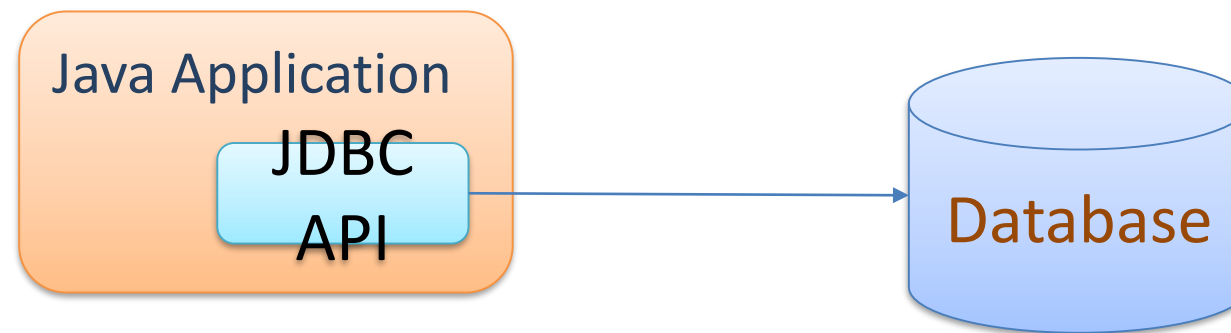


JDBC Basics

JDBC API

Introduction to JDBC

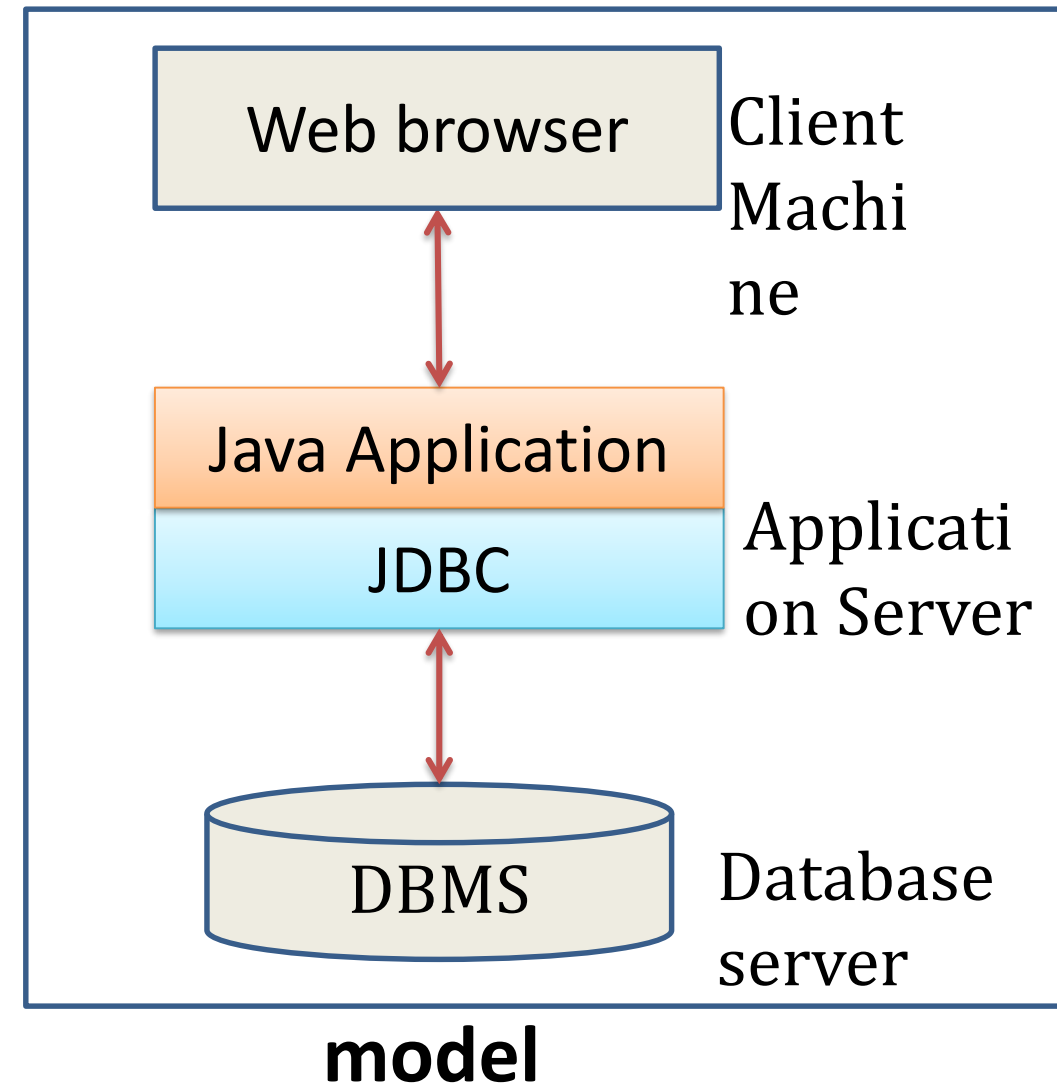
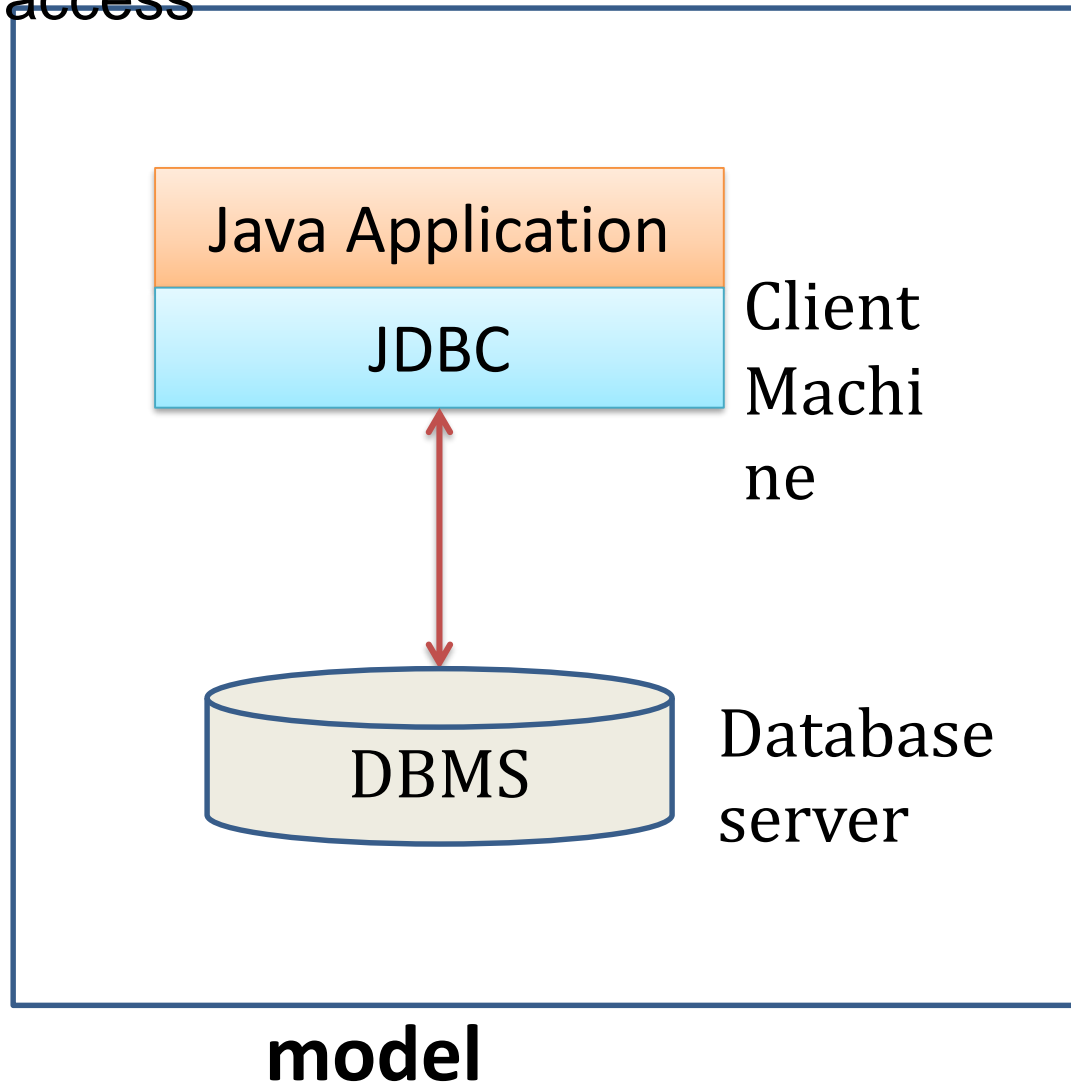
- JDBC stands for Java Database Connectivity
- Standard API which enables java applications to interact with relational databases
- JDBC API makes java application code independent of the database the application uses



- JDBC API defines interfaces and classes which standardize the way in which java application
 - Connect to Database
 - Execute queries
 - Retrieve results and navigate the results

JDBC Architecture

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



Steps for accessing database using JDBC



JDBC API classes and interfaces



Class class, Driver Interface

- **java.sql** and **javax.sql** packages define the classes and interfaces required for accessing data from a data source

DriverManager class

- **SQLException** is thrown in case of error while accessing the data source

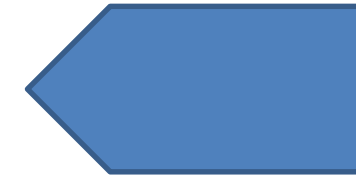
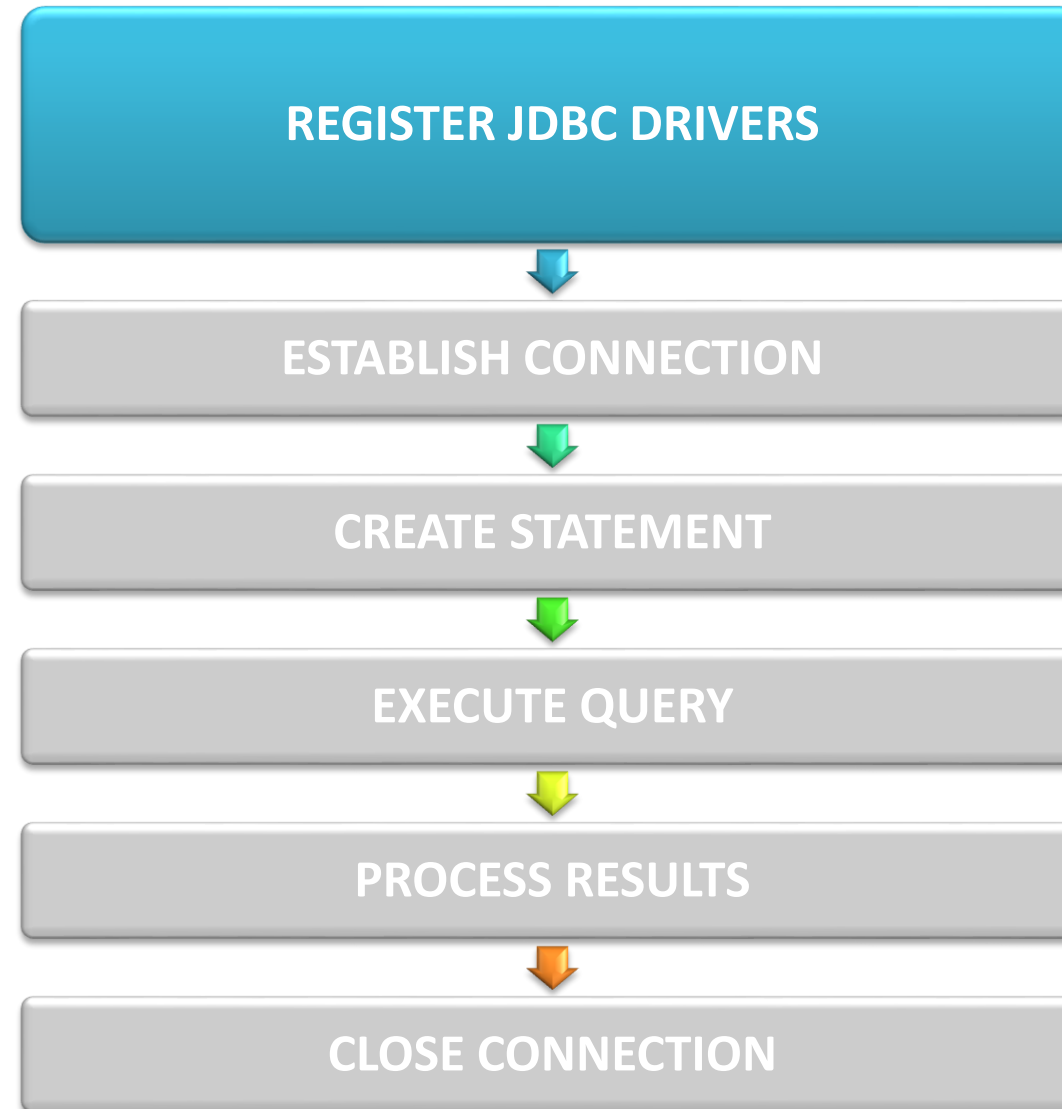
Connection Interface

- Every Database vendor providing java connectivity to their databases implements the JDBC API

Statement Interface

- The implemented API is provided as a jar file containing specific classes for the database

ResultSet Interface



JDBC Driver and its types

- JDBC Driver is a software component that enables java application to interact with the database

Type 1 : Bridge

- JDBC ODBC Bridge Driver

Type 2 : Native

- Native API Driver

Type 3: Middleware

- Network Protocol Driver

Type 4 : Pure

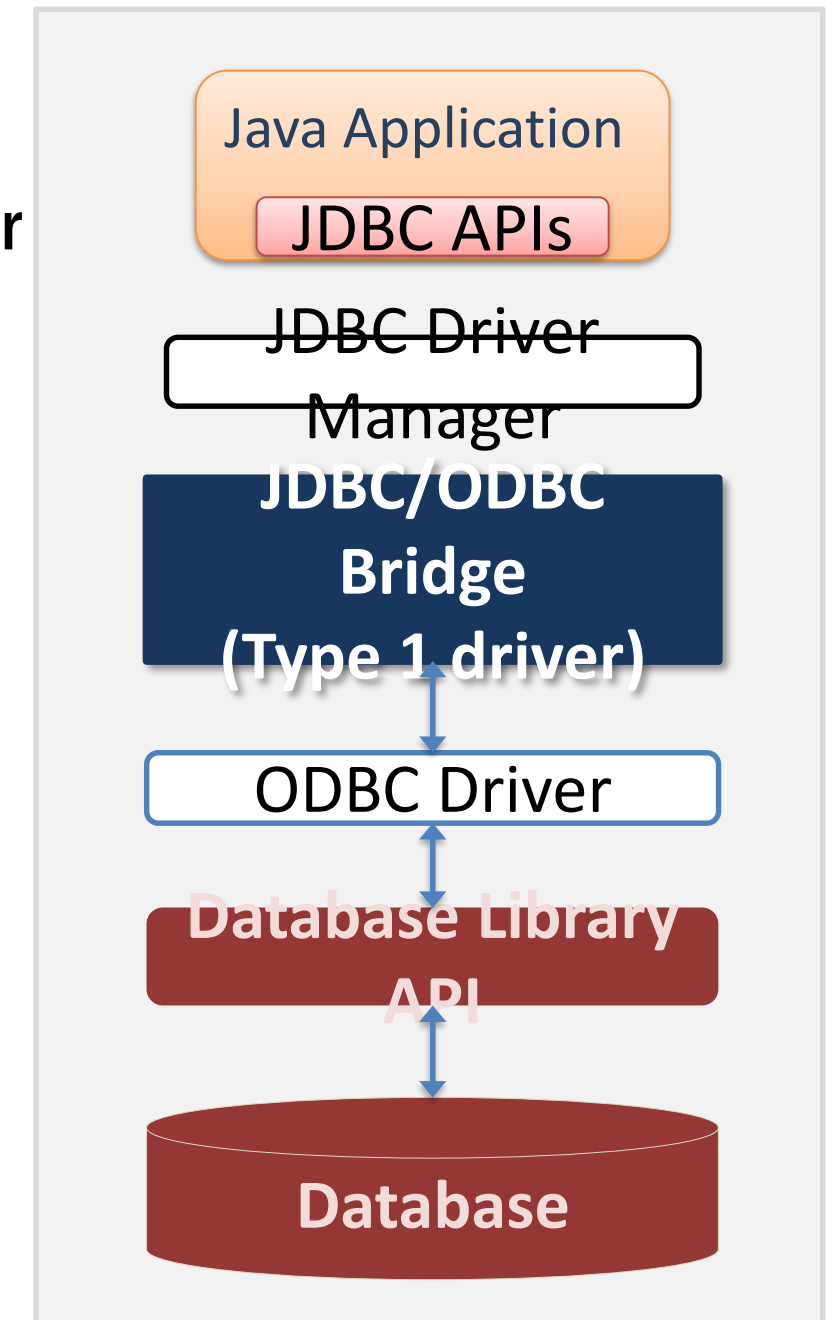
- Native Protocol Driver

Type-1: JDBC-ODBC Bridge Driver

- Translates JDBC calls into ODBC calls and sends to the ODBC driver
- ODBC requires configuring a Data Source Name (DSN)

- Allows access to almost any database which has ODBC driver

- Slowest of all drivers
- ODBC driver needs to be installed on the client machine
- Recommended only for experimental use or when no other alternative is available
- Not suitable for Web

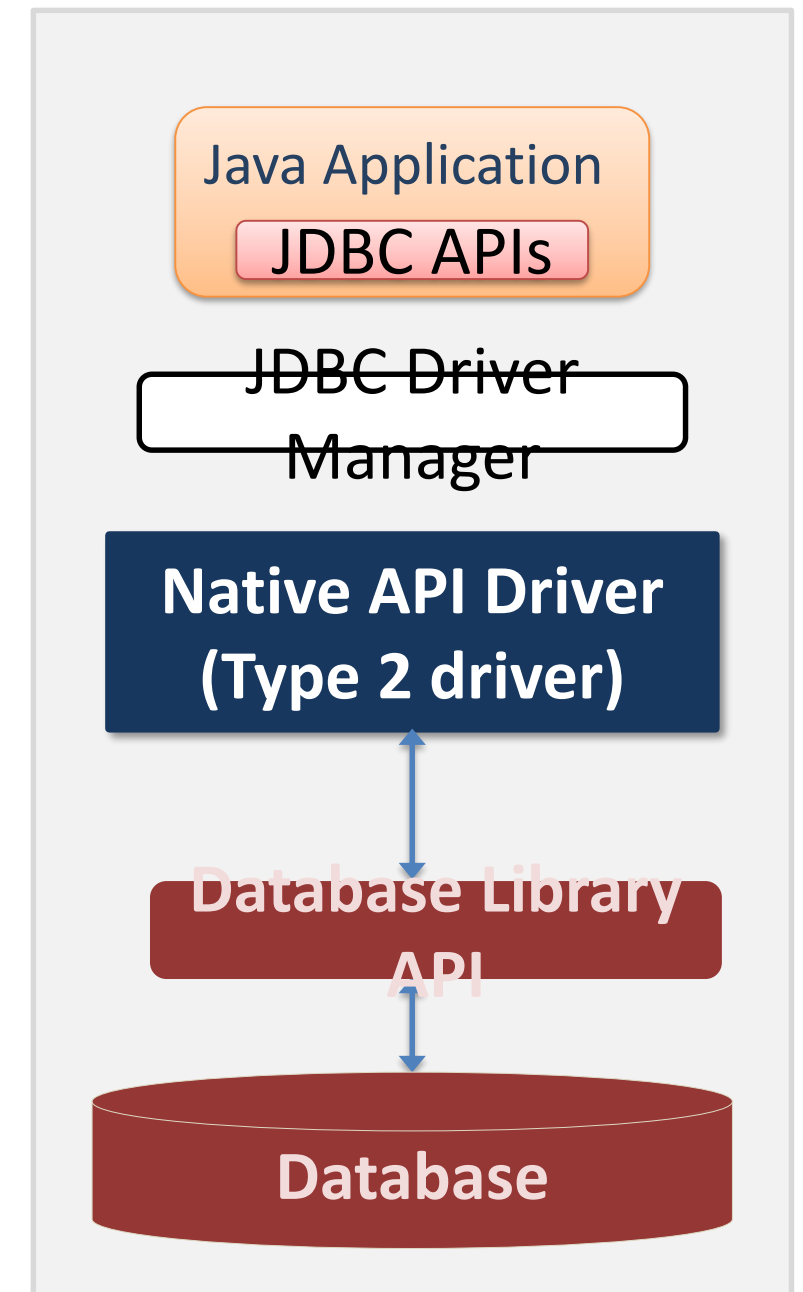


Type-2: Native API Driver

- Driver implementation uses the client-side libraries of the database
- It converts JDBC method calls into native calls of the database API

Ex. Oracle Call Interface (OCI) driver

- Offers better performance than the JDBC-ODBC Bridge
-
- The database client library needs to be installed on the client machine
 - It is platform dependent
 - Not suitable for Web

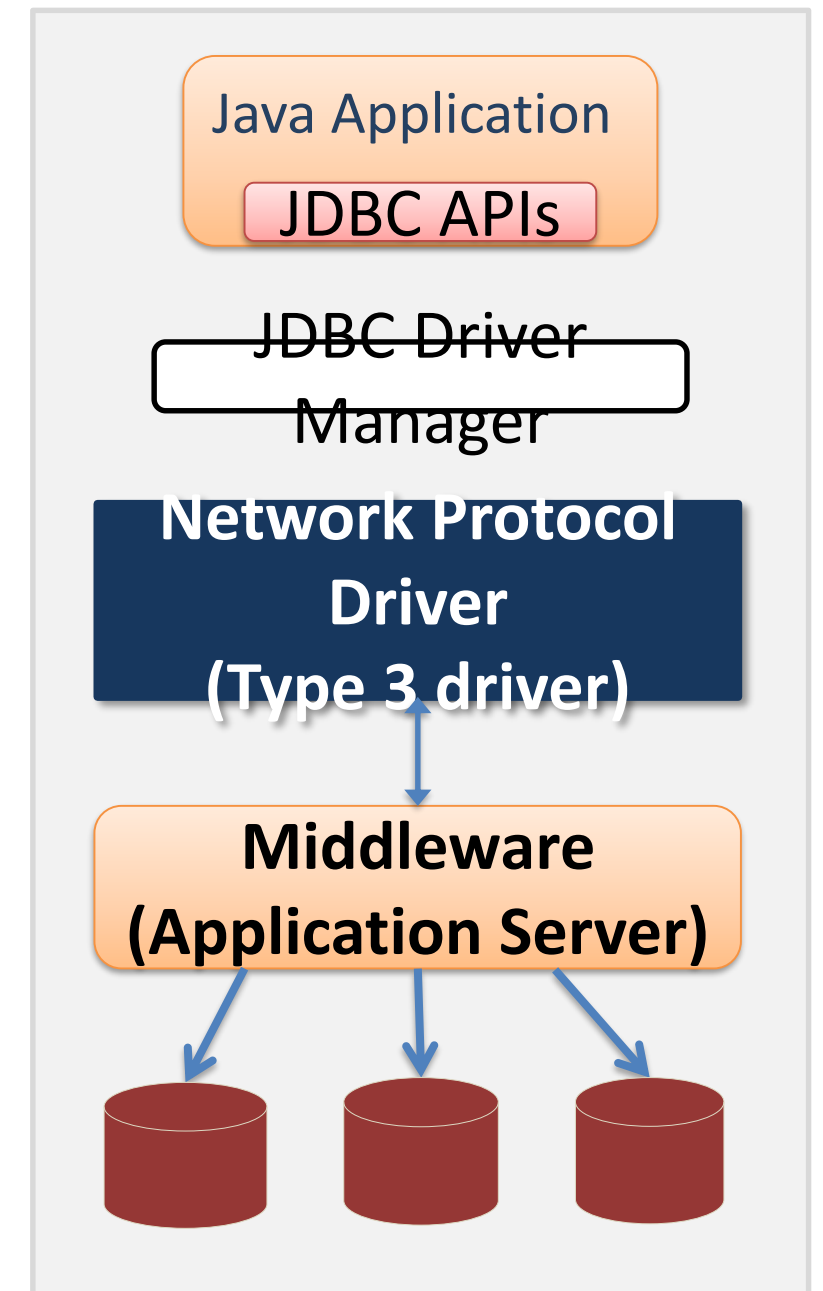


Type-3: Network Protocol Driver

- Follows a three tier communication approach
- JDBC calls are passed to the middle-tier server that translates the JDBC calls to vendor specific database calls
- Extremely Flexible as single driver can provide access to multiple databases
- Fully written in java

- Communication between client and the middleware server is database independent
- Can connect to multiple databases
- Portable across platforms

- Requires database-specific coding to be done in the middle tier

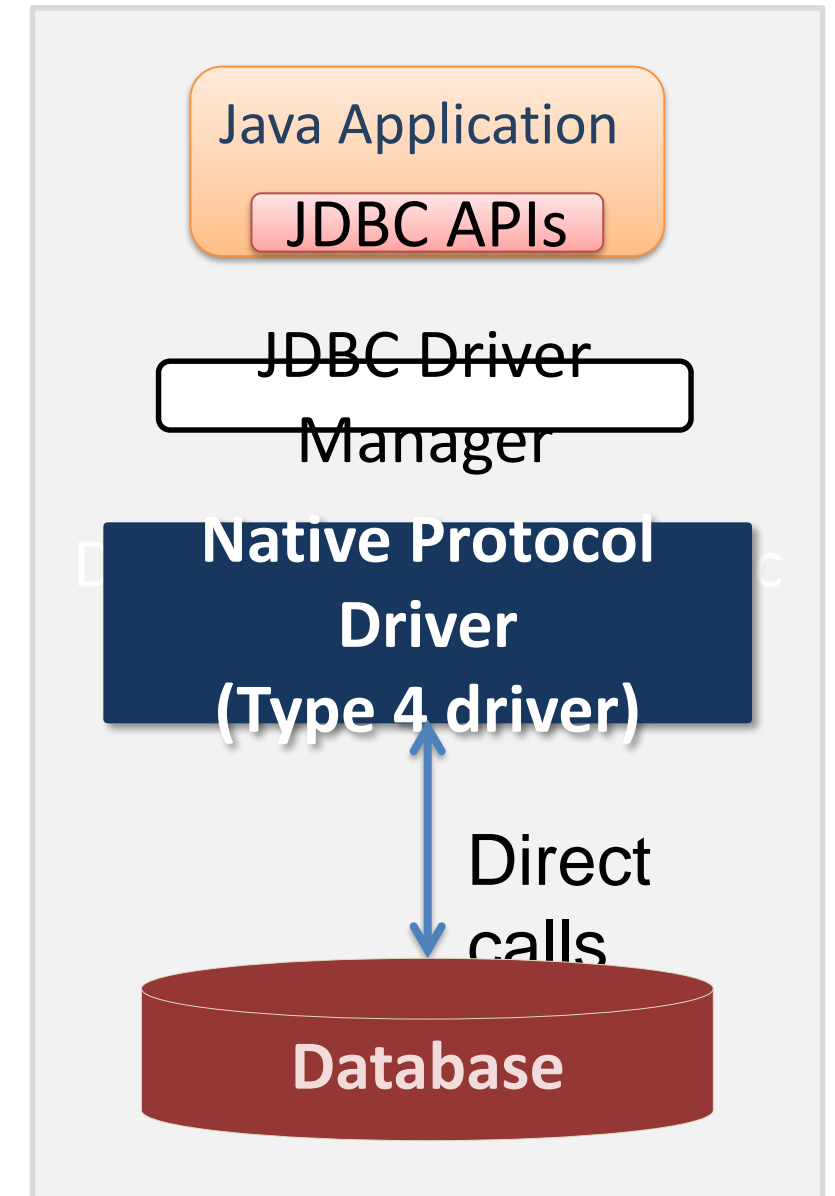


Type-4 : Native Protocol Driver

- Also known as the Direct to Database Pure Java Driver
- Uses java networking libraries to directly communicate with the database server
- Converts JDBC calls directly into a vendor-specific database protocol

- Highest performance driver available for the database
- Completely written in java to achieve platform independence

- Drivers are database dependent, hence separate driver is required for each database



Driver Selection

- **Type-4**
 - Preferred driver, When java application needs to access one type of database, such as Oracle, MySQL, etc
- **Type-3**
 - Preferred driver, If a Java application is accessing multiple types of databases
- **Type-2**
 - Are useful in situations where a type 3 or type 4 driver is not available yet
- **Type-1**
 - Not considered a deployment-level driver and is typically used for development and testing purposes only

Register JDBC Driver

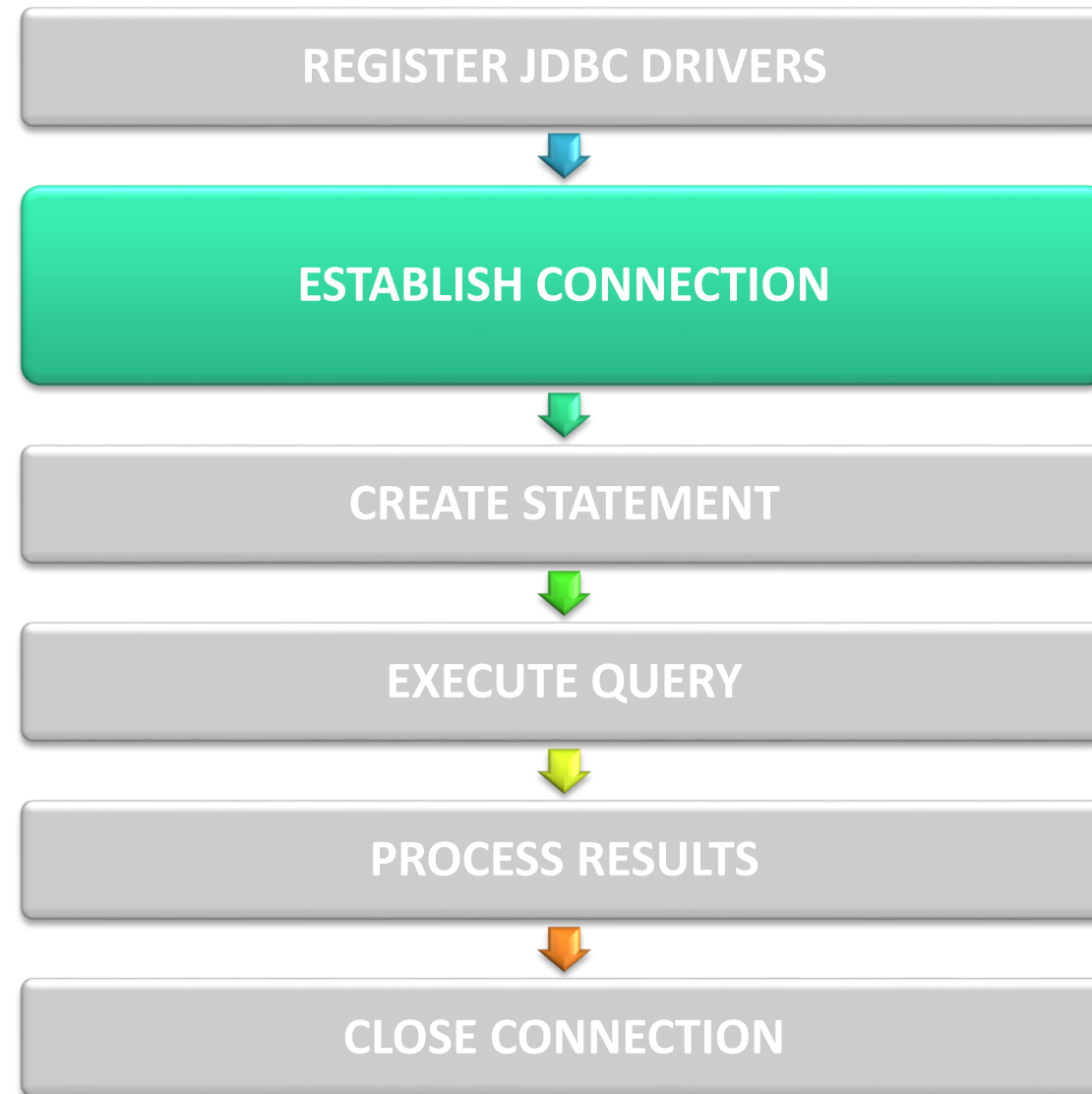
- `Class.forName("driver class name")` method loads the driver class in memory and registers it with the `DriverManager`
- When a driver is registered, it becomes available for making a connection with DBMS
- Example of Loading and registering a Oracle Driver

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

Oracle
Driver
class
Name

Note:

- `DriverManager.registerDriver(...)` can also be used to register a driver. Hardly used unless it is a custom driver
- Class name of the driver will be available in the documentation provided by database vendor



Establishing Connection with DBMS

- **DriverManager** class is used to establish a connection with the DBMS
- **DriverManager**
 - Maintains a list of drivers that are registered with it
 - Responsible for finding a driver corresponding to the database connection URL
 - Responsible for using appropriate driver to connect to the corresponding database
- **Below are the static methods of DriverManager used to open a connection**

```
getConnection( String url, String user, String password ) : Connection  
getConnection( String url ) : Connection
```
- **getConnection(...)** method requires a database connection URL, user id and password to connect to the DBMS

Format of Database Connection URL

- Database connection Url provides the necessary information needed by DriverManager to connect to the DBMS
- Connection Url is database specific i.e varies depending on the database

URL Syntax for Oracle

```
jdbc:oracle:<driver  
type>@<hostname>:<port>:<Oracle SID>
```

URL: `jdbc:oracle:thin@localhost:1521:XE`

URL with userid and password included

URL for MySQL

```
jdbc:mysql://localhost:3306/test
```

Opening a Connection using DriverManager

```
static final String DB_URL = "jdbc:oracle:thin:@localhost:1521:XE";
static final String USERID = "hr";
static final String PASSWORD = "hr";
try{
    Connection conn = DriverManager.getConnection(DB_URL, USERID, PASSWORD);
} catch (SQLException ex) {
    System.out.println("Error: Unable to open connection");
}
```

- **getConnection(...)** method returns a Object of type Connection
- The object returned is assigned to the Connection Interface

Connection Interface

- **Connection Interface provides methods for**
 - **Creating Statement Objects**
 - **Transaction Management**
 - **Getting meta data of Database**
 - **Ending the Connection**
- **Connection object represents a connection session with a database**
- **Java Application can have connections to multiple database or multiple connections to a single database**
- **Connection has to be closed once all the data access is complete for the session**



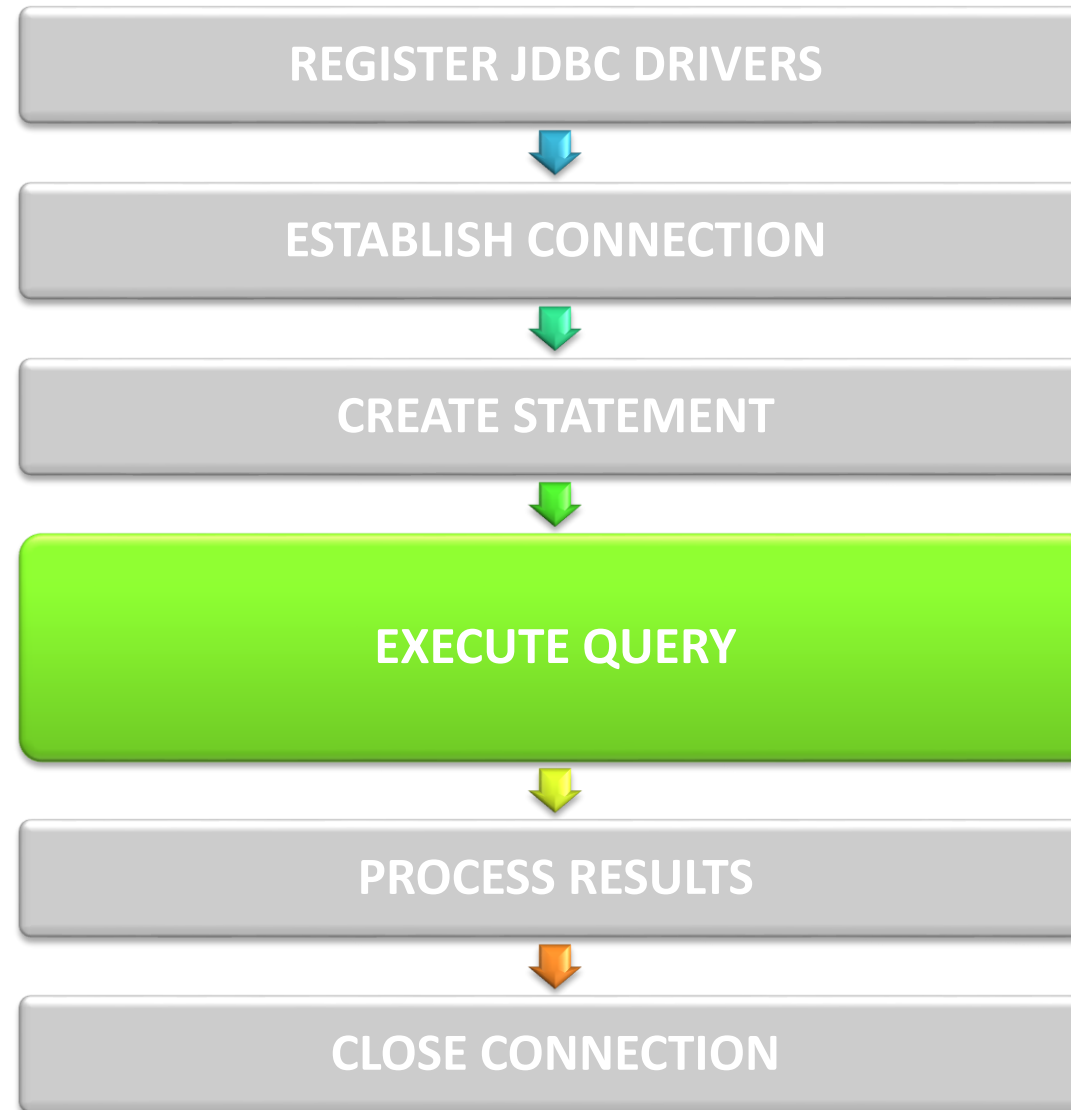
Create Statement

- **createStatement()** method of Connection Interface is used to create a Statement object
- **Statement** object is used to send SQL statements to the database.

```
try{
    Connection conn = DriverManager.getConnection(DB_URL, USERID, PASSWORD);

    Statement statement = conn.createStatement();

catch(SQLException ex) {
    System.out.println("Error: Unable to open connection");
}
```



Executing Query

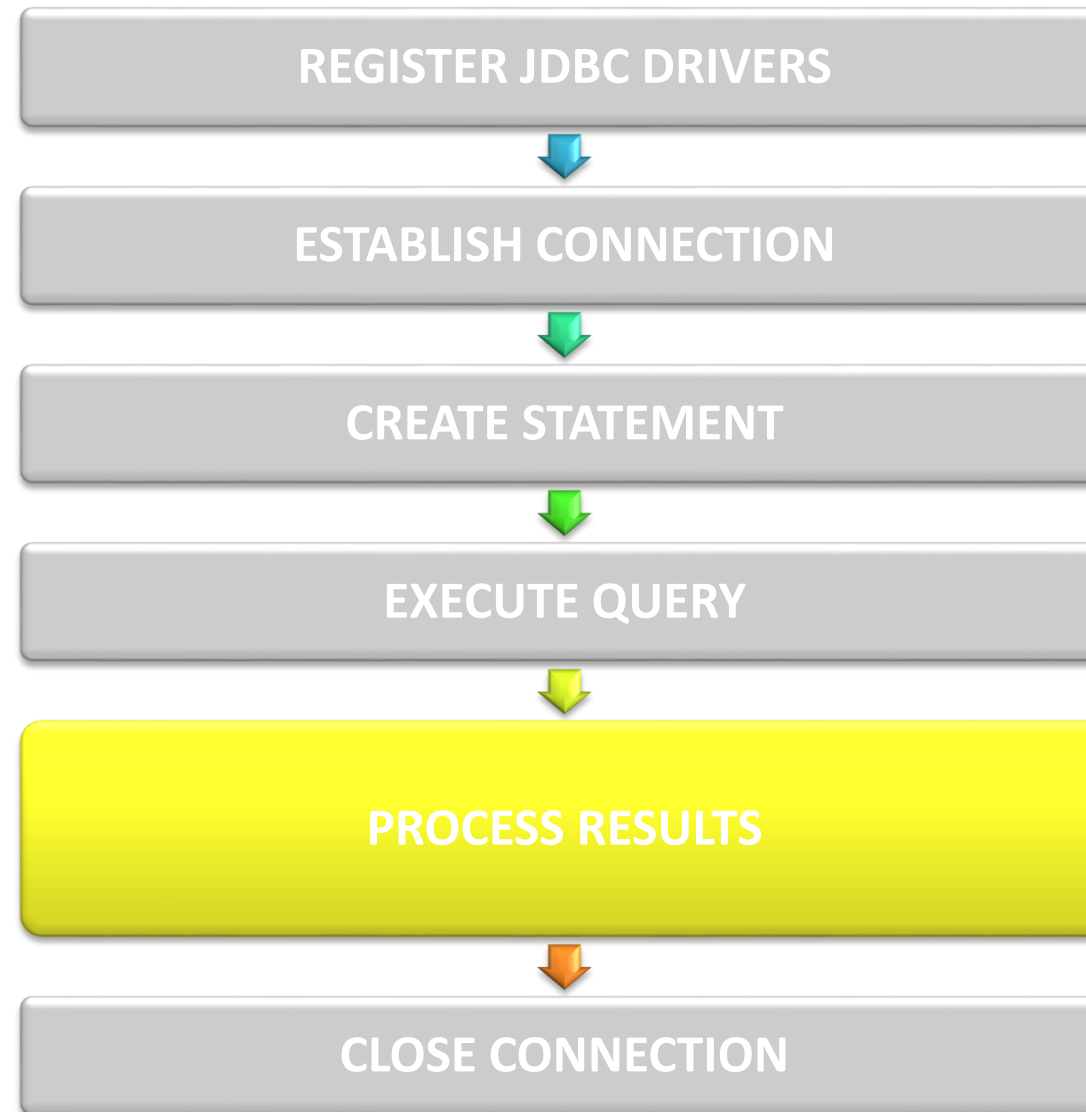
- Statement object is used to execute a static SQL query and return the results of the query
- **Statement Interface** provides methods for execution of static SQL queries
 - `executeQuery(sql)` method is used to execute a SQL statement (SELECT Query) which returns a **ResultSet**

```
String sql = "Select COUNTRY_ID, COUNTRY_NAME from COUNTRIES";
try{
    Connection conn = ...
    Statement statement = conn.createStatement();

    ResultSet result = statement.executeQuery(sql);

    catch(SQLException ex) {
        System.out.println("Error: Unable to open connection");
    }
```

- **ResultSet Object** contains the data returned ²³by the database in a table form



ResultSet

- The results of the SELECT query executed using Statement object is returned in a ResultSet object

```
Select COUNTRY_ID, COUNTRY_NAME from COUNTRIES  
where REGION_ID = 3;
```

- The above query when executed by the Statement Object will return the Resultset as shown

- ResultSet object maintains a cursor that points to the current row in the result set

- The initial position of the cursor for a new ResultSet is before the first row

COUNTRY_ID char(2)	COUNTRY_NAME varchar(40)	REGION_ID Number
AR	Argentina	2
BE	Belgium	1
IN	India	3
AU	Australia	3

COUNTRIES TABLE



IN	India
AU	Australia

**RESUL
TSET**

Process Results


- **ResultSet** Interface provides methods for navigating the **ResultSet**, retrieving and manipulating the data in the **ResultSet**
- For retrieval, the cursor has to be positioned on the row using **next()** method before retrieving the data

```
ResultSet result = statement.executeQuery(sql);  
result.next(); //returns true if row exists
```

- **ResultSet** interface provides **getXXX(..)** methods to retrieve the column values from the current row based on the column type

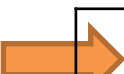
```
String ctryId = result.getString("COUNTRY_ID");  
String ctryName = result.getString("COUNTRY_NAME");
```

**O
R**



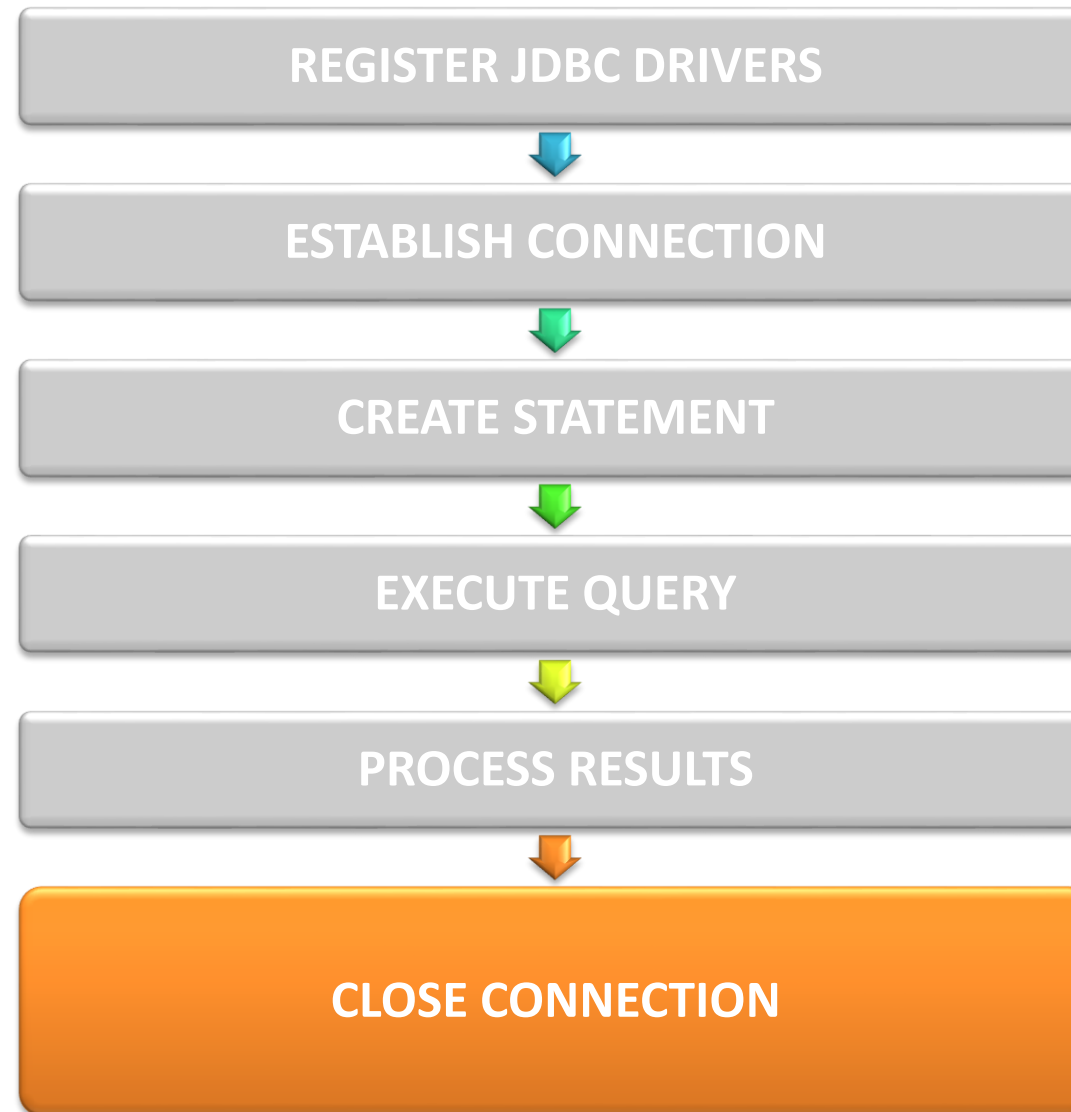
IN	India
AU	Australia

**RESUL
TSET**



IN	India
AU	Australia

```
result.getString(1);  
result.getString(2);
```



CLOSE CONNECTION

- After retrieving the data from the Resultset, the Statement object has to be closed to release the resources held
- `close()` method releases the database and JDBC resources immediately
- Closing a statement object automatically closes the ResultSet object that was generated by the Statement

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

- If the Connection to Database is no more needed, It should be closed as below to release JDBC and database resources

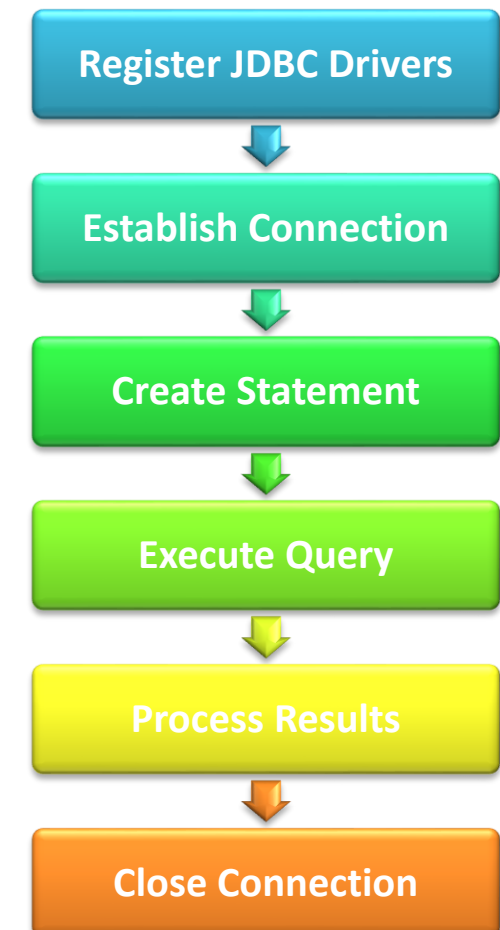
```
connection.close();
```

SQLException

- SQLException is thrown, If any errors occur while accessing the database using jdbc
- SQLException is defined in java.sql package
- Methods for getting details about the Exception
 - `getErrorCode()`
 - Returns an error code that is specific to each vendor. Normally this will be the actual error code returned by the underlying database.
 - `getMessage()`
 - Returns a string describing the error
 - `getSQLState()`
 - Returns a string with the SQLState of the database error
 - `getNextException()`
 - Gets the next Exception object in the exception chain

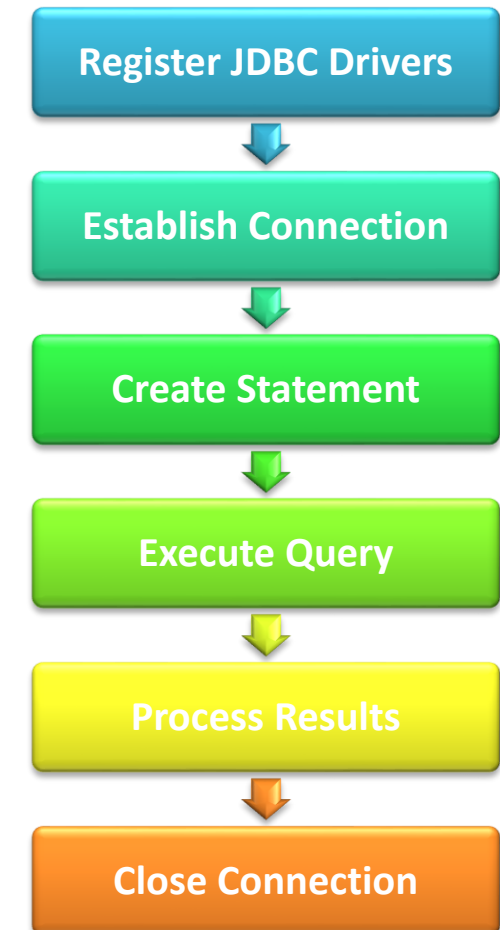
Putting it all together

```
public class JDBCdemo1 {  
    private static final String DB_URL =  
        "jdbc:oracle:thin:@localhost:1521:xe";  
    private static final String USER_ID = "hr";  
    private static final String PASSWORD = "hr";  
  
    public static void main(String[] args) {  
        final String sql =  
            "Select COUNTRY_ID, COUNTRY_NAME from COUNTRIES";  
        Connection conn = null;  
        try {  
            // Register Driver  
            Class.forName("oracle.jdbc.OracleDriver");  
  
            // Establish Connection with DBMS  
            conn = DriverManager.getConnection(DB_URL, USER_ID, PASSWORD);  
  
            // Create Statement  
            Statement statement = conn.createStatement();  
  
            // Execute Query  
            ResultSet result = statement.executeQuery(sql);  
        }  
    }  
}
```



Putting it all together

```
// Process Results
while (result.next()) {
    String id = result.getString("COUNTRY_ID");
    String name = result.getString("COUNTRY_NAME");
    System.out.println(id + " " + name);
}
} catch (SQLException e) {
    e.printStackTrace();
} catch (ClassNotFoundException e) {
    e.printStackTrace();
} finally {
    // Close Connection
    try {
        conn.close();
    } catch (SQLException e) {
        e.printStackTrace();
    }
}
}
```



DATA TYPE MAPPING

- The following tables shows the mapping between Java types and SQL Data types
- The JDBC driver does the data type conversion implicitly

SQL	JAVA
VARCHAR, CHAR, LONGVARCHAR	java.lang.String
TINYINT	byte
SMALLINT	short
INTEGER	int
BIGINT	long
REAL, FLOAT	float
DOUBLE	double

SQL	JAVA
DATE	java.sql.Date
TIME	java.sql.Time
TIMESTAMP	java.sql.Timestamp
ARRAY	java.sql.Array
BINARY	byte[]
VARBINARY	byte[]
NUMERIC	java.math.BigDecimal

SQLWarning

- **SQLWarning objects are a subclass of SQLException that deal with database access warnings.**
- **Warnings do not stop the execution of an application, as exceptions do; they simply alert the user that something did not happen as planned**
- **A warning can be reported on a Connection object, a Statement object or a ResultSet object**

```
➤ SQLWarning warning = stmt.getWarnings();  
if (warning != null) {  
    System.out.println("Warnings");  
    while (warning != null) {  
        System.out.println("Message: " + warning.getMessage());  
        System.out.println("SQLState: " + warning.getSQLState());  
        warning = warning.getNextWarning();  
    }  
}
```