

# Chapter 4...

---

## Interacting with Database

---

**Weightage of Marks = 20, Teaching Hours = 06**

### **Contents**

- 4.1 Introduction
- 4.2 Software Architectures
  - 4.2.1 Two-Tier Model
  - 4.2.2 Three-Tier Model
  - 4.2.3 JDBC
    - 4.2.3.1 JDBC Architecture
    - 4.2.3.2 JDBC Components
    - 4.2.3.3 JDBC Drivers
  - 4.2.4 ODBC
  - 4.2.5 Other Applications
    - 4.2.5.1 JDBC API
    - 4.2.5.2 ODBC API
- 4.3 Connecting to Database
  - 4.3.1 Java Interfaces
  - 4.3.2 Driver Interface
  - 4.3.3 Driver Manager Class
  - 4.3.4 Connection Interface
    - 4.3.4.1 Connecting / Establishing Connection Object
    - 4.3.4.2 Retrieving Information
  - 4.3.5 Resultset Interface
- Important Points
- Practice Questions

### **Objectives**

- To Create Database Driven Business Applications using the Database API's Two Tier and Three Tier Models and the Java.Sql Package

## 4.1 INTRODUCTION

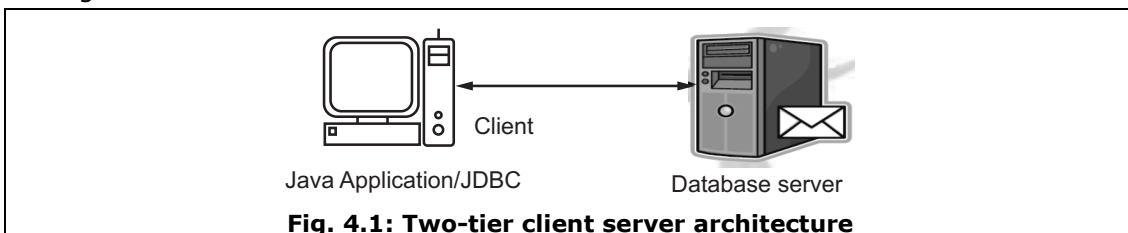
- Java offers several benefits to the software developer creating a front-end application for a database server.
- Java is 'Write Once Run Everywhere' language. This means that Java programs may be deployed without recompilation on any computer architectures and operating systems that possesses a Java Virtual Machine (JVM).
- Java interacts with database using a common database application programming interface called as JDBC (Java DataBase Connectivity).
- Java based clients (thin clients) that operate with minimum of hardware resources, yet run the complete Java environment are expected to cost around \$70 per seat.
- The specifications required a JDBC driver to be a translator that converted low-level proprietary DBMS messages to low-level messages understood by JDBC API and vice-versa. This meant that Java programmer could use high level Java data-objects defined in the JDBC API to write a routine that interacted with the DBMS.
- Java data objects convert the routine into low-level message that conform to the JDBC driver specification and send them to the JDBC driver.
- The JDBC driver translates the routine into low-level messages that understood and processed by DBMS.
- Open Database Connectivity (ODBC) is an open standard Application Programming Interface (API) for accessing a database. By using ODBC statements in a program, you can access files in a number of different databases, including Access, dBase, DB2, Excel and so on.

## 4.2 SOFTWARE ARCHITECTURES

### 4.2.1 Two-Tier Model

- The first generation of client-server architectures is called two-tiered.
- It contains two active components i.e., the client, which requests data, and the server, which delivers data.
- Basically, the application's processing is done separately for database queries and updates, and for user interface presentations.
- Usually the network binds the back end to the front end, although both tiers could be present on the same hardware.
- For example, hundreds or thousands of airline seat reservation applications can connect to a central DBMS to request, insert, or modify data. While the clients process the graphics and data entry validation, the DBMS does all the data processing.

- Actually, it is inadvisable to overload the database engine with data processing that is irrelevant to the server, thus some processing usually also happens on the clients.
- The two tiers are often called as Application layer includes JDBC drivers, business logic and user interfaces whereas second layer i.e. Database layer consists of RDBMS server.
- Fig. 4.1 shows two-tier architecture.

**Advantages:**

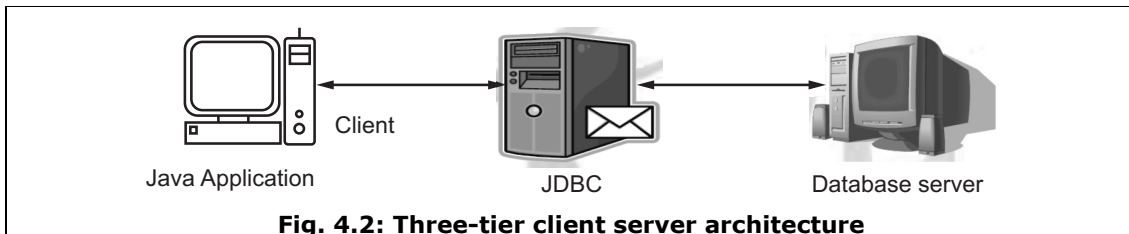
1. It is simple in design.
2. Client-side scripting offloads work onto the client.

**Disadvantages:**

1. It is not scalable, because each client requires its own database session.
2. It is inflexible.

**4.2.2 Three-Tier Model**

- Although the two-tiered architecture is common, another design is starting to appear more frequently. To avoid embedding the application's logic at both the database side and the client side, a third software tier may be inserted.
- In three-tiered architectures, most of the business logic is frozen in the middle tier. In this architecture, when the business activity or business rules change, only the middleware must be modified.
- Fig. 4.2 illustrates the three-tier architecture.
- Following the main components of a three-tier architecture:
  - 1. Client tier:** Typically, this is a thin presentation layer that may be implemented using a Web browser.
  - 2. Middle tier:** This tier handles the business or application logic. This may be implemented using a servlet engine such as Tomcat or an application server such as JBOSS. The JDBC driver also resides in this layer.
  - 3. Data source layer:** This component includes the RDBMS.



**Fig. 4.2: Three-tier client server architecture**

**Advantages:**

1. It is flexible because it can change one part without affecting others.
2. It can connect to different databases without changing code.
3. In this architecture, business logic is clearly separated from the database.
4. Performance can be improved by separating the application server and database server.

**Disadvantage:**

1. Higher complexity.
2. Higher maintenance.
3. Lower network efficiency.
4. More parts to configure and buy. So time consuming and costly.

### **4.2.3 JDBC**

- JDBC stands for Java DataBase Connectivity.
- It refers to several things, depending on context:
  1. It's a specification for using data sources in Java applets and applications.
  2. It's an API for using low-level JDBC drivers.
  3. It's an API for creating the low-level JDBC drivers, which do the actual connecting/transacting with data sources.
  4. It's based on the X/Open SQL Call Level Interface (CLI) that defines how client/server interactions are implemented for database systems.
- The JDBC defines every aspect of making data-aware Java applications and applets. The low-level JDBC drivers perform the database-specific translation to the high-level JDBC interface.
- This interface is used by the developer so he does not need to worry about the database-specific syntax when connecting to and querying different databases.
- The JDBC is a package, much like other Java packages such as java.awt.
- It is not currently a part of the standard Java Developer's Kit (JDK) distribution, but it is slated to be included as a standard part of the general Java API as the java.sql package. Soon after its official incorporation into the JDK and Java API, it will also become a standard package in Java-enabled Web browsers, though there is no definite time frame for this inclusion.

- The exciting aspect of the JDBC is that the drivers necessary for connection to their respective databases do not require any pre installation on the clients. A JDBC driver can be downloaded along with an applet.

#### 4.2.3.1 JDBC Architecture

- The JDBC is two-dimensional. The reasoning for the split is to separate the low-level programming from the high-level application interface.
- The low level programming is the JDBC driver. The idea is that database vendors and third-party software vendors will supply pre-built drivers for connecting to different databases.
- JDBC drivers are quite flexible. They can be local data sources or remote database servers.
- The implementation of the actual connection to the data source/database is left entirely to the JDBC driver.
- The structure of the JDBC includes following key concepts:
  1. The goal of the JDBC is a DBMS independent interface, a "generic SQL database access framework," and a uniform interface to different data sources.
  2. The programmer writes only one database interface; using JDBC, the program can access any data source without recoding.
- Fig. 4.3 shows architecture of JDBC.

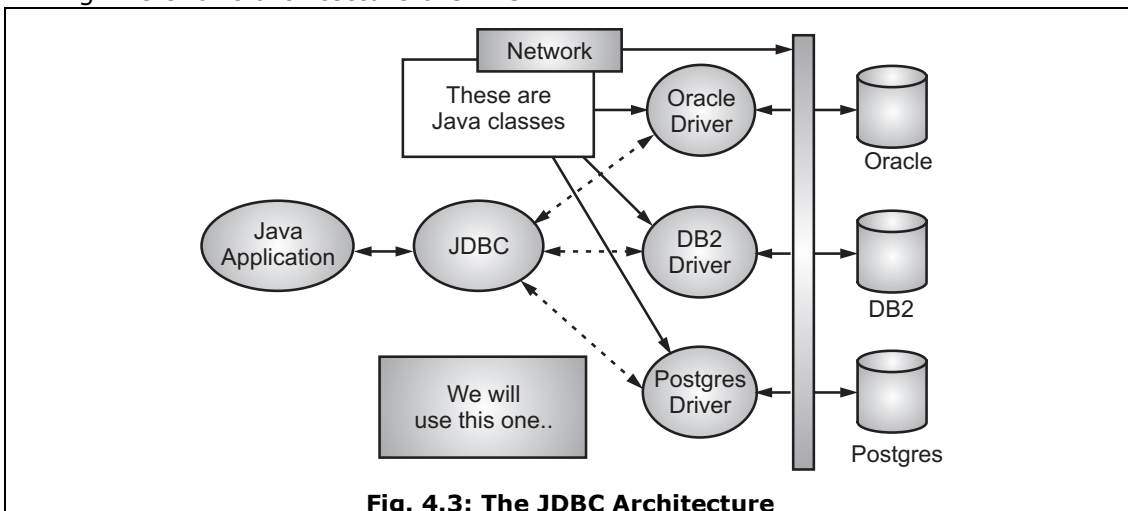
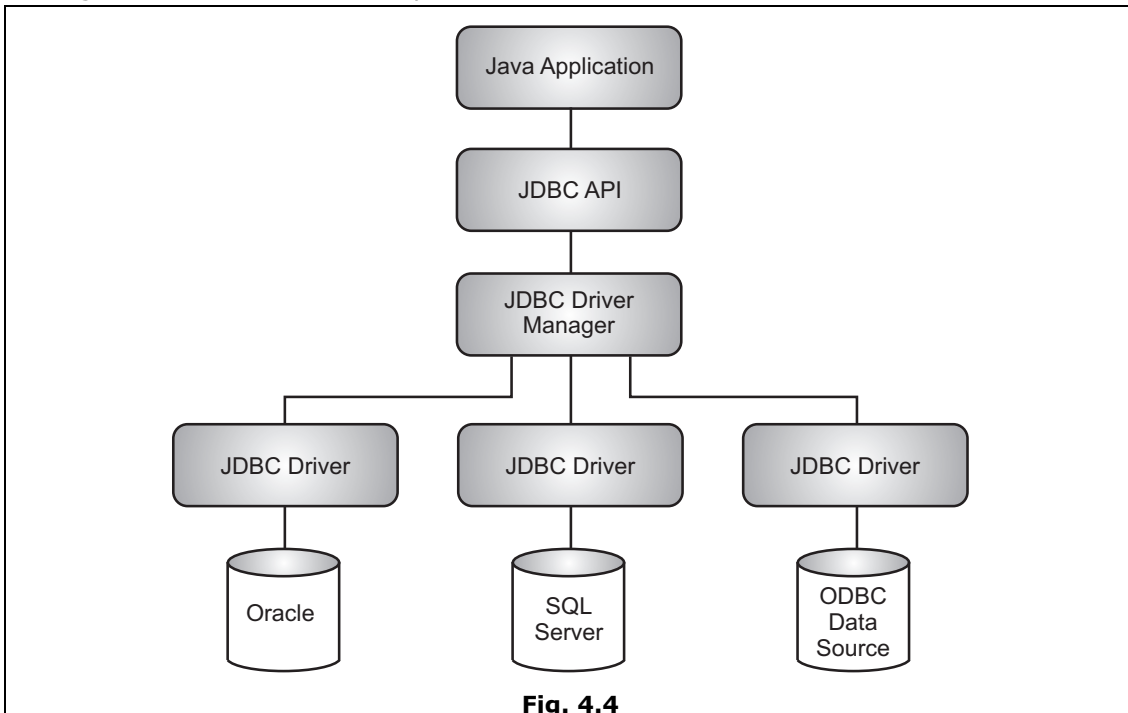


Fig. 4.3: The JDBC Architecture

#### 4.2.3.2 JDBC Components

- Java database connectivity (JDBC) is a driver that is used to communicate with database and is specially designed for java programs.
- JDBC API provides classes and interfaces to handle request made by user and response made by database.

- Fig. 4.4 shows various components of JDBC.



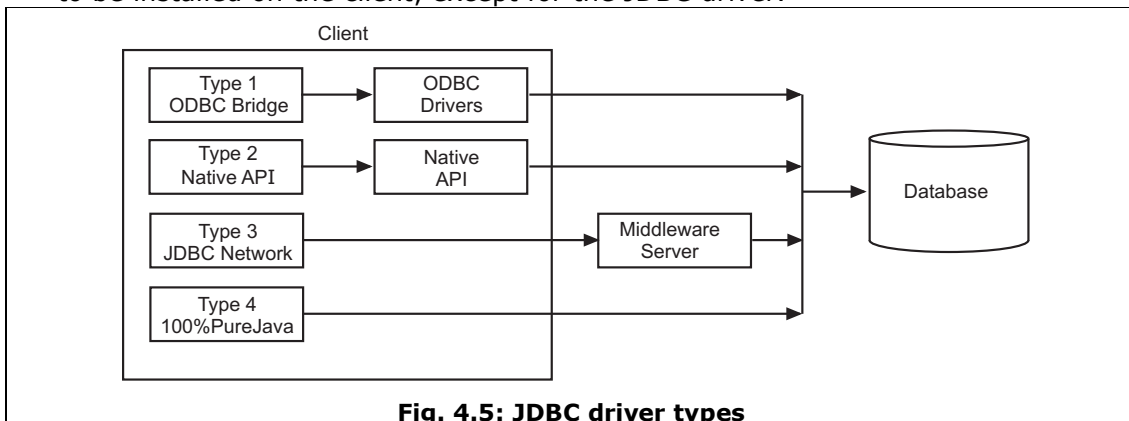
**Fig. 4.4**

- Various JDBC components in Fig. 4.4 are listed below:
  - 1. Application:** JDBC API provides the application to manage JDBC connection. The application where the JDBC methods are used to execute the SQL statements and get the results. In its simplest form it would be a java program.
  - 2. JDBC Driver API:** This supports JDBC manager-to-driver connection.
  - 3. Driver manager:** The main purpose of this component is to load the specific drivers for the user application. The following tasks are performed by this component
    - (i) Locate driver for a specific database, and
    - (ii) Process initialization calls for JDBC.
  - 4. Driver:** We have explain the different types of drivers in Section 4.2.3.3.
  - 5. Data source:** The data source is what user application interacts with to get the results. This is generally the database. For example, Microsoft Access database or Oracle database.

#### **4.2.3.3 JDBC Drivers**

- Sun Microsystems has defined four categories of JDBC drivers. The categories delineate the differences in architecture for the drivers. One difference between architectures lies in whether a given driver is implemented in native code or in Java code.

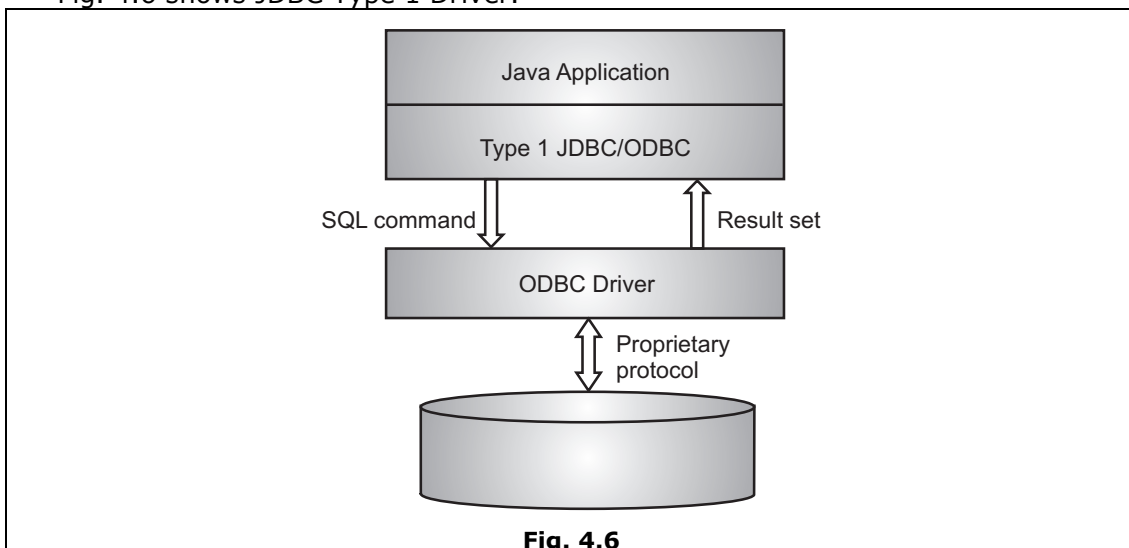
- Native code means whatever machine code is supported by a particular hardware configuration. For example, a driver may be written in C and then compiled to run on a specific hardware platform. Another difference lies in how the driver makes the actual connection to the database.
- Fig. 4.5 provides a high-level overview of the various types of drivers.
- Types 1 and 2 rely heavily on additional software, (typically C/C++ DLLs) installed on the client computer to provide database connectivity.
- Java and JDBC use these components to interact with the database.
- Types 3 and 4 are pure Java implementations and require no additional software to be installed on the client, except for the JDBC driver.



**Fig. 4.5: JDBC driver types**

### 1. Type 1 Driver: JDBC/ODBC Bridge:

- This type uses bridge technology to connect a Java client to a third-party API such as Open DataBase Connectivity (ODBC).
- Sun's JDBC-ODBC bridge is an example of a Type 1 driver.
- These drivers are implemented using native code. This driver connects Java to a Microsoft ODBC data source.
- Fig. 4.6 shows JDBC Type 1 Driver.



**Fig. 4.6**

- This driver typically requires the ODBC driver to be installed on the client computer and normally requires configuration of the ODBC data source.
- The bridge driver was introduced primarily to allow Java programmers to build data-driven Java applications before the database vendors had Type 3 and Type 4 drivers.

**Advantages:**

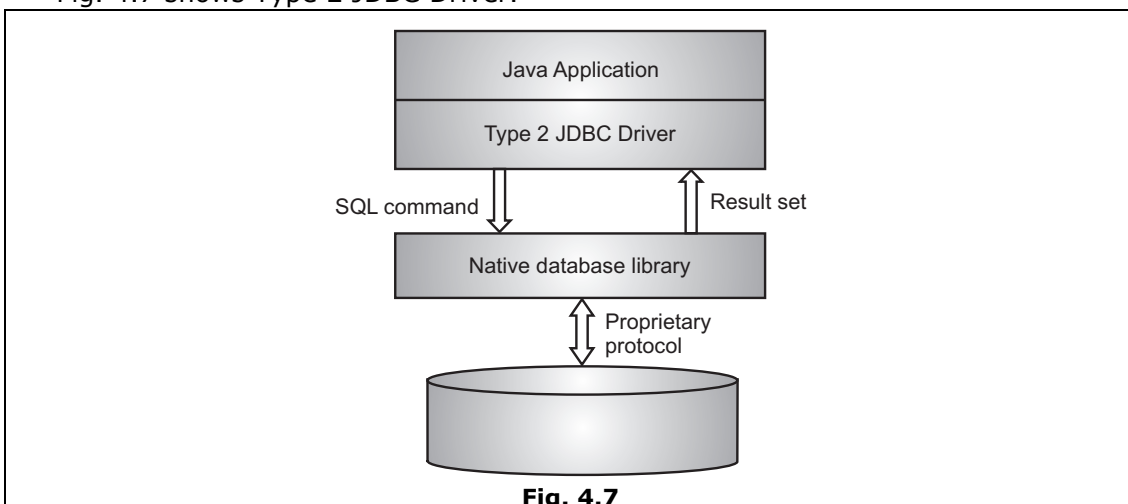
- (i) Easy to use.
- (ii) Allow easy connectivity to all database supported by the ODBC Driver.
- (iii) It is free there is no any money charge.

**Disadvantages:**

- (i) Slow execution time.
- (ii) Dependent on ODBC Driver.
- (iii) Uses Java Native Interface (JNI) to make ODBC call.
- (iv) ODBC drivers must also be loaded on the target machine.
- (v) Translation between JDBC and ODBC affects performance.
- (vi) Consumes more time.
- (vii) It is not a platform independent.

**2. Type 2 Driver: Native API Driver:**

- This type of driver wraps a native API with Java classes.
- The Oracle Call Interface (OCI) driver is an example of a Type 2 driver. Because a Type 2 driver is implemented using local native code, it is expected to have better performance than a pure Java driver.
- These drivers enable JDBC programs to use database-specific APIs (normally written in C or C++) that allow client programs to access databases via the Java Native Interface.
- This driver type translates JDBC into database-specific code. Type 2 drivers were introduced for reasons similar to the Type 1 ODBC bridge driver.
- Fig. 4.7 shows Type 2 JDBC Driver.



**Advantages:**

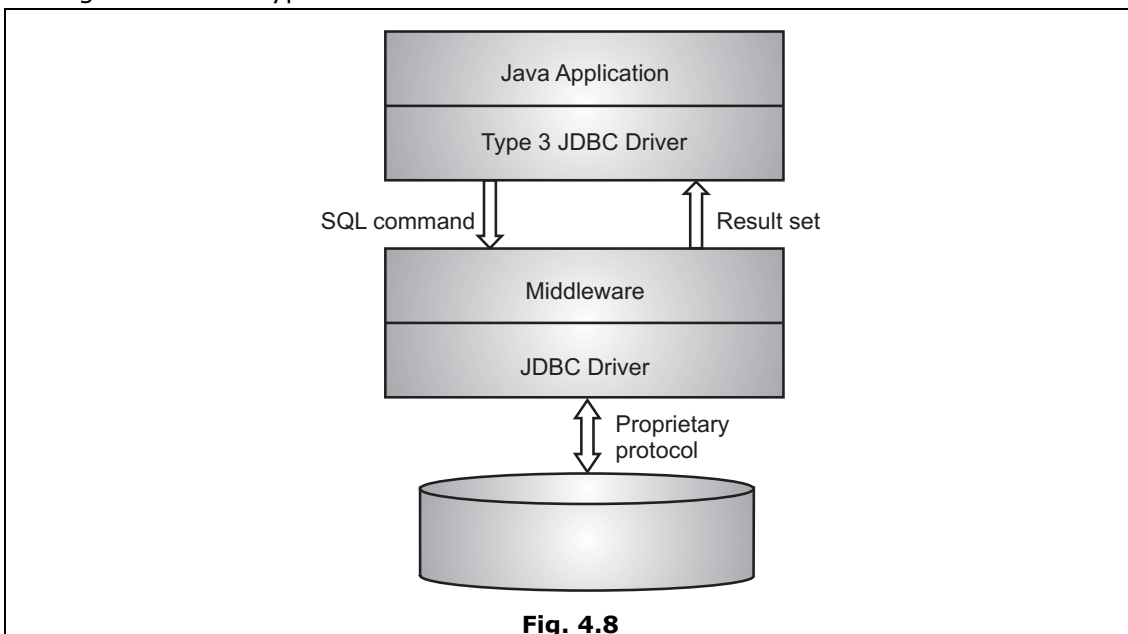
- (i) Faster as compared to Type-1 Driver.
- (ii) Contains additional features.

**Disadvantages:**

- (i) Requires native library.
- (ii) Increased cost of application.

**3. Type 3 Driver: Network Protocol, Pure Java Driver:**

- These drivers take JDBC requests and translate them into a network protocol that is not database specific.
- These requests are sent to a server, which translates the database requests into a database-specific protocol.
- Fig. 4.8 shows Type 3 JDBC Driver.



- This type of driver communicates using a network protocol to a middle tier server. The middle tier in turn communicates to the database.
- Oracle does not provide a Type 3 driver. They do, however, have a program called Connection Manager that, when used in combination with Oracle's Type 4 drivers, acts as a Type 3 driver in many respects.

**Advantages:**

- (i) It does not require any native library to be installed.
- (ii) It has database independency.
- (iii) It provide facility to switch over from one database to another database.

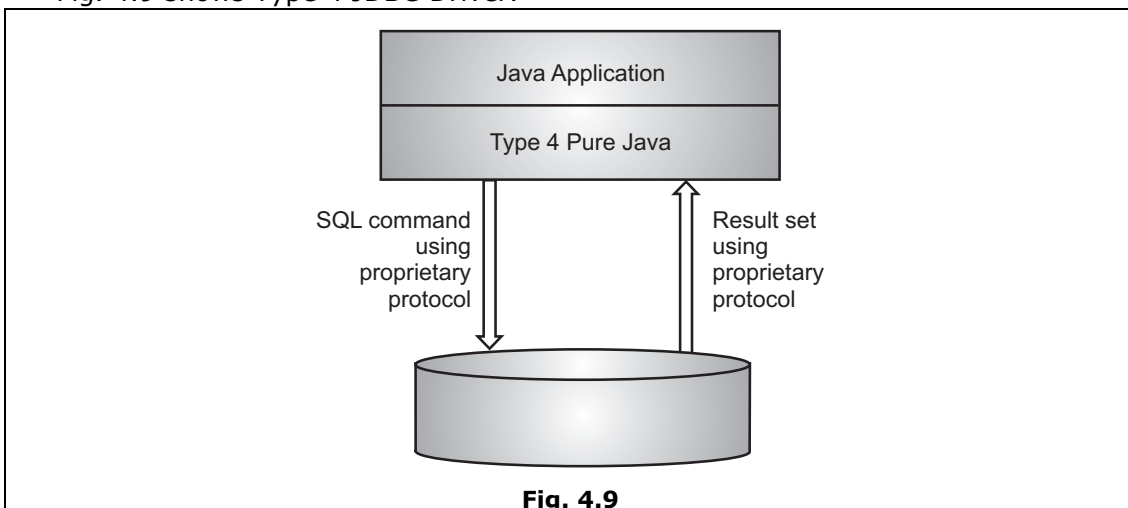
- (iv) Type 3 drivers do not require any native binary code on the client.
- (v) They do not need client installation.
- (vi) They support several networking options, such as HTTP tunneling.

**Disadvantages:**

- (i) Slow due to increase number of network call.
- (ii) They can be difficult to set up since, the architecture is complicated by the network interface.
- (iii) It is costliest JDBC driver.

**4. Type 4 Driver: Native Protocol, Pure Java Driver:**

- These convert JDBC requests to database-specific network protocols, so that Java programs can connect directly to a database.
- This type of driver, written entirely in Java, communicates directly with the database. No local native code is required.
- Oracle's thin driver is an example of a Type 4 driver.
- Fig. 4.9 shows Type 4 JDBC Driver.

**Fig. 4.9****Advantages:**

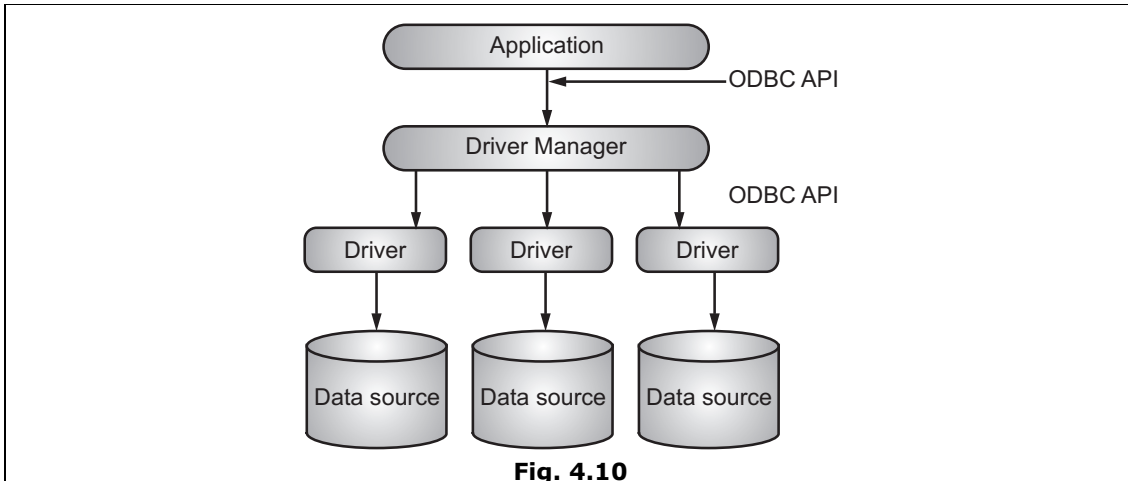
- (i) They are completely written in Java to achieve platform independence and eliminate deployment administration issues. It is most suitable for the web.
- (ii) Number of translation layers is very less i.e., type 4 JDBC drivers don't have to translate database requests to ODBC or a native connectivity interface or to pass the request on to another server, performance is typically quite good.
- (iii) You do not need to install special software on the client or server. Further, these drivers can be downloaded dynamically.

**Disadvantage:**

- (i) With type 4 drivers, the user needs a different driver for each database.

#### 4.2.4 ODBC

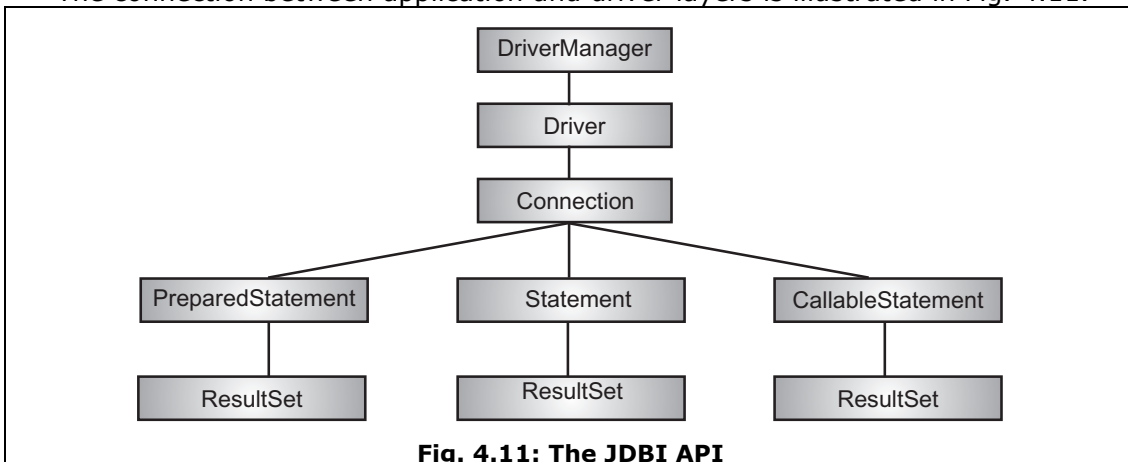
- ODBC (Open DataBase Connectivity) is a standard programming language middleware API for accessing DataBase Management Systems (DBMS).
- The designers of ODBC aimed to make it independent of database systems and operating systems.
- An application written using ODBC can be ported to other platforms, both on the client and server side, with few changes to the data access code.
- ODBC accomplishes DBMS independence by using an ODBC driver as a translation layer between the application and the DBMS.
- The application uses ODBC functions through an ODBC driver manager with which it is linked, and the driver passes the query to the DBMS.
- An ODBC driver can be thought of as analogous to a printer or other driver, providing a standard set of functions for the application to use, and implementing DBMS-specific functionality.
- Fig. 4.10 shows the ODBC architecture. The ODBC architecture has four components:
  1. The **Application** performs processing and calls ODBC functions to submit SQL statements and retrieve results.
  2. The **Driver Manager** is a DLL provided on a Windows platform as part of the ODBC components. It manages certain tasks common to all ODBC clients. For example, it manages loading and unloading of driver DLLs, and creation and maintenance of pointers to driver functions so you do not have to use LoadLibrary and GetProcAddress for each driver you want to use. It performs some basic error checking before a call is forwarded to the driver and also implements certain functions like SQLDataSources, SQLDrivers, and SQLGetFunctions within itself.
  3. The **Driver processes** ODBC function calls, submits SQL requests to a specific data source, and returns results to the application. If necessary, the driver modifies an application's request so that the request conforms to syntax supported by the associated DBMS.
  4. The **Data source** consists of the data the user wants to access and its associated operating system, DBMS, and network platform (if any) used to access the DBMS.



## 4.2.5 Other Applications

### 4.2.5.1 JDBC API

- JDBC is a database connectivity API that is a part of java enterprise APIs provided by Sun Microsystems. JDBC defines a set of API.
- The JDBC API is contained in two packages named java.sql and javax.sql.
- The java.sql package contains core Java objects of JDBC API. There are two distinct layers within the JDBC API; the application layer, which database application developers use and driver layer which the drivers vendors implement.
- The connection between application and driver layers is illustrated in Fig. 4.11.



- There are four 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 implementation is where the connection to the database is made. In most applications, Driver is accessed through Driver Manager class.

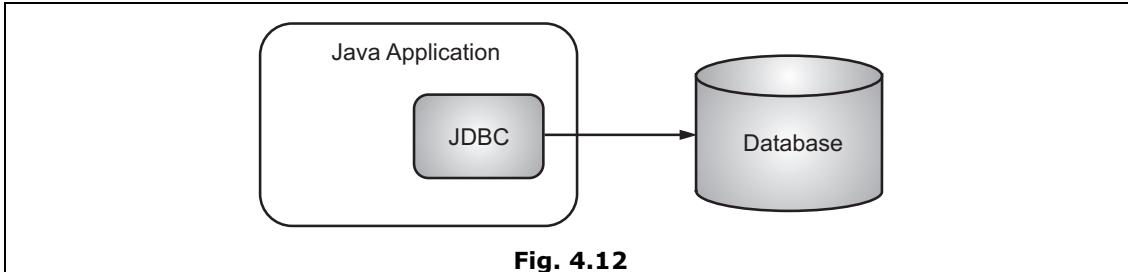
- The Table 4.1 lists the objects in the JDBC API.

**Table 4.1**

<b>Objects</b>	<b>Object Description</b>
Date	Provides an object that can accept database Date values.
DriverManager	Provides another way to make a connection to the database.
DriverPropertyInfo	Used to manage Driver objects.
Time	Provides an object that can accept database Time values.
Timestamp	Provides an object that can accept database Timestamp values.
Types	Provides a list of predefined integer values that identify the various data types that can be used in JDBC.

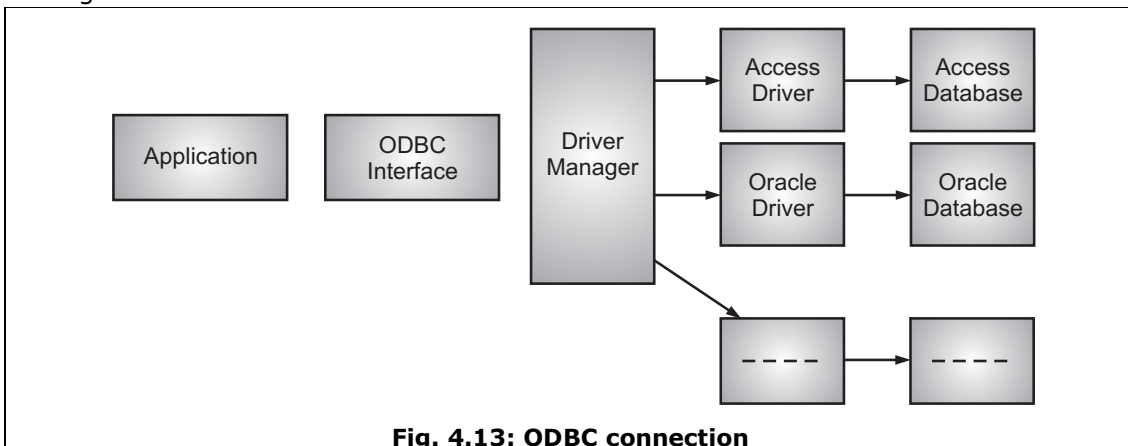
- Let us look above objects in detail:
  - 1. Date Object:** The Date object is inherited from the normal Java Date object, but provides methods for accessing the various values within the Data object.
  - 2. DriverManager Object:** The DriverManager object provides another way to make a connection to the database. The object is mainly used to manage JDBC Driver objects and can be used to create a connection to the database.
  - 3. DriverPropertyInfo Object:** The DriverPropertyInfo object is used mainly by advanced programmers to manage specific properties of a Driver object.
  - 4. Time Object:** The Time object is inherited from the Java Date object. It provides various methods for getting and setting the values from the object. It can be used to get Time data values from the database.
  - 5. Timestamp Object:** The Timestamp object can be used to get data values from the database that are of the timestamp data type. The object provides various methods for comparing the values of two different Timestamp objects.
  - 6. Types Object:** The Types object contains a listing of predefined integer values that identify each of the different data types available for use in JDBC applications. The values are used in different methods of the JDBC API to specify or identify the data type of particular data values.
- The three main **functions of JDBC** are as follows:
  1. Establishing a connection with a database or other tabular data source.
  2. Sending SQL commands to the database.
  3. Processing the results.
- The Java JDBC API enables Java applications to connect to relational databases via a standard API, so your Java applications become independent (almost) of the database the application uses.

- Fig. 4.12 shows Java application using JDBC to connect to a database.



#### 4.2.5.2 ODBC API

- ODBC (Open Database Connectivity) is an API (Application Programming Interface) provided by Microsoft for accessing the database.
- ODBC API uses structured Query Language or SQL as its database language. It provides functions to insert modify and delete data and obtain information regarding the database.
- Fig. 4.13 shows ODBC connection.



- The application could be a GUI program written in java, VC++ or in any other language.
- The application make use of ODBC to interact with the database.
- The driver manager is a part of Microsoft ODBC and is used to manage various drivers in the system including loading etc.
- In cases where the application uses multiple databases, it is the function of driver manager to make sure that the function calls gets diverted to the correct database management system.
- The following are the tasks performed within an application that uses ODBC:
  - Establish a connection with the database.
  - Send the SQL statement to the data source.
  - Define the storage area to store the result set and also the data type of the result set.

- Trace errors if any and process them.
- Get the results.
- Control the transactions.
- Terminate the connection.

### 4.3 CONNECTING TO DATABASE

#### Accessing JDBC / ODBC Bridge with the Database:

- Before actual performing the Java database application, we associate the connection of database source using JDBC – ODBC Bridge. The steps are as follows:

**Step 1:** Go to Control Panel → Administrative Tools → Data Sources.

**Step 2:** Open Data Sources ODBC icon, (See Fig. 4.14).



**Fig. 4.14**

**Step 3:** Select the tab with heading “User DSN”.

**Step 4:** Click on ‘Add’ button.

**Step 5:** Select the appropriate driver as per the database to be used. (e.g. Microsoft ODBC driver for Oracle to access Oracle Database).

**Step 6:** Click finish button and the corresponding ODBC database setup window will appear.

**Step 7:** Type DSN name and provide the required information such as user name and password for the database (.mdb files) of Microsoft Access Database etc. and click on OK button.

**Step 8:** Our DSN name will get appeared in user data sources.

- There are six different steps to use JDBC in our Java application program. These can be shown in Fig. 4.15.

1. Load the driver
- ↓
2. Define and establish the Connection
- ↓
3. Create a Statement object
- ↓
4. Execute a query
- ↓
5. Process the results
- ↓
6. Close the connection

**Fig. 4.15**

- Let use see above steps in detail:

**Step 1: Loading the JDBC driver:**

- The JDBC drivers must be loaded before the Java application connects to the DBMS. The `Class.forName()` is used to load the JDBC driver.
- The Developer must write routine that loads the JDBC/ODBC Bridge. The BridgeDriver called `sun.jdbc.odbc.JdbcOdbcDriver`.
- It is done in following way:

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

**Step 2: Connect to the DBMS:**

- After loading the driver the application must get connected to DBMS. For this we use `DriverManager.getConnection()` method. The `DriverManager` is highest class in `Java.sql` hierarchy and is responsible for managing driver related information.
- The `DriverManager.getConnection()` method is passed the URL of the database and user ID and password required by the database. The URL is the string object that contains the driver name that is being accessed by the Java program.
- The `DriverManager.getConnection()` method returns `Connection` interface that is used throughout the process to reference the database.
- The signature of this method is:

```
Connection DriverManager.getConnection  
                (String url, String userID, String password);
```

Here, the URL format is specified as follows:

```
<protocol>:<subprotocol>:<dsn-name>
```

- The 'protocol' is a JDBC protocol that is used to read the URL. The 'subprotocol' is JDBC driver name and 'dsn-name' is the name of the database that we provided while creating JDBC Bridge through control panel.
- We use the following URL for our application:

```
jdbc:odbc:customer
```

Here, 'customer' is an example of DSN name given to our database. The user name and password are also provided at the time of creating DSN. It is not compulsory to provide the username and password. For example:

```
Connction con;  
con = DriverManager.getConnection("jdbc:odbc:customer","micro", "pitch");
```

**Step 3: Create Statement object:**

- The `createStatement()` method of `Connection` interface is used to create the `Statement` object which is then used to execute the query.
- For example:

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

**Step 4: Execute the query:**

- The `executeQuery()` method of `Statement` object is used to execute and process the query which returns the `ResultSet` object.
- `ResultSet` is the object which actually contains the result returned by the query.
- For example:

```
ResultSet rs = st.executeQuery("select * from customer");
```

Here, the 'customer' is neither database name nor DSN name but it is a table name.

**Step 5: Process the results:**

- The `ResultSet` object is assigned the results received from the DBMS after the query is processed.
- The `ResultSet` object consists of methods used to interact with data that is returned by the DBMS to a Java application program.
- For example, the `next()` method is used to proceed through out the result set. It returns `true`, if the data is available in result set to read.
- The `ResultSet` also contains several `getXxx()` methods to read the value from a particular column of current row.
- For example, `getString("name")` will read the value from column 'name' in the form of string. Instead of passing column name as parameter, we can pass column index as parameter also. Such as `getString(1)`.

**For example:**

```
String name;  
int age;  
do  
{  
    name = rs.getString("name");  
    age = rs.getInt("age");  
    System.out.println(name+"="+age);  
} while(rs.next());
```

**Step 6: Terminate the connection:**

- The `Connection` to the DBMS is terminated by using the `close()` method of the `Connection` object once a Java program has finished accessing the DBMS.
- The `close()` method throws an exception if a problem is encountered when disengaging the DBMS.
- For example:

```
con.close();
```

- The `close()` method of `Statement` object is used to close the statement object to stop the further processing.

### 4.3.1 Java Interfaces

- Java provides various interfaces for connecting to the database and executing SQL statements.
- Using the JDBC API interfaces, you can execute normal SQL statements, dynamic SQL statements.
- Following Table 4.2 displays the interfaces provided in the JDBC API.

**Table 4.2**

Interfaces	Interface description
Connection	The interface connects your Java application to the database.
DatabaseMetaData	Provides Java with information concerning the database Driver and interface that is built specifically for each individual vendor database.
PreparedStatement	An interface used to execute dynamic SQL statements.
ResultSet	An interface that accepts results from a SQL Select statement.
ResultSetMetaData	An interface that provides information on a result set returned from the database.
Statement	An interface for executing normal SQL statements.

- 1. Connection Interface:** The Connection interface is the object that provides your Java applications with a connection to the database. This object can be used to create all of the various Statement objects for executing SQL statements. It also enables you to set the transaction properties for the connection.
- 2. DatabaseMetaData Interface:** The DatabaseMetaData interface provides various methods for getting information about the database. The interface provides methods to get the listing of tables for a specific database as well as the primary keys, columns, and various other information for specified tables.
- 3. Driver Interface:** Driver interface object is a database-specific Driver object provided by the JDBC vendor. It contains specific information about connecting your Java application. It also provides information about the database (for example, the version information).
- 4. PreparedStatement Interface:** The PreparedStatement interface enables you to execute dynamic SQL statements. Dynamic SQL statements differ from the normal SQL statements in that the values in Dynamic statements are not known at the time of creation.

- 5. ResultSet Interface:** The ResultSet interface is the object that is created and used to get information from a SQL Select statements. A SQL Select statement return, a cursor, that is, used by the ResultSet interface to navigate through the result returned by the Select statements. It provides various methods for getting information from the different columns contained in the cursor.
- 6. ResultSetMetaData Interface:** The ResultSetMetaData interface enables you to get information about a returned result set. The ResultSetMetaData object is created from a ResultSet object and provides information specific to that object. It enables you to get the number of columns in the result set, the names and types of the columns, as well as other information pertaining to the returned ResultSet.
- 7. Statement Interface:** The Statement interface is created from the Connection object and can be used to execute standard SQL statements. The object provides three main methods: execute(), executeQuery() and executeUpdate(). These methods let you execute SQL queries and SQL updates. The executeQuery() method will return a ResultSet object. This object is the ancestor for the PreparedStatement.

### **4.3.2 Driver Interface**

- JDBC provides a programming-level interface for uniformly communicating with databases.
- A JDBC driver is a software component enabling a Java application to interact with a database.
- To connect with individual databases, JDBC (the Java Database Connectivity API) requires drivers for each database.
- The JDBC driver gives out the connection to the database and implements the protocol for transferring the query and result between client and database.
- The JDBC Driver interface provides vendor-specific implementations of the abstract classes provided by the JDBC API.
- Each vendors driver must provide implementations of the java.sql.Connection, Statement, PreparedStatement, CallableStatement, ResultSet and Driver.

### **4.3.3 Driver Manager Class**

- The JDBC Driver Manager is a very important class that defines objects which connect Java applications to a JDBC driver.
- Usually Driver Manager is the backbone of the JDBC architecture. It is very simple and small that is used to provide a means of managing the different types of JDBC database driver running on an application.

- The main responsibility of JDBC database driver is to load all the drivers found in the system properly as well as to select the most appropriate driver from opening a connection to a database.
- The Driver Manager also helps to select the most appropriate driver from the previously loaded drivers when a new open database is connected.
- The DriverManager class works between the user and the drivers. The task of the DriverManager class is to keep track of the drivers that are available and handles establishing a connection between a database and the appropriate driver.
- It even keeps track of the driver login time limits and printing of log and tracing messages. This class is mainly useful for the simple application, the most frequently used method of this class is DriverManager.getConnection().
- We can know by the name of the method that this method establishes a connection to a database.
- The DriverManager class maintains the list of the Driver classes. Each driver has to be get registered in the DriverManager class by calling the method DriverManager.registerDriver().
- By calling the Class.forName() method the driver class get automatically loaded. The driver is loaded by calling the Class.forName() method.
- JDBC drivers are designed to tell the DriverManager about themselves automatically when their driver implementation class get loads.
- This class has many methods. Some of the commonly used methods are given below:
  1. `deregisterDriver(Driver driver)`: It drops the driver from the list of drivers registered in the DriverManager class.
  2. `registerDriver(Driver driver)`: It registers the driver with the DriverManager class.
  3. `getConnection(String url)`: It tries to establish the connection to a given database URL.
  4. `getConnection(String url, Sting user, String password)`: It tries to establish the connection to a given database URL.
  5. `getConnection(String url, Properties info)`: It tries to establish the connection to a given database URL.
  6. `getDriver(String url)`: It attempts to locate the driver by the given string.
  7. `getDrivers()`: It retrieves the enumeration of the drivers which has been registered with the DriverManager class.

### 4.3.4 Connection Interface

- A Connection is the session between java application and database.
- The JDBC Connection interface provides methods to handle transaction processing, create objects for executing SQL statements.
- It also provides some basic error-handling methods.
- Because the Connection interface is an interface, it cannot directly be created in code; instead, the Connection interface is created by the DriverManager.getConnection() method.
- When the getConnection() method is called, it returns a Connection object that instantiates your Connection object as demonstrated in the following code:

```
Connection con = DriverManager.getConnection(URL);
```

- When you call the getConnection() method of the Driver object, you create a permanent connection to the database.
- This connection is a static connection to the database, it will remain open until the variable goes out of scope or until you explicitly close the connection.

#### 4.3.4.1 Creating/Establishing a Connection Object

- To create a Connection object, you must first import the Java class files that contain the JDBC objects and interfaces, i.e. java .sql. \*.
- In this example, you will first import one package.

```
import java .sql. *;
```

- Now create a Driver object using the following syntax:

```
String driverName = "sun.jdbc.odbc.JdbcOdbcDriver";
```

- You might be wondering what the driverName string was used for.
- In JDBC, you must specify a driver to use when you connect to a database.
- Here, sun.jdbc.odbc.JdbcOdbcDriver is the name of the driver we are using.
- Keep in mind that we are using the jdbc odbc bridge driver over here.
- To create the connection to the database, simply specify the URL of the database to which you want to connect (refer getConnection() method of DriverManager Object).

```
String URL ="jdbc:odbc:xyz";
```

```
Connection con = DriverManager.getConnection(URL);
```

- Almost all of the JDBC objects and interfaces throw the SQLException exception when they are created or their methods are called.
- You must catch this exception or throw it again to prevent the compiler from giving an error.

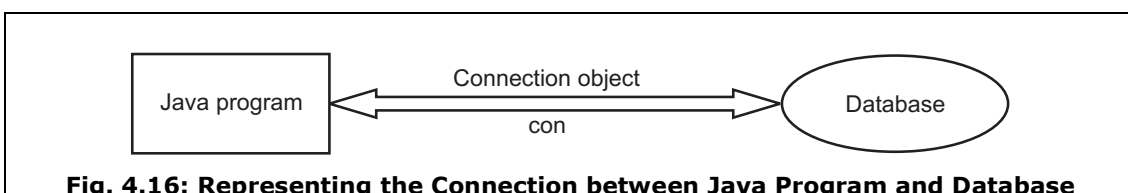
- By this we have created the connection object.
- Keep in mind that the name of the driver must be written as `sun.jdbc.odbc.JdbcOdbcDriver` otherwise, the object will not be created, and URL must be created properly.

**Program 4.1:**

```
import java.sql.*;
class ConnectionTest
{
public static void main(String args[])
{
    try
    {
        Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
        System.out.println("Driver loaded");
        String url="jdbc:odbc:xyz";
        Connection con = DriverManager.getConnection(url);
        System.out.println("connection to database created");
    }
    catch (SQLException e)
    {
        System.out.println("SQL error");
    }
    catch(Exception e)
    {
        System.out.println(" error");
    }
}
}
```

**Output:**

```
Driver loaded
connection to database created
```

**Fig. 4.16: Representing the Connection between Java Program and Database**

#### 4.3.4.2 Retrieving Information

- The Connection interface provides many methods to handle, informat retrieving processing, transaction processing and the creation of SQL statements. It also provides some basic error-handling methods.
- The list of methods is as follows:

Method	Description
<code>getAutoCommit</code>	Returns the value of the current AutoCommit parameter.
<code>getCatalog</code>	Returns a string that contains the current catalog for the connection.
<code>getMetaData</code>	Returns a DatabaseMetaData object that can be used to determine database features.
<code>getTransactionIsolation</code>	Returns the current isolation mode of the transaction associated with the Connection object.
<code>getWarnings</code>	Returns a SQLWarning object that contains the current warning for the Connection object.
<code>isClosed</code>	Returns a Boolean true if the connection is closed or a Boolean false if the connection is open.
<code>isReadOnly</code>	Returns a Boolean true if the connection is readonly or cannot be updated and a Boolean false if the connection is not read only or can be updated.
<code>nativeSQL</code>	Returns the SQL statement as the JDBC driver presents it to the database.
<code>prepareStatement</code>	Returns a PreparedStatement object that can be used execute dynamic SQL statements.
<code>Rollback</code>	Issues a rollback to the database. Useful only if AutoCommit = FALSE.
<code>setAutoCommit</code>	Sets AutoCommit to the Boolean variable passed.
<code>setCatalog</code>	Sets the current connection catalog to a different database catalog.
<code>setReadOnly</code>	Sets the connection to be read-only (no updates).
<code>setTransactionIsolation</code>	Sets the connection's transaction isolation value to the passed integer.

**Methods:****1. close():**

- The close() method of the Connection object explicitly closes the connection to the database.
- The connection will be closed automatically when the application is terminated.
- The **syntax** for the close() method is as follows:

```
connection.close();
```

**2. commit():**

- The commit() method issues a commit command to the database.
- The commit command tells the database to empty the transaction log and in effect, make permanent changes to the database since, the last commit or rollback.
- Whenever, SQL statements are executed against the database, the changes are stored in the database transaction log.
- Whenever, a database commit or rollback is executed, the transaction log is emptied.
- A commit will empty the transaction log and apply the changes to the database.
- A rollback will empty the transaction log and convert the database back to its state before the changes in the transaction log were executed.
- The commit() method applies to the database only when AutoCommit is set to Boolean false.
- If AutoCommit is set to Boolean true, then a commit is implicitly applied after the execution of every SQL statement.
- The syntax for committing a transaction is as follows:

```
Connection.setAutoCommit(false)
    ..some successful database processing
connection.commit()
```

**3. createStatement():**

- The createStatement() method of the Connection object creates a Statement object and returns it as the return value.
- The Statement object is used to execute SQL statements and to get results. The Statement object is used to execute only static SQL statements.
- To execute dynamic SQL statements, use the PreparedStatement.
- The syntax for the createStatement() method is as follows:

```
Statement state = connection.createStatement();
```

- In terms of above example,

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

- Here, the SQL statements will be executed through the statement object i.e. state object carries SQL statement towards database where it will be executed.

#### 4. prepareCall():

- The prepareCall() method returns a CallableStatement object that contains the precompiled SQL stored procedure statement.
- Calling a stored procedure in this way is necessary only if the stored procedure uses any INOUT or OUT parameters.
- All other stored procedures can be executed with the normal Statement object.
- To create a CallableStatement object use the following syntax:

```
try
{
    String sql = "GetEmployees";
    CallableStatement c = connection.prepareCall(sql);
}
catch (SQLException e) {}
```

#### 5. prepareStatement:

- The prepareStatement() method creates a PreparedStatement object and returns it as a return value.
- The PreparedStatement object is used to execute dynamic SQL statements against the database.
- A dynamic SQL statement is a statement in which some of the parameters(values) in the statement are unknown when the statement is created.
- The parameters(values) are placed into the SQL statement as they are determined by the application.
- When all the parameters have been specified for the SQL statement, the dynamic SQL statement will be executed just as a static statement is executed.
- To create a dynamic SQL statement that takes a FirstName and LastName as parameters, use the following code:

```
try {
    String sql = "select * from tEmployee" + "where Firstname =?" +
                "and Lastname =?";
    PreparedStatement p = connection.prepareStatement(sql);
}
catch (SQLException e) { }
```

Here, the Dynamic SQL statements will be executed through the PreparedStatement object i.e. prestate object carries dynamic SQL statement towards database where it will be executed.

## 6. rollback():

- The rollback() method of the Connection object issues a SQL rollback command to the database.
- When a rollback is issued to the database, the database transaction log is emptied and all changes made to the database revert back to their original values.
- A rollback is useful only ifAutoCommit is set to false. IfAutoCommit is true, then a commit will be implicitly issued after every SQL statement.
- This implicit commit will empty the transaction log and make permanent all changes to the database.
- The code to perform a rollback is as follows:

```
connection.setAutoCommit (false);  
... some unsuccessful database processing  
connection.rollback();
```

## 7. setAutoCommit():

- The setAutoCommit() method enables you to determine whether transaction processing statements such as, commit and rollback will be handled by you or handled automatically by the database.
- If AutoCommit is turned on (true), then after every SQL statement a commit command will be sent to the database.
- This will immediately empty the transaction log and apply the changes to the database.
- Using a rollback after a SQL statement has executed with AutoCommit turned ON will not roll back the database because the changes have already been implicitly committed.
- Using manual transaction processing also enables your business logic to be more complete and robust.
- If you are updating information with AutoCommit turned ON and one SQL statement fails, then any previous information will remain in the database.
- This type of approach can create incomplete data. Using manual transaction processing, you could execute all SQL statements and, if all statements are successful, issue a commit to the database to make the changes permanent.
- This would ensure that all data in the database remains complete and accurate.
- To turn AutoCommit on, which is also the default for the Connection object, use the following code:

```
connection.setAutoCommit(true);
```

- To turn AutoCommit OFF and allow manual transaction processing control, use the following code:

```
connection.setAutoCommit(false);
```

**The Statement Interface:**

- In the previous section, you learned how to create a Connection object that lets you connect to the database.
- The Connection object does just that it connects you to the database. To execute SQL statements and get results back from the database, you must use the Statement object.
- The Statement object, much like the Connection object, cannot be directly created.
- As with the Connection object, you create a Statement object by assigning it the return value from a method of another object, in this case the Connection object.
- The createStatement() method returns a Statement object as a return value.
- You can use a Statement object to execute static SQL statements and get results back from SQL queries.
- A static SQL statement does not take any arguments to be complete.
- A dynamic statement, for example, is not complete until a certain number of arguments are passed into the SQL statement.
- A static SQL statement can be a select statement, delete statement, update statement, insert statement.
- The update, delete, and insert statements do not return any results; these procedures simply update data in the database and do not return any new data from the database.
- A select statement will return data from the database, in most cases.
- A stored procedure can be executed in both ways.
- A stored procedure can carry out any of the update statements on data, and it can also return a selection of results from the database.

**(i) Creating a Statement Object:**

- Creating a Statement object is a very simple procedure.
- createStatement() method of Connection interface returns a Statement object as its return value.
- To create a Statement object, you must first import all of the necessary classes that will be used. Then create the connection as,

```
Connection con =DriverManager.getConnection(URL);
```

- After creation of the connection object i.e. establishing the connection to the database, you can create the Statement object by calling the createStatement() method of the Connection object.
- The method returns a Statement object as shown.

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

**Program 4.2:**

```
import java.sql.*;
class StatementTest
{
public static void main(String args[])
{
    try
        {
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
            System.out.println("Driver loaded");
            String url="jdbc:odbc:xyz";
            Connection con = DriverManager.getConnection(url);
            System.out.println("connection to database created");
            Statement st =con.createStatement();
            System.out.println("Statement object created");
        }
    catch (SQLException e)
    {
        System.out.println("SQL error");
    }
    catch(Exception e)
    {
        System.out.println(" error");
    }
}
}
```

**Output:**

```
Driver loaded
connection to database created
Statement object created
```

---

**(ii) Statement Interface Methods:**

- The Statement interface provides many methods as shown here.

<b>Method</b>	<b>Description</b>
<code>cancel()</code>	Enables a thread outside of the thread executing the SQL statement to cancel the execution of the statement.
<code>clearWarnings()</code>	Clears the current warning for the Statement object.
<code>close()</code>	Closes the current Statement object.
<code>execute()</code>	Executes the Statement object. Used primarily when a SQL statement will return multiple result sets. i.e. create.
<code>executeQuery()</code>	Executes a SQL Select statement and returns a result set containing the records matching the criteria of the specified SQL statement. i.e. select.
<code>executeUpdate()</code>	Executes a SQL update statement which could be a SQL Update, Delete, or Insert.
<code>getMaxFieldSize()</code>	Returns the current maximum size that a field returned in a result set can be.
<code>getMaxRows()</code>	Returns the current maximum number of rows that a returned result set can contain.
<code>getMoreResults()</code>	Moves to the next result in the Statement object. Used for SQL statements that returns multiple results.
<code>getQueryTimeout()</code>	Returns the number of seconds that the JDBC driver will wait for the Statement object to execute a SQL statement.
<code>getResultSet()</code>	Returns the current result set.
<code>getUpdateCount()</code>	Returns the current result in multiple result statements.
<code>getWarnings()</code>	Returns a SQLWarning object that contains the current warning for the statement.
<code>setCursorName()</code>	Sets the Statement object to use the passed cursor.
<code>setEscapeProcessing()</code>	Turns on or off escape processing for the statement.
<code>setMaxFieldSize()</code>	Sets the maximum size of a field returned in a result set.
<code>setMaxRows()</code>	Sets the maximum number of rows that can be returned in a result set.
<code>setQueryTimeout()</code>	Sets the timeout a JDBC driver will wait for a Statement object to execute a SQL statement.

**Methods:****1. cancel():**

- The cancel() method of the Statement object enables you to cancel the execution of the statement after it has begun.
- The statement being executed will be stopped and will not continue. The syntax for the cancel() method is as follows:

```
statement.cancel();
```

**2. close():**

- The close() method of the Statement object enables you to explicitly close the Statement object.
- This releases the resources being used by both the JDBC driver objects as well as the resources being used on the database server to handle the execution of the statement and the return of results.
- The syntax for the close() method is as follows:

```
statement.close();
```

**3. execute():**

- The execute() method of the Statement object will execute the passed SQL statement against the database using the JDBC driver.
- The execute() method is used primarily when a SQL statement will return multiple result sets.
- This occurs mainly in stored procedures where the stored procedure will return different selection results.
- To illustrate the need to use the execute() method, suppose you have a stored procedure called pResults that returns a selection that contains all rows from tEmployee and tArticle.
- Because the stored procedure returns more than one result set, you must use the execute() method in place of the executeQuery() method.
- First, create a string with a call to the stored procedure as follows:

```
String sql = 'pResults';
```

- Then, create a Statement object and call the execute() method to execute the stored procedure and return the results to the Statement object.
- The execute() method will return a Boolean true if the first result set is a ResultSet object, and a Boolean false if the first result is an integer result.
- The syntax for calling the execute() method is as follows:

```
Statement sqlStatement = connection.createStatement();  
boolean result = sqlStatement.execute(sql);
```

#### 4. executeUpdate():

- The executeUpdate() method of the Statement object enables you to execute SQL update statements such as delete, insert and update.
- The method takes a String containing the SQL update statement and returns an integer that determines how many records were affected by the SQL statement.
- For example, create an application that will change all students who have the current first name Deepak to first name deepu.
- The following is the SQL statement to execute this:

```
update StudentInfo set FirstName = "Deepu" where Firstname
                                     = "Deepak"
```

- The syntax for calling the executeUpdate() method is very similar to the syntax you use to call the executeQuery() method.
- To execute the preceding SQL statement, use the following code.

```
String sql="update StudentInfo set FirstName = "Deepu" where
FirstName= "Deepak"
int records= sqlStatement.executeUpdate(sql);
```

- The records will contain the number of records that were affected by the update Statement.
- Therefore, the records variable will contain the total number of students who have the FirstName Deepak.

---

#### Program 4.3:

```
import java.sql.*;
class ExecuteUpdateTest
{
public static void main(String args[])
{
    try
    {
        int i;
        Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
        System.out.println("Driver loaded");
        String url="jdbc:odbc:xyz";
        Connection con = DriverManager.getConnection(url);
        System.out.println("connection to database created");
```

```
Statement state =con.createStatement();
System.out.println("Statement object created");
String sql= "Insert into tanni values('Tanmay',2)";
i=state.executeUpdate(sql);
System.out.println("Record inserted" +i);
String sql1= "Insert into tanni values('Riddhi',2)";
i=state.executeUpdate(sql1);
System.out.println("Record inserted" +i);
String sql2= "Insert into tanni values('Siddhi',2)";
i=state.executeUpdate(sql2);
System.out.println("Record inserted" +i);
}
catch (SQLException e)
{
    System.out.println("SQL error");
}
catch(Exception e)
{
    System.out.println(" error");
}
}
}
```

**Output:**

```
Driver loaded
connection to database created
statement object created
Record inserted 1
Record inserted 2
Record inserted 3
```

- 
- The above program insert the three records in the table tanni using execute update statement.
  - Please check in MS Access whether the table is created or not in your database file whose path you have specified while creating the DSN. You will find the following entries in the Table 4.3.

**Table 4.3: Representation of Values Inserted in table**

Name	Age
Tanmay	2
Riddhi	2
Siddhi	2

**5. executeQuery():**

- The executeQuery() method of the Statement object enables you to send SQL select statements to the database and to receive results from the database.
- Executing a query, in effect, sends a SQL select statement to the database and returns the appropriate results back in a ResultSet object.
- The executeQuery() method takes a SQL select statement as a parameter and returns a ResultSet object that contains all of the records that match the select statement's criteria.
- For Example, select all of the students from the database table StudentInfo and display the first name and last name of every student.
- The SQL select statement to get the first and last names of every student is as follows:

```
select FirstName, LastName from StudentInfo
```

- Taking what you have learned about creating a Connection and a Statement object, execute the preceding SQL statement using the preceding SQL select statement.
- The following is the syntax for this:

```
String sql = 'select FirstName, LastName from StudentInfo';  
ResultSet results = sqlStatement.executeQuery(sql);
```

- After you have executed the SQL select statement, loop through all rows returned in the results returned by the ResultSet object, and display the first and last names.
- The code for this is as follows:

```
String text = " ";  
while (results.next())  
{  
text+= results.getString(1)+ " " + results.getInt(2)+ '\n';  
}  
System.out.println(text);
```

- Notice, that a method called next() was called for the ResultSet object results.

- This method moves the ResultSet object to the next record.
- The ResultSet object initially places the record pointer on a blank record preceding the first record.
- The next() method will return a true as long as there is a record to go to, and a false if there are no more records, meaning that you are at the end of the records.

---

**Program 4.4:**

```
import java.sql.*;
class ExecuteQueryTest
{
public static void main(String args[])
{
    try
    {
        Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
        System.out.println("Driver loaded");
        String url="jdbc:odbc:xyz";
        Connection con = DriverManager.getConnection(url);
        System.out.println("connection to database created");
        Statement state =con.createStatement();
        System.out.println("Statement object created");
        String sql= "select Name, Age from tanni";
        ResultSet results =state.executeQuery(sql);
        String text = " ";
        while (results.next())
        {
            text += results.getString(1)+ " " + results .getInt (2)+ '\n' ;
        }
        System.out.println(text);
    }
    catch (SQLException e)
    {
        System.out.println("SQL error");
    }
    catch(Exception e)
    {
        System.out.println("error");
    }
}
}
```

**Output:**

```

Driver loaded
connection to database created
statement object created
Tanmay 2
Riddhi 2
Siddhi 2

```

- o The next() method we will "cover" in ResultSet.

**java.sql Package:**

- This package include classes and interface to perform almost all JDBC operation such as creating and executing SQL Queries.
- JDBC 4.0 API is mainly divided into two package:
  1. java.sql
  2. javax.sql
- Important classes and interface of java.sql package are given in following table.

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	Executed 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 instasnce of a driver with the DriverManager.
java.sql.DriverManager	This class manages database drivers.
java.sql.PreparedStatement	Used to create an 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.

**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 interfaces of javax.sql package are:

<b>Classes/interface</b>	<b>Description</b>
<code>javax.sql.ConnectionEvent</code>	Provide information about occurrence of event.
<code>javax.sql.ConnectionEventListener</code>	Used to register event generated by PooledConnection object.
<code>javax.sql.DataSource</code>	Represent the DataSource interface used in an application.
<code>javax.sql.PooledConnection</code>	Provide object to manage connection pools.

**4.3.5 Resultset Interface**

- A ResultSet provides access to a table of data.
- 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:
  - 1. Navigational methods:** Used to move the cursor around.
  - 2. Get methods:** Used to view the data in the columns of the current row being pointed to by the cursor.
  - 3. 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.
- A ResultSet object is usually generated by executing a Statement. The ResultSet object is actually a tabular data set; that is, it consists of rows of data organized in uniform columns.
- In JDBC, the Java program can see only one row of data at one time. The program uses the `next()` method to go to the next row.
- JDBC does not provide any methods to move backwards along the ResultSet or to remember the row positions (called bookmarks in ODBC).
- A ResultSet maintains a cursor pointing to its current row of data. Initially the cursor is positioned before the first row.

- The 'next' method moves the cursor to the next row. So in this program we are using while (rs.next()).
- The method rs.next() returns a Boolean value depending on the record set. If it reaches last record false is returned and loop lapses.
- For maximum portability, ResultSet columns within each row should be read in left-to-right order and each column should be read only once.
- The JDBC driver attempts to convert the underlying data to the specified Java type and returns a suitable Java value through the getXXX methods.
- Column names used as input to getXXX methods are case insensitive. When performing a getXXX using a column name, if several columns have the same name, then the value of the first matching column will be returned.
- The column name option is designed to be used when column names are used in the SQL query. For columns that are NOT explicitly named in the query, it is best to use column numbers. If column names are used, there is no way for the programmer to guarantee that they actually refer to the intended columns.
- A ResultSet is automatically closed by the Statement that generated it when that Statement is closed, re-executed, or used to retrieve the next result from a sequence of multiple results.

#### ResultSet Navigation Methods:

- There are several methods in the ResultSet interface that involve moving the cursor, including:

Sr. No.	Methods and Description
1.	public void beforeFirst() throws SQLException: Moves the cursor to just before the first row.
2.	public void afterLast() throws SQLException: Moves the cursor to just after the last row.
3.	public boolean first() throws SQLException: Moves the cursor to the first row.
4.	public void last() throws SQLException: Moves the cursor to the last row.
5.	public boolean absolute(int row) throws SQLException: Moves the cursor to the specified row.
6.	public boolean relative(int row) throws SQLException: Moves the cursor the given number of rows forward or backwards from where it currently is pointing.

**contd. ...**

7.	<code>public boolean previous() throws SQLException:</code> Moves the cursor to the previous row. This method returns false if the previous row is off the result set.
8.	<code>public boolean next() throws SQLException:</code> Moves the cursor to the next row. This method returns false if there are no more rows in the result set.
9.	<code>public int getRow() throws SQLException:</code> Returns the row number that the cursor is pointing to.
10.	<code>public void moveToInsertRow() throws SQLException:</code> Moves the cursor to a special row in the result set that can be used to insert a new row into the database. The current cursor location is remembered.
11.	<code>public void moveToCurrentRow() throws SQLException:</code> Moves the cursor back to the current row if the cursor is currently at the insert row; otherwise, this method does nothing.

**Important points**

- JDBC enables java application to connect to any database using various drivers and new set of java API objects and methods.
- A database is a collection of data that is organized so that it may be easily searched and updated.
- At physical level, database server stores database information in specific location within the particular files, directories and disk volume used by the server.
- The JDBC provides various objects for connecting to a database, executing SQL statements.
- The Structure Query Language (SQL) is the language for interacting with the database.
- Client/server is based on the idea that one computer specializing in information presentation displays the data stored and processed on a remote machine.
- In the two-tier model, a Java application is designed to interact directly with a database.
- In the three-tier model, the client typically sends requests to an application server, forming the middle tier.
- ODBC API uses structured Query Language or SQL as its database language. It provides functions to insert, modify and delete data and obtain information regarding the database.
- JDBC is a database connectivity API that is a part of java enterprise APIs provided by Sun microsystem. JDBC defines a set of API.
- JDBC defines a low-level API designed to support basic SQL functionality independently of any specific SQL implementation.

- The JDBC 2.0 API includes two packages: `java.sql`, known as the JDBC 2.0 core API, and `javax.sql`, known as the JDBC Standard Extension.
- JDBC is a standard interface for Java programs to access relational databases.
- JDBC-ODBC bridge category works with ODBC drivers supplied by your database vendor or a third party.
- JDBC Native API category requires an operating system-specific API that handles the database communications.
- Type 3 drivers use a three-tier approach for accessing databases.
- 100% Java driver is implemented entirely in Java and encapsulates the database-specific network protocols to communicate directly with the DBMS.
- The JDBC architecture is based on the three tier architecture design of business applications. In this design, the client communicates with an intermediate server that provides a layer of abstraction from the RDBMS.
- Java provides a variety of interfaces and objects for manipulating data and executing SQL statements against the database.
- Using the JDBC API interfaces, you can execute normal SQL statements, dynamic SQL statements.
- The `Connection` interface is the object that provides your Java applications with a connection to the database.
- The `DatabaseMetaData` interface provides various methods for getting information about the database.
- `Driver` interface object is a database-specific `Driver` object provided by the JDBC vendor.
- The `PreparedStatement` interface enables you to execute dynamic SQL statements.
- The `ResultSet` interface is the object that is created and used to get information from a SQL `Select` statements.
- The `ResultSetMetaData` interface enables you to get information about a returned result set.
- The `Statement` interface is created from the `Connection` object and can be used to execute standard SQL statements.

---

**Practice Questions**

---

1. What is JDBC?
2. Explain two and three tier architectures diagrammatically.
3. With the help of diagram describe architecture of JDBC.
4. What is the difference between a connection and statement?
5. Differentiate between `Statement` and `PreparedStatement` interface.
6. Create an application that opens two connections to the database.

7. Write small program that opens a connection to database.
8. Explain various types of JDBC drivers.
9. Explain some functions of Statement Interface.
10. Explain the following functions of Statement Interface:
  - (a) execute()
  - (b) executeQuery()
  - (c) executeUpdate().
11. How do you create the object of the Statement Interface?
12. Explain the following function of PreparedStatement Interface:
  - (a) execute()
  - (b) executeQuery()
  - (c) executeUpdate().
13. How do you create the object of the PreparedStatement Interface?
14. Explain the next() method of ResultSet.
15. Explain how you will create the DataBaseMetaData object.
16. What are the advantages of using dynamic SQL statement?
17. Create a dynamic SQL statement that lets you specify the FirstName, LastName, MiddleName and RollNo. of Student.
18. What character do you use to indicate a dynamic parameter?
19. On what record is the cursor initially positioned in the result set?
20. Which method moves you to the next record in the result set?
21. Write an Application program /Applet to make connectivity with database using JDBC API
22. Write an Application program/Applet to send queries through JDBC bridge and handle result.

