidates";
```

```
        try (Connection conn = MySQLJDBCUtil.getConnection();
             Statement stmt  = conn.createStatement();
             ResultSet rs    = stmt.executeQuery(sql)) {

            // loop through the result set
            while (rs.next()) {
                System.out.println(rs.getString("first_name") + "\t" +
                                   rs.getString("last_name")  + "\t" +
                                   rs.getString("email"));

            }
        } catch (SQLException ex) {
            System.out.println(ex.getMessage());
        }
    }
}
```

The output of the program is as follows:
mysql jdbc query data

5.5 SUMMARY

After you've installed the appropriate driver, it is time to establish a database connection using JDBC. The programming involved to establish a JDBC connection is fairly simple. Here are these simple four steps −

- **Import JDBC Packages:** Add **import** statements to your Java program to import required classes in your Java code.
- **Register JDBC Driver:** This step causes the JVM to load the desired driver implementation into memory so it can fulfill your JDBC requests.
- **Database URL Formulation:** This is to create a properly formatted address that points to the database to which you wish to connect.
- **Create Connection Object:** Finally, code a call to the DriverManager object's getConnection( ) method to establish actual database connection.

5.5 SELF ASSESMENT QUESTIONS

1. Explain JDBC architecture
2. Explain Creating and executing JDBC statements
3. Explain types of result sets
4. Explain Methods of result set interface

5.6 SUGGESTED READINGS

1. The complete reference Java –2: V Edition By Herbert Schildt Pub. TMH.
2. SAMS teach yourself Java – 2: 3rd Edition by Rogers Cedenhead and Leura Lemay Pub. Pearson Education
3. Java: A Beginner's Guide by Herbert Schildt
4. Java: Programming Basics for Absolute Beginners by Nathan Clark.