UNIT 6) Working with Database & Connecting to Database
T.Y.I.T.

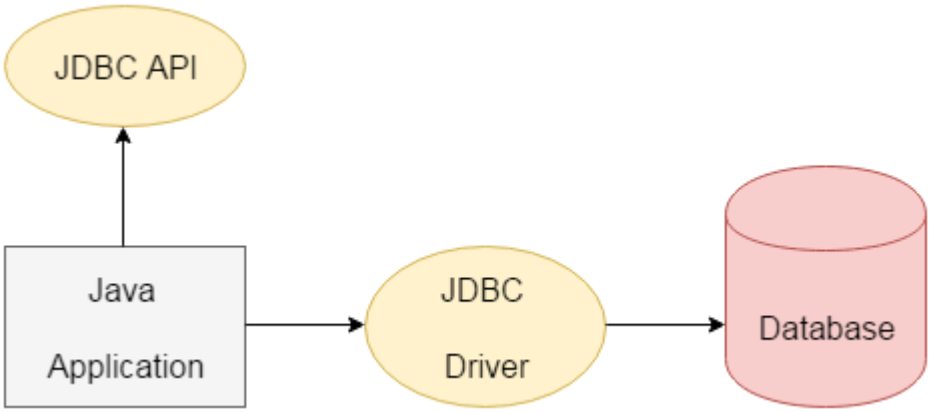
- **JDBC:**

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

We have discussed the above four drivers in the next chapter.

We can use JDBC API to access tabular data stored in any relational database. By the help of JDBC API, we can save, update, delete and fetch data from the database. It is like Open Database Connectivity (ODBC) provided by Microsoft.



The current version of JDBC is 4.3. It is the stable release since 21st September, 2017. It is based on the X/Open SQL Call Level Interface. The **java.sql** package contains classes and interfaces for JDBC API. A list of popular *interfaces* of JDBC API are given below:

- Driver interface
- Connection interface
- Statement interface
- PreparedStatement interface
- CallableStatement interface
- ResultSet interface
- ResultSetMetaData interface
- DatabaseMetaData interface
- RowSet interface

A list of popular *classes* of JDBC API are given below:

- DriverManager class
- Blob class
- Clob class
- Types class

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.

Do You Know

- How to connect Java application with Oracle and Mysql database using JDBC?
- What is the difference between Statement and PreparedStatement interface?
- How to print total numbers of tables and views of a database using JDBC?
- How to store and retrieve images from Oracle database using JDBC?
- How to store and retrieve files from Oracle database using JDBC?

What is API

API (Application programming interface) is a document that contains a description of all the features of a product or software. It represents classes and interfaces that software programs can follow to communicate with each other. An API can be created for applications, libraries, operating systems, etc.

Topics in Java JDBC

2) JDBC Drivers

In this JDBC tutorial, we will learn four types of JDBC drivers, their advantages and disadvantages.

3) 5 Steps to connect to the Database

UNIT 6) Working with Database & Connecting to Database
T.Y.I.T.

In this JDBC tutorial, we will see the five steps to connect to the database in Java using JDBC.

4) Connectivity with Oracle using JDBC

In this JDBC tutorial, we will connect a simple Java program with the Oracle database.

5) Connectivity with MySQL using JDBC

In this JDBC tutorial, we will connect a simple Java program with the MySQL database.

6) Connectivity with Access without DSN

Let's connect java application with access database with and without DSN.

7) DriverManager class

In this JDBC tutorial, we will learn what does the DriverManager class and what are its methods.

8) Connection interface

In this JDBC tutorial, we will learn what is Connection interface and what are its methods.

9) Statement interface

In this JDBC tutorial, we will learn what is Statement interface and what are its methods.

10) ResultSet interface

In this JDBC tutorial, we will learn what is ResultSet interface and what are its methods. Moreover, we will learn how we can make the ResultSet scrollable.

11) PreparedStatement Interface

In this JDBC tutorial, we will learn what is benefit of PreparedStatement over Statement interface. We will see examples to insert, update or delete records using the PreparedStatement interface.

12) ResultSetMetaData interface

In this JDBC tutorial, we will learn how we can get the metadata of a table.

13) DatabaseMetaData interface

In this JDBC tutorial, we will learn how we can get the metadata of a database.

14) Storing image in Oracle

Let's learn how to store image in the Oracle database using JDBC.

15) Retrieving image from Oracle

Let's see the simple example to retrieve image from the Oracle database using JDBC.

16) Storing file in Oracle

Let's see the simple example to store file in the Oracle database using JDBC.

17) Retrieving file from Oracle

Let's see the simple example to retrieve file from the Oracle database using JDBC.

18) CallableStatement

Let's see the code to call stored procedures and functions using Callable Statement.

19) Transaction Management using JDBC

Let's see the simple example to use transaction management using JDBC.

20) Batch Statement using JDBC

Let's see the code to execute batch of queries.

21) JDBC RowSet

Let's see the working of new JDBC RowSet interface.

Open Database Connectivity (ODBC)

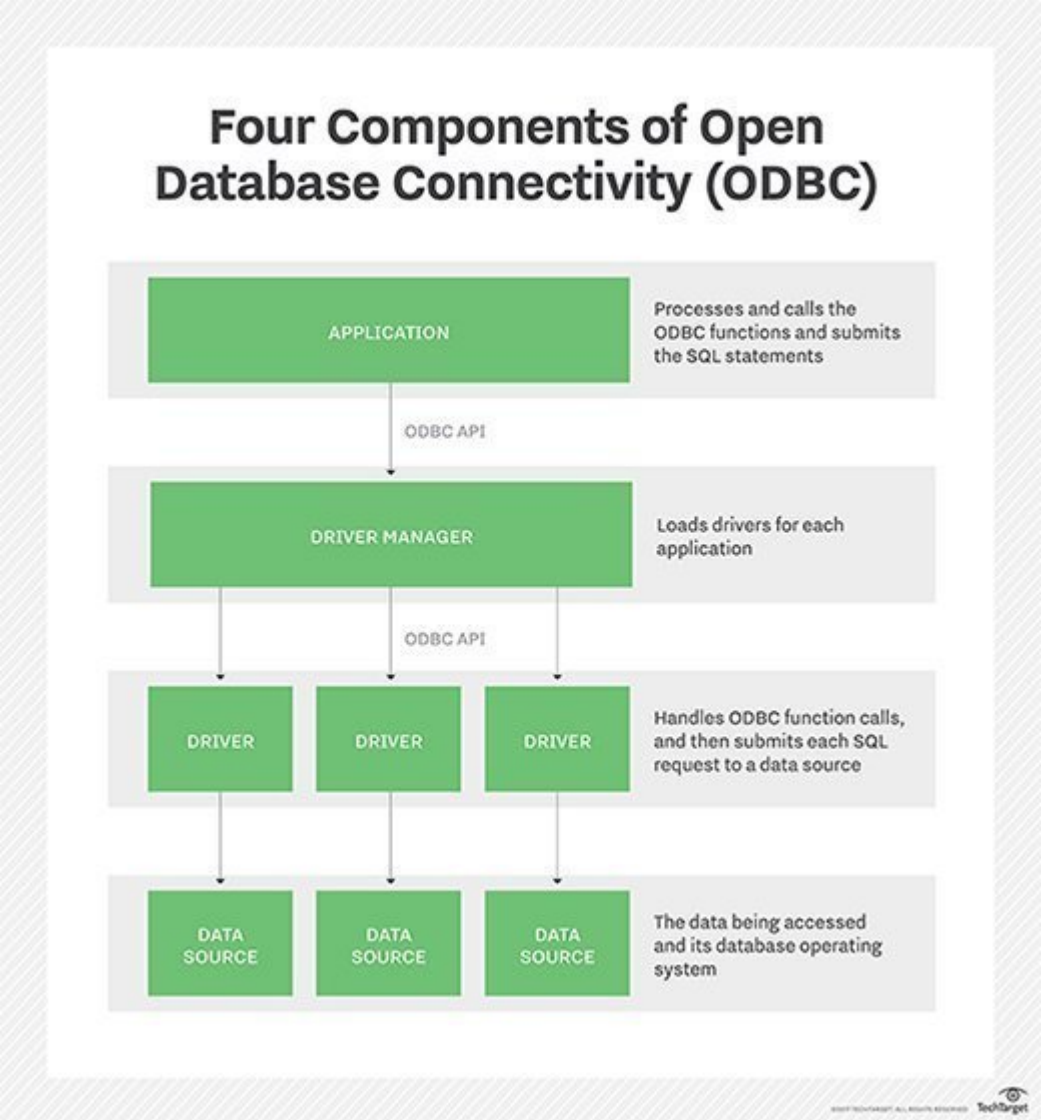
Open Database Connectivity (ODBC) is an open standard application programming interface (API) that allows application programmers to access any database.

The main proponent and supplier of ODBC programming support is Microsoft, but ODBC is based on and closely aligned with The Open Group standard Structured Query Language (SQL) Call-Level Interface (CLI). The Open Group is sponsored by many major vendors, including Oracle, IBM and Hewlett Packard Enterprise, and this consortium develops and manufactures The Open Group Architecture Framework (TOGAF). In addition to CLI specifications from The Open Group, ODBC also aligns with the ISO/IEC for database APIs.

How ODBC works

ODBC consists of four components, working together to enable functions. ODBC allows programs to use SQL requests that access databases without knowing the proprietary

interfaces to the databases. ODBC handles the SQL request and converts it into a request each database system understands.



Flowchart of the ODBC process

The four different components of ODBC are:

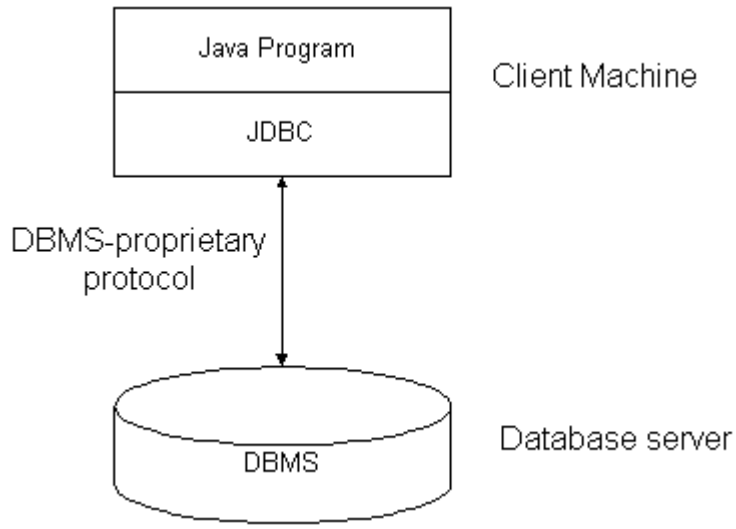
- **Application**: Processes and calls the ODBC functions and submits the SQL statements;
- **Driver manager:** Loads drivers for each application;
- **Driver**: Handles ODBC function calls, and then submits each SQL request to a data source; and
- **Data source**: The data being accessed and its database management system (DBMS) OS.

OBDC can also work with MySQL when its driver is called *MyODBC*. Sometimes, this is referred to as the MySQL Connecter/ODBC.

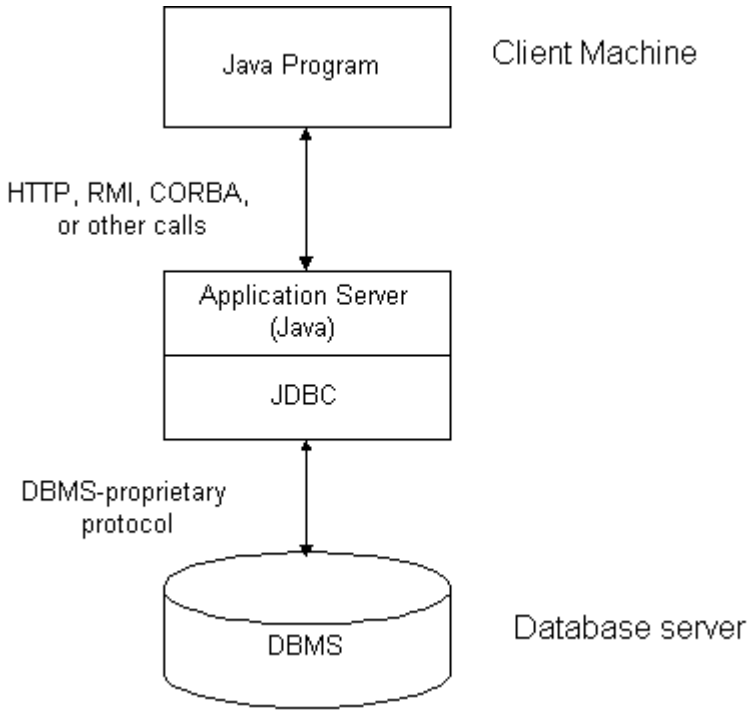
Two-tier and three-tier architecture

The JDBC API supports a two-tier and a three-tier architecture for database access.

In a two-tier model, a Java application/applet communicates directly with the database, via the JDBC driver. The Java application/applet and the database can be on the same machine, or the database can be on a server and the Java application/applet can be on a client machine using any network protocol.



In a three-tier model, a Java application/applet communicates with a middle tier component that functions as an application server. The application server talks to a given database using JDBC.



JDBC Architecture - 2 tier and 3 tier both

JDBC APIs support both the architectures - 2-tier and 3-tier models for accessing a relational database.

JDBC for a 2-tier architecture

In such an architecture, the Java application (or Applet) communicates directly with the data source (or database). The database may reside on the same machine or may be on another machine to which the clinet machine needs to be connected through a network. In the latter case it will be a clinet-server configuration and the server will hold the database. The client will send the statements to the database residing on the server and the result will be communicated back to the client.

In this architecture, the client machine will typically be a Java application and the server machine will have a Relational Database installed on it.

JDBC for a 3-tier architecture

In such an architecture, the client machine will send the database access statements to the middleware, which will then send the statements to the database residing on the third tier of the archirecture, which is normally referred to as the back end. The statements will be processed there and the result be returned back to the client through the middle tier. This approach will have all the advantages associated with the 3-tier architecture, such as better maintainability, easier deployment, scalability, etc.

This scenario will typically have a Java Applet or a Web Browser as the clinet tier, an Application Server as the Middle-tier, and a Relational DBMS as the Back-end.

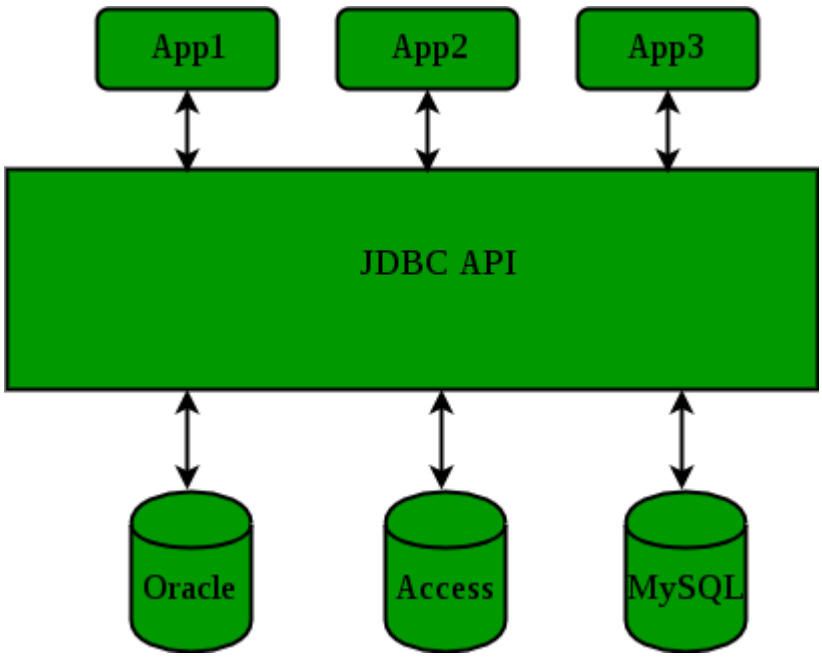
- **Connecting to Database:**
- **Driver Interface:**

JDBC Drivers

Java Database Connectivity (JDBC) is an application programming interface (API) for the programming language Java, which defines how a client may access any kind of tabular data, especially relational database. It is part of Java Standard Edition platform, from Oracle Corporation. It acts as a middle layer interface between java applications and database.
The JDBC classes are contained in the Java Package **java.sql** and **javax.sql**.
JDBC helps you to write Java applications that manage these three programming activities:

1. Connect to a data source, like a database.
2. Send queries and update statements to the database
3. Retrieve and process the results received from the database in answer to your query

Structure of JDBC



JDBC Drivers

JDBC drivers are client-side adapters (installed on the client machine, not on the server) that convert requests from Java programs to a protocol that the DBMS can understand. There are 4 types of JDBC drivers:

1. Type-1 driver or JDBC-ODBC bridge driver
2. Type-2 driver or Native-API driver
3. Type-3 driver or Network Protocol driver
4. Type-4 driver or Thin driver

Type-1 driver

Type-1 driver or JDBC-ODBC bridge driver uses ODBC driver to connect to the database. The JDBC-ODBC bridge driver converts JDBC method calls into the ODBC function calls. Type-1 driver is also called Universal driver because it can be used to connect to any of the databases.

- As a common driver is used in order to interact with different databases, the data transferred through this driver is not so secured.

- The ODBC bridge driver is needed to be installed in individual client machines.
- Type-1 driver isn't written in java, that's why it isn't a portable driver.

Type-2 driver

The Native API driver uses the client -side libraries of the database. This driver converts JDBC method calls into native calls of the database API. In order to interact with different database, this driver needs their local API, that's why data transfer is much more secure as compared to type-1 driver.

- Driver needs to be installed separately in individual client machines
- The Vendor client library needs to be installed on client machine.
- Type-2 driver isn't written in java, that's why it isn't a portable driver

Type-3 driver

The Network Protocol driver uses middleware (application server) that converts JDBC calls directly or indirectly into the vendor-specific database protocol. Here all the database connectivity drivers are present in a single server, hence no need of individual client-side installation.

- Type-3 drivers are fully written in Java, hence they are portable drivers.
- No client side library is required because of application server that can perform many tasks like auditing, load balancing, logging etc.
- Network support is required on client machine.
- Maintenance of Network Protocol driver becomes costly because it requires database-specific coding to be done in the middle tier.

Type-4 driver

Type-4 driver is also called native protocol driver. This driver interact directly with database. It does not require any native database library, that is why it is also known as Thin Driver.

- Does not require any native library and Middleware server, so no client-side or server-side installation.
- It is fully written in Java language, hence they are portable drivers.

Which Driver to use When?

- If you are accessing one type of database, such as Oracle, Sybase, or IBM, the preferred driver type is type-4.
- If your Java application is accessing multiple types of databases at the same time, type 3 is the preferred driver.
- Type 2 drivers are useful in situations, where a type 3 or type 4 driver is not available yet for your database.
- The type 1 driver is not considered a deployment-level driver, and is typically used for development and testing purposes only.

- **Driver Manager Class:**

The DriverManager class acts as an interface between user and drivers. It keeps track of the drivers that are available and handles establishing a connection between a database and the appropriate driver. The DriverManager class maintains a list of Driver classes that have registered themselves by calling the method DriverManager.registerDriver().

Useful methods of DriverManager class

Method	Description
1) public static void registerDriver(Driver driver):	is used to register the given driver with DriverManager.
2) public static void deregisterDriver(Driver driver):	is used to deregister the given driver (drop the driver from the list) with DriverManager.
3) public static Connection getConnection(String url):	is used to establish the connection with the specified url.
4) public static Connection getConnection(String url,String userName,String password):	is used to establish the connection with the specified url, username and password.

Connection interface

A Connection is the session between java application and database. The Connection interface is a factory of Statement, PreparedStatement, and DatabaseMetaData i.e. object of Connection can be used to get the object of Statement and DatabaseMetaData. The Connection interface provide many methods for transaction management like commit(), rollback() etc.

By default, connection commits the changes after executing queries.

Commonly used methods of Connection interface:

1) public Statement createStatement(): creates a statement object that can be used to execute SQL queries.

2) public Statement createStatement(int resultSetType,int resultSetConcurrency): Creates a Statement object that will generate ResultSet objects with the given type and concurrency.

3) public void setAutoCommit(boolean status): is used to set the commit status.By default it is true.

4) public void commit(): saves the changes made since the previous commit/rollback permanent.

5) public void rollback(): Drops all changes made since the previous commit/rollback.

6) public void close(): closes the connection and Releases a JDBC resources immediately.

Statement interface

The **Statement interface** provides methods to execute queries with the database. The statement interface is a factory of ResultSet i.e. it provides factory method to get the object of ResultSet.

Commonly used methods of Statement interface:

The important methods of Statement interface are as follows:

1) public ResultSet executeQuery(String sql): is used to execute SELECT query. It returns the object of ResultSet.

2) public int executeUpdate(String sql): is used to execute specified query, it may be create, drop, insert, update, delete etc.

3) public boolean execute(String sql): is used to execute queries that may return multiple
results.

4) public int[] executeBatch(): is used to execute batch of commands.

Example of Statement interface

Let's see the simple example of Statement interface to insert, update and delete the record.

```
1. import java.sql.*;
2. class FetchRecord{
3. public static void main(String args[])throws Exception{
4. Class.forName("oracle.jdbc.driver.OracleDriver");
5.
   Connection con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe","s
   ystem","oracle");
6. Statement stmt=con.createStatement();
7.
8. //stmt.executeUpdate("insert into emp765 values(33,'Irfan',50000)");
9.
   //int result=stmt.executeUpdate("update emp765 set name='Vimal',salary=10000 where id
   =33");
10. int result=stmt.executeUpdate("delete from emp765 where id=33");
11. System.out.println(result+" records affected");
12. con.close();
13. }}
```

- **The java.sql.package establishing connection:**

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.

Import JDBC Packages

The **Import** statements tell the Java compiler where to find the classes you reference in your code and are placed at the very beginning of your source code.

To use the standard JDBC package, which allows you to select, insert, update, and delete data in SQL tables, add the following *imports* to your source code −

```
import java.sql.* ;  // for standard JDBC programs
import java.math.* ; // for BigDecimal and BigInteger support
```

Register JDBC Driver

You must register the driver in your program before you use it. Registering the driver is the process by which the Oracle driver's class file is loaded into the memory, so it can be utilized as an implementation of the JDBC interfaces.

You need to do this registration only once in your program. You can register a driver in one of two ways.

Approach I - Class.forName()

The most common approach to register a driver is to use Java's **Class.forName()** method, to dynamically load the driver's class file into memory, which automatically registers it. This method is preferable because it allows you to make the driver registration configurable and portable.

The following example uses Class.forName( ) to register the Oracle driver −

```
try {
   Class.forName("oracle.jdbc.driver.OracleDriver");
}
catch(ClassNotFoundException ex) {
   System.out.println("Error: unable to load driver class!");
   System.exit(1);
}
```

You can use **getInstance()** method to work around noncompliant JVMs, but then you'll have to code for two extra Exceptions as follows −

```
try {
   Class.forName("oracle.jdbc.driver.OracleDriver").newInstance();
}
catch(ClassNotFoundException ex) {
   System.out.println("Error: unable to load driver class!");
   System.exit(1);
catch(IllegalAccessException ex) {
   System.out.println("Error: access problem while loading!");
```

```
  System.exit(2);
catch(InstantiationException ex) {
  System.out.println("Error: unable to instantiate driver!");
  System.exit(3);
}
```

Approach II - DriverManager.registerDriver()

The second approach you can use to register a driver, is to use the static **DriverManager.registerDriver()** method.

You should use the *registerDriver()* method if you are using a non-JDK compliant JVM, such as the one provided by Microsoft.

The following example uses registerDriver() to register the Oracle driver −

```
try {
  Driver myDriver = new oracle.jdbc.driver.OracleDriver();
  DriverManager.registerDriver( myDriver );
}
catch(ClassNotFoundException ex) {
  System.out.println("Error: unable to load driver class!");
  System.exit(1);
}
```

Database URL Formulation

After you've loaded the driver, you can establish a connection using the **DriverManager.getConnection()** method. For easy reference, let me list the three overloaded DriverManager.getConnection() methods −

- getConnection(String url)
- getConnection(String url, Properties prop)
- getConnection(String url, String user, String password)

Here each form requires a database **URL**. A database URL is an address that points to your database.

Formulating a database URL is where most of the problems associated with establishing a connection occurs.

Following table lists down the popular JDBC driver names and database URL.

RDBMS	JDBC driver name	URL format
MySQL	com.mysql.jdbc.Driver	**jdbc:mysql:**//hostname/ databaseName

ORACLE	oracle.jdbc.driver.OracleDriver	**jdbc:oracle:thin:@**hostname:port Number:databaseName
DB2	COM.ibm.db2.jdbc.net.DB2Driver	**jdbc:db2:**hostname:port Number/databaseName
Sybase	com.sybase.jdbc.SybDriver	**jdbc:sybase:Tds:**hostname: port Number/databaseName

All the highlighted part in URL format is static and you need to change only the remaining part as per your database setup.

Create Connection Object

We have listed down three forms of **DriverManager.getConnection()** method to create a connection object.

Using a Database URL with a username and password

The most commonly used form of getConnection() requires you to pass a database URL, a *username*, and a *password*:

Assuming you are using Oracle's **thin** driver, you'll specify a host:port:databaseName value for the database portion of the URL.

If you have a host at TCP/IP address 192.0.0.1 with a host name of amrood, and your Oracle listener is configured to listen on port 1521, and your database name is EMP, then complete database URL would be −

```
jdbc:oracle:thin:@amrood:1521:EMP
```

Now you have to call getConnection() method with appropriate username and password to get a **Connection** object as follows −

```
String URL = "jdbc:oracle:thin:@amrood:1521:EMP";
String USER = "username";
String PASS = "password"
Connection conn = DriverManager.getConnection(URL, USER, PASS);
```

Using Only a Database URL

A second form of the DriverManager.getConnection( ) method requires only a database URL −

```
DriverManager.getConnection(String url);
```

However, in this case, the database URL includes the username and password and has the following general form −

```
jdbc:oracle:driver:username/password@database
```

So, the above connection can be created as follows −

```
String URL = "jdbc:oracle:thin:username/password@amrood:1521:EMP";
Connection conn = DriverManager.getConnection(URL);
```

Using a Database URL and a Properties Object

A third form of the DriverManager.getConnection( ) method requires a database URL and a Properties object −

```
DriverManager.getConnection(String url, Properties info);
```

A Properties object holds a set of keyword-value pairs. It is used to pass driver properties to the driver during a call to the getConnection() method.

To make the same connection made by the previous examples, use the following code −

```
import java.util.*;

String URL = "jdbc:oracle:thin:@amrood:1521:EMP";
Properties info = new Properties( );
info.put( "user", "username" );
info.put( "password", "password" );

Connection conn = DriverManager.getConnection(URL, info);
```

Closing JDBC Connections

At the end of your JDBC program, it is required explicitly to close all the connections to the database to end each database session. However, if you forget, Java's garbage collector will close the connection when it cleans up stale objects.

Relying on the garbage collection, especially in database programming, is a very poor programming practice. You should make a habit of always closing the connection with the close() method associated with connection object.

To ensure that a connection is closed, you could provide a 'finally' block in your code. A *finally* block always executes, regardless of an exception occurs or not.

To close the above opened connection, you should call close() method as follows −

```
conn.close();
```

Explicitly closing a connection conserves DBMS resources, which will make your database administrator happy.

- **Retrieving information ResultSet interface:**

JDBC - 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.

Viewing a Result Set

The ResultSet interface contains dozens of methods for getting the data of the current row.

There is a get method for each of the possible data types, and each get method has two versions −

- One that takes in a column name.
- One that takes in a column index.

For example, if the column you are interested in viewing contains an int, you need to use one of the getInt() methods of ResultSet −

S.N.	Methods & Description
1	**public int getInt(String columnName) throws SQLException** Returns the int in the current row in the column named columnName.
2	**public int getInt(int columnIndex) throws SQLException** Returns the int in the current row in the specified column index. The column index starts at 1, meaning the first column of a row is 1, the second column of a row is 2, and so on.

Updating a Result Set

The ResultSet interface contains a collection of update methods for updating the data of a result set.

As with the get methods, there are two update methods for each data type −

- One that takes in a column name.
- One that takes in a column index.

For example, to update a String column of the current row of a result set, you would use one of the following updateString() methods −

S.N.	Methods & Description
1	**public void updateString(int columnIndex, String s) throws SQLException** Changes the String in the specified column to the value of s.
2	**public void updateString(String columnName, String s) throws SQLException** Similar to the previous method, except that the column is specified by its name instead of its index.

Commonly used methods of ResultSet interface

1) public boolean next():	is used to move the cursor to the one row next from the current position.
2) public boolean previous():	is used to move the cursor to the one row previous from the current position.
3) public boolean first():	is used to move the cursor to the first row in result set object.
4) public boolean last():	is used to move the cursor to the last row in result set object.
5) public boolean absolute(int row):	is used to move the cursor to the specified row number in the ResultSet object.
6) public boolean relative(int row):	is used to move the cursor to the relative row number in the ResultSet object, it may be positive or negative.
7) public int getInt(int columnIndex):	is used to return the data of specified column index of the current row as int.
8) public int getInt(String columnName):	is used to return the data of specified column name of the current row as int.
9) public String getString(int columnIndex):	is used to return the data of specified column index of the current row as String.
10) public String getString(String columnName):	is used to return the data of specified column name of the current row as String.

Retrieving and Modifying Values from Result Sets

The following method, CoffeesTable.viewTable outputs the contents of the COFFEES tables, and demonstrates the use of ResultSet objects and cursors:

```
public static void viewTable(Connection con, String dbName)
    throws SQLException {

    Statement stmt = null;
    String query =
        "select COF_NAME, SUP_ID, PRICE, " +
        "SALES, TOTAL " +
        "from " + dbName + ".COFFEES";

    try {
        stmt = con.createStatement();
        ResultSet rs = stmt.executeQuery(query);
        while (rs.next()) {
            String coffeeName = rs.getString("COF_NAME");
            int supplierID = rs.getInt("SUP_ID");
            float price = rs.getFloat("PRICE");
            int sales = rs.getInt("SALES");
            int total = rs.getInt("TOTAL");
            System.out.println(coffeeName + "\t" + supplierID +
                               "\t" + price + "\t" + sales +
                               "\t" + total);
        }
    } catch (SQLException e ) {
        JDBCTutorialUtilities.printSQLException(e);
    } finally {
        if (stmt != null) { stmt.close(); }
    }
}
```

A ResultSet object is a table of data representing a database result set, which is usually generated by executing a statement that queries the database. For example, the CoffeeTables.viewTable method creates a ResultSet, rs, when it executes the query through the Statement object, stmt. Note that a ResultSet object can be created through any object that implements the Statement interface, including PreparedStatement, CallableStatement, and RowSet.

You access the data in a ResultSet object through a cursor. Note that this cursor is not a database cursor. This cursor is a pointer that points to one row of data in the ResultSet. Initially, the cursor is positioned before the first row. The method ResultSet.next moves the cursor to the next row. This method returns false if the cursor is positioned after the last row. This method repeatedly calls the ResultSet.next method with a while loop to iterate through all the data in the ResultSet.

ResultSet Interface

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

ResultSet Types

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

The sensitivity of a ResultSet object is determined by one of three different ResultSet types:

- TYPE_FORWARD_ONLY: The result set cannot be scrolled; its cursor moves forward only, from before the first row to after the last row. The rows contained in the result set depend on how the underlying database generates the results. That is, it contains the rows that satisfy the query at either the time the query is executed or as the rows are retrieved.
- TYPE_SCROLL_INSENSITIVE: The result can be scrolled; its cursor can move both forward and backward relative to the current position, and it can move to an absolute position. The result set is insensitive to changes made to the underlying data source while it is open. It contains the rows that satisfy the query at either the time the query is executed or as the rows are retrieved.
- TYPE_SCROLL_SENSITIVE: The result can be scrolled; its cursor can move both forward and backward relative to the current position, and it can move to an absolute position. The result set reflects changes made to the underlying data source while the result set remains open.

The default ResultSet type is TYPE_FORWARD_ONLY.

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

ResultSet Concurrency

The concurrency of a ResultSet object determines what level of update functionality is supported.

There are two concurrency levels:

- CONCUR_READ_ONLY: The ResultSet object cannot be updated using the ResultSet interface.
- CONCUR_UPDATABLE: The ResultSet object can be updated using the ResultSet interface.

The default ResultSet concurrency is CONCUR_READ_ONLY.

UNIT 6) Working with Database & Connecting to Database
T.Y.I.T.