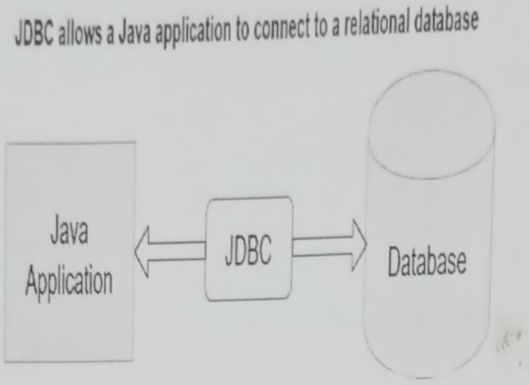
Unit – I : JDBC (Java Database Connectivity)

Contents

- JDBC Concept
- Related Classes
- JDBC Architecture
- JDBC API
- Types of JDBC Drivers
- Java SQL Packages
- Steps to create JDBC Application
- Simple JDBC Program
- *Question Bank*

- **Concept of JDBC (Java Database Connectivity):**

JDBC stands for **Java Database Connectivity**, which is a standard Java API(**Application Programming Interface**) for database independent connectivity between the Java Programming Language and a wide range of databases.



The JDBC library includes APIs for each tasks mentioned below which 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 a database.

Java can be used to write different types of executables, such as:

- ❖ Java Applications
- ❖ Java Applets
- ❖ Java Servlets
- ❖ Java Server Pages (JSPs)
- ❖ Enterprise Java Beans(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 such as ODBC allowing Java programs to contain database independent code.

What is API?

API is an acronym for **Application Programming Interface** which is a computing interface which defines interactions between multiple software intermediaries.

❖ **What are the main classes and interfaces of JDBC?**

JDBC API is mainly divided into two package.

❖ java.sql
❖ javax.sql

When we are using JDBC, we have to import these packages to use classes and interfaces in our application. Following are the important classes and interfaces of JDBC.

Class/interface	Description
1)DriverManager	This class manages the JDBC drivers. You need to register your drivers to this. It provides methods such as registerDriver() and getConnection().
2)Driver	This interface is the Base interface for every driver class i.e. if you want to create a JDBC Driver of your own you need to implement this interface. If you load a Driver class (implementation of this interface), it will create an instance of itself and register with the driver manager.
3)Statement	This interface represents a static SQL statement. Using the Statement object and its methods, you can execute an SQL statement and get the results of it. It provides methods such as execute(), executeBatch(), executeUpdate() etc. to execute the statements.
4)PreparedStatement	This represents a precompiled SQL statement. An SQL statement is compiled and stored in a prepared statement and you can later execute this multiple times. You can get an object of this interface using the method of the Connection interface named prepareStatement(). This provides methods such as executeQuery(), executeUpdate(), and execute() to execute the prepared statements
5)CallableStatement	Using an object of this interface you can execute the stored procedures. This returns single or multiple results. It will accept input parameters too. You can create a CallableStatement using the prepareCall() method of the Connection interface.
6)Connection	This interface represents the connection with a specific database. SQL statements are executed in the context of a connection.This interface provides methods such as close(), commit(), rollback(), createStatement(), prepareCall(), prepareStatement(), setAutoCommit() setSavepoint() etc.
7)ResultSet	This interface represents the database result set, a table which is generated by executing statements. This interface

Class/interface	Description
	Provides getter and update methods to retrieve and update its contents respectively.
8)ResultSetMetaData	This interface is used to get the information about the result set such as, number of columns, name of the column, data type of the column, schema of the result set, table name, etc. It provides methods such as getColumnCount(), getColumnName(), getColumnType(), getTableName(), getSchemaName() etc.

❖ JDBC Architecture:

The JDBC API supports both two-tier and three-tier processing models for database access.

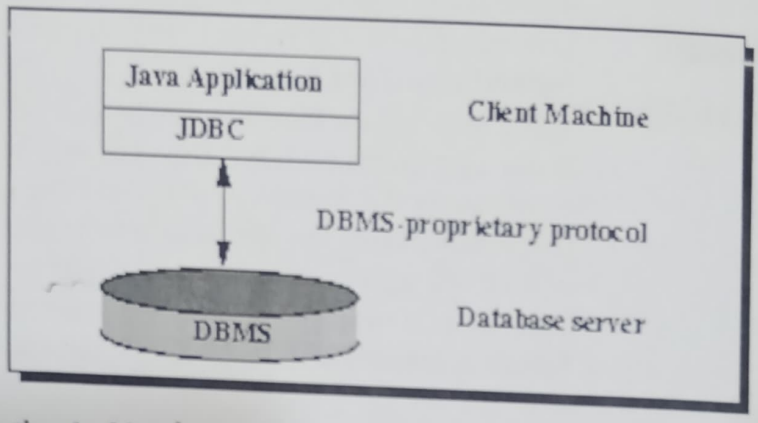
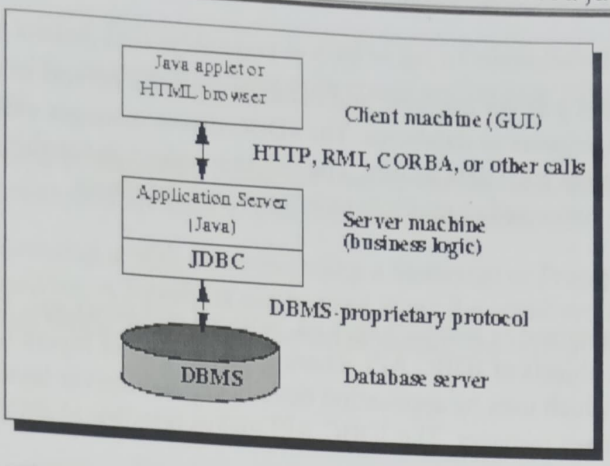


Diagram 1: Two-tier Architecture for Data Access.

In the two-tier model, a Java application talks directly to the data source. This requires a JDBC driver that can communicate with the particular data source being accessed. A user's commands are delivered to the database or other data source, and the results of those statementsare sent back to the user. The data source may be located on another machine to which the user is connected via a network. This is referred to as a client/server configuration, with the user's machine as the client, and the machine housing the data source as the server. The network can be an intranet, which, for example, connects employees within a corporation, or it can be the internet.

In the three-tier model, commands are sent to a "middle tier" of services, which then sends the commands to the data source. The data source processes the commands and sends the results back to the middle tier, which then sends them to the user. MIS directors find the three-tier model very attractive because the middle tier makes it possible to maintain control over access and the kinds of updates that can be made to corporate data. Another advantage is that it simplifies the deployment of applications. Finally, in many cases, the three-tier architecture can provide performance advantages.

Diagram 2: Three-tier Architecture for Data Access.



Until recently, the middle tier has often been written in languages such as C or C++, which offer fast performance. However, with the introduction of optimizing compilers that translate Java byte code into efficient machine-specific code and technologies such as Enterprise JavaBeans™, the Java platform is fast becoming the standard platform for middle-tier development. This is a big plus, making it possible to take advantage of Java's robustness, multithreading, and security features.

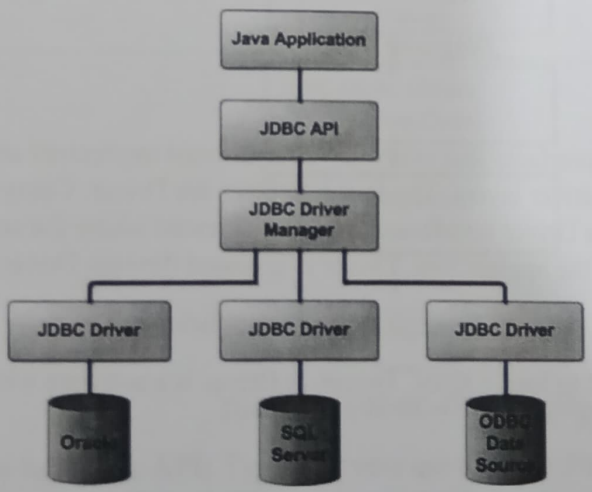
With enterprises increasingly using the Java programming language for writing server code, the JDBC API is being used more and more in the middle tier of a three-tier architecture. Some of the features that make JDBC a server technology are its support for connection pooling, distributed transactions, and disconnected row sets. The JDBC API is also what allows access to a data source from a Java middle tier.

Diagram 3: General 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 i.e.

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

Following architecture shows the location of the driver manager with respect to the JDBC drivers and the Java application.



Thus 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.

❖ JDBC API:

JDBC API is contained in two packages i.e. *java.sql* and *javax.sql*. The java.sql package contains core Java objects of JDBC API. There are two distinct layers within JDBC API; Application layer which uses by application developers and Driver layer which can implemented by driver manager. The JDBC API makes possible to following three things i.e.

- Establish a connection with a database or access any data source.
- Send SQL statement
- Process the results.

The connection between application and driver layer is shown in following figure;

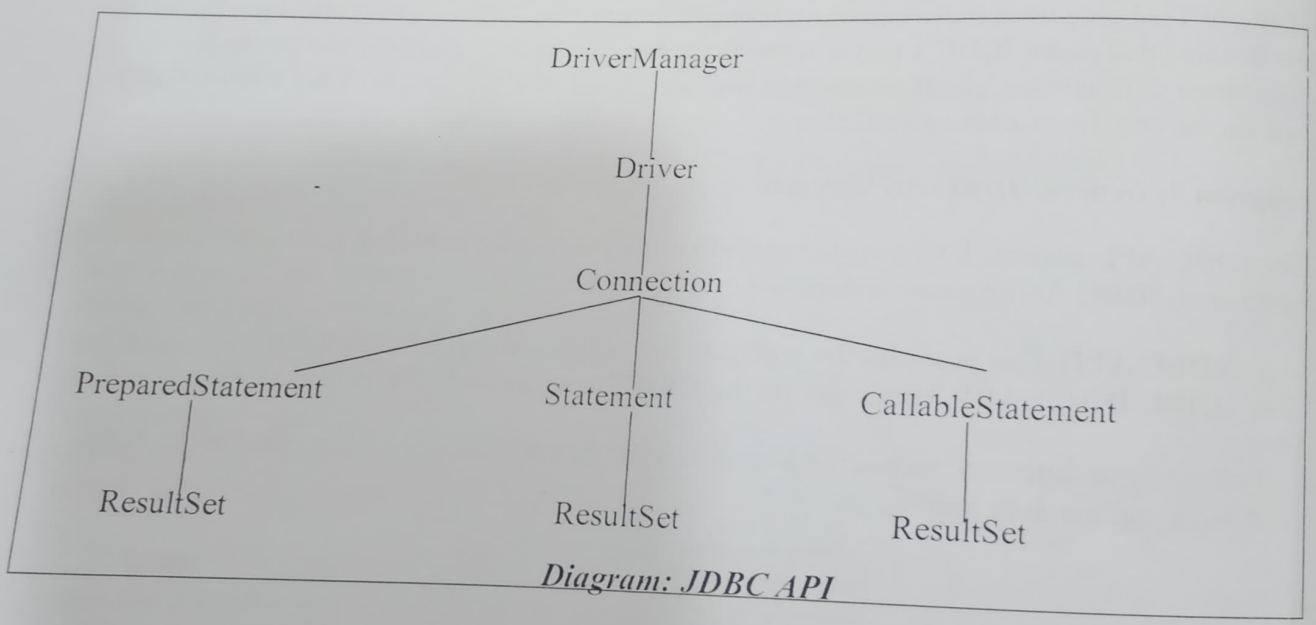


Diagram: JDBC API

There are seven main interfaces that every driver layer must implement and one class that bridges the Application and driver layers. The four interfaces are Driver, Connection, Statement and ResultSet. The Driver interface can be implemented where the connection to the database is made. In most of the applications, Driver is accessed through Driver Manager class.

In the following ways, JDBC API can be implemented in application:

- DriverManager is used to load a JDBC Driver. A Driver is a software which is used for implementation of JDBC API.

- After a driver is loaded, DriverManager is used to get a Connection. In turn, a Connection is used to create a Statement or to create and prepare a PreparedStatement or CallableStatement.
- Statement and PreparedStatement object are used to execute SQL statements and CallableStatement objects are used to execute store procedures.
- The results of executing a SQL statement using a Statement or PreparedStatement are returned as a ResultSet. A ResultSet can be used to get the actual returned data or a ResultSetMetaData object that can be queried to identify the type of data returned in ResultSet.

❖ Types of JDBC Driver:

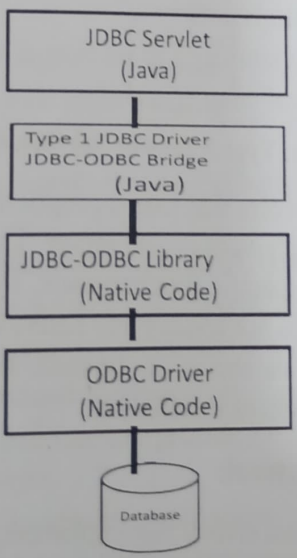
A JDBC driver translates standard JDBC calls that facilitates communication with the database using API. This translation provides JDBC applications with database independence. If the back-end database changes, only the JDBC driver must be replaced with few code modification required. There are four distinct types of JDBC Drivers:
Type 1: JDBC-ODBC Bridge.

Type 2: Java to Native API.

Type 3: Java to Network Protocol.

Type 4: Java to Database Protocol.

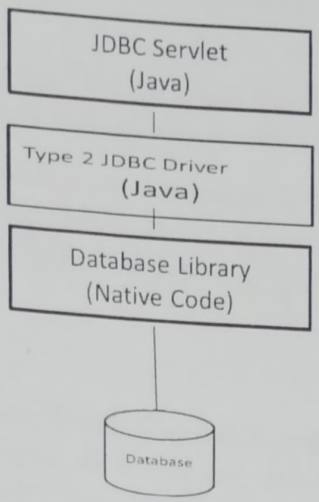
Type 1 JDBC-ODBC Bridge: Type 1 drivers acts as a "bridge" between JDBC and another database connectivity mechanism such as ODBC. The JDBC-ODBC bridge provides JDBC access using most standard ODBC drivers. This driver is included in *sun.jdbc.odbc* package. Thus, JDBC-ODBC bridge requires the native ODBC libraries, drivers and required support files be installed and configured on each client. This requirement may present a serious limitations for many applications.



Dia: Type 1 JDBC-ODBC Bridge

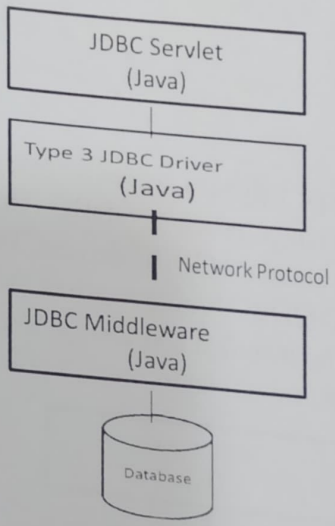
Type 2 Java to Native API: Type 2 drivers use the Java Native Interface(JNI) to make calls to database API. Type 2 drivers are usually faster than Type 1 drivers. Like Type 1

drivers, Type 2 drivers require native database client libraries to be installed and configured on the client machine.



Dia: Type 2 Java to Native API.

Type 3 Java to Network Protocol:

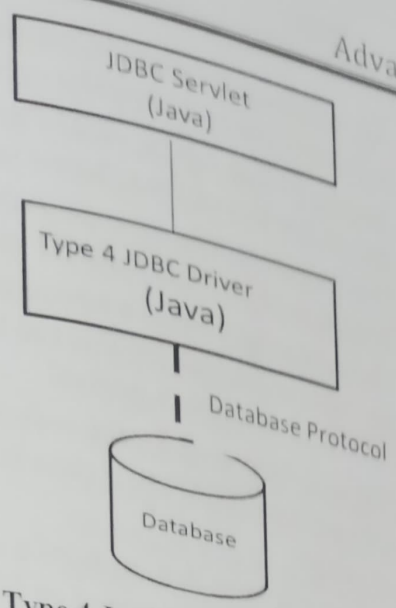


Dia: Type 3 Java to Network Protocol.

Type 3 driver are pure Java drivers that uses a proprietary network protocol to communicate with JDBC middleware on the server. The middleware then translates the network protocol to database specific function calls. Type 3 drivers are most flexible JDBC solution because they do not require any native database libraries on the client and can connect to many different databases on the back- end. Type 3 drivers can be deployed over the Internet without any client installation.

Type 4 Java to Database Protocol:

Type 4 drivers are pure Java drivers that implement a proprietary database protocol to communicate directly with the database. Like Type 3 drivers, they do not require any native database libraries and can be deployed over the Internet without any client installation required. One drawback of Type 4 driver is that they data specific.



Dia: Type 4 Java to Database Protocol.

❖ Java SQL Packages:

JDBC API is mainly divided into two packages. Each when we are using JDBC, we have to import these packages to use classes and interfaces in our application. Those packages are,

1. java.sql
2. javax.sql

1. java.sql package:

This package includes classes and interface to perform almost all JDBC operation such as creating and executing SQL Queries.

Following are important classes and interfaces of `java.sql` package.

classes/interface	Description
java.sql.BLOB	Provide support for BLOB(Binary Large Object) SQL type.
java.sql.Connection	creates a connection with specific database
java.sql.CallableStatement	Execute stored procedures
java.sql.CLOB	Provide support for CLOB(Character Large Object) SQL type.
java.sql.Date	Provide support for Date SQL type.
java.sql.Driver	create an instance of a driver with the DriverManager.
java.sql.DriverManager	This class manages database drivers.
java.sql.PreparedStatement	Used to create and execute parameterized query.
java.sql.ResultSet	It is an interface that provide methods to access the result row-by-row.
java.sql.Savepoint	Specify savepoint in transaction.
java.sql.SQLException	Encapsulate all JDBC related exception.
java.sql.Statement	This interface is used to execute SQL statements.
DatabaseMetaData	Comprehensive information about the database as a whole.

DriverAction	An interface that must be implemented when a Driver wants to be notified by DriverManager.
ResultSetMetaData	An object that can be used to get information about the types and properties of the columns in a ResultSet object.
RowId	The representation (mapping) in the Java programming language of an SQL ROWID value.
Savepoint	The representation of a savepoint, which is a point within the current transaction that can be referenced from the Connection.rollback method.
SQLData	The interface used for the custom mapping of an SQL user-defined type (UDT) to a class in the Java programming language.
SQLInput	An input stream that contains a stream of values representing an instance of an SQL structured type or an SQL distinct type.
SQLOutput	The output stream for writing the attributes of a user-defined type back to the database.
SQLType	An object that is used to identify a generic SQL type, called a JDBC type or a vendor specific data type.
SQLXML	cThe mapping in the JavaTM programming language for the SQL XML type.
Statement	The object used for executing a static SQL statement and returning the results it produces.
Struct	The standard mapping in the Java programming language for an SQL structured type.
Wrapper	Interface for JDBC classes which provide the ability to retrieve the delegate instance when the instance in questionis in fact a proxy class.

2. The javax.sql package:

This package is also known as JDBC extension API. It provides classes and interface to access server-side data.

Important classes and interface of `javax.sql` package

Classes/interface	Description
javax.sql.ConnectionEvent	Provide information about occurence of event.
javax.sql.ConnectionEventListener	Used to register event generated by PooledConnection object.
javax.sql.DataSource	Represent the DataSource interface used in an application.
javax.sql.PooledConnection	provide object to manage connection pools.
CommonDataSource	Interface that defines the methods which are common between DataSource, XADataSource and ConnectionPoolDataSource.

RowSet	The interface that adds support to the JDBC API for the JavaBeans component model.
RowSetInternal	The interface that a RowSet object implements in order to present itself to a RowSetReader or RowSetWriter object.
RowSetListener	An interface that must be implemented by a component that wants to be notified when a significant event happens in the life of a RowSet object.
RowSetMetaData	An object that contains information about the columns in a RowSet object.
RowSetReader	The facility that a disconnected RowSet object calls on to populate itself with rows of data.
RowSetWriter	An object that implements the RowSetWriter interface, called a writer.
StatementEventListener	An object that registers to be notified of events that occur on PreparedStatements that are in the Statement pool.

❖ **Steps to create JDBC Application:**

There are following six steps involved in building a JDBC application. These six steps are also called basic steps involve in connecting a Java application with Database using JDBC.

1. **Import the Packages:** Requires that we include the packages containing the JDBC classes needed for database programming. Most often by using *import java.sql.** ;

2.Register the JDBC driver(Register the Driver): Requires that we initialize a driver so you can open a communication channel with the database.

3.Open a connection(Create a Connection): Requires using the *DriverManager.getConnection()* method to create a Connection object, which represents a physical connection with the database.

4.Execute a query(Executing SQL Statement): Requires using an object of type Statement for building and submitting an SQL statement to the database.

5.Extract data from result set: Requires that we use the appropriate method to retrieve the data from the result set.

6.Clean up the environment(Closing the Connection): Requires explicitly closing all database resources. i.e. after executing SQL statement we need to close the connection and end the session.

```
/* Simple JDBC Program to create a student table that contain fields(sid int, sname varchar(20),marks int) */
import java.sql.*;        //step 1- Import the package
class DemoJDBC
{
public static void main(String args[ ])
{
try
{
Class.forName("com.mysql.jdbc.Driver"); // step 2- Register the Driver
System.out.println("Driver is Loaded");
Connection con=DriverManager.getConnection("jdbc:mysql://localhost:3306/JDBCDemo","root","root"); // step 3- Create
a Connection
System.out.println("Connection is Established");
Statement stmt=con.createStatement();
System.out.println("Statement Object is Created");
int i= stmt.executeUpdate("create table student(sid int, sname varchar(20),marks int)"); // step 4- Execute a Query
System.out.println("Result is" +i); // step 5- Generate result
System.out.println("Table is Created");
stmt.close(); // step 6- Closing the Connection
con.close();
}
catch(Exception e){ System.out.println(e);}
}
}
```