UNIT-III

JDBC

- **JDBC Overview:**
    - JDBC, or Java Database Connectivity, is a Java-based API that enables Java applications to interact with relational databases.
    - The JDBC API consists of a set of interfaces and classes which enables java programs to execute SQL statements.
    - It provides standard methods for connect relational databases, executing SQL queries, retrieving results etc.

Features of JDBC:

1. **Database Independence**: JDBC abstracts the underlying database, allowing developers to write code that can work with different database management systems (DBMS) without needing significant changes.
2. **Standardized API**: It provides a consistent set of interfaces and classes, making it easier to develop database-driven applications.
3. **Support for SQL**: JDBC allows you to execute SQL queries, updates, and stored procedures, providing complete control over database operations.
4. **Connection Management**: It handles connections to databases efficiently, allowing for transaction management and error handling.

Components:

1. **JDBC Drivers**: These are the components that enable Java applications to communicate with a database. There are four types of JDBC drivers:
    1. **Type 1**: JDBC-ODBC Bridge Driver
    2. **Type 2**: Native-API Driver
    3. **Type 3**: Network Protocol Driver
    4. **Type 4**: Thin Driver
2. **Connection Management**: JDBC provides methods for establishing and managing connections to databases using the Driver Manager class.

Syntax:

```
Connection connection = DriverManager.getConnection(url, user, password)
```

3. **Statement Interfaces**: JDBC supports several types of statements for executing SQL commands.
    1. **Statement**: For executing simple SQL queries without parameters.
    2. **PreparedStatement**: For executing precompiled SQL queries, which can include parameters.
    3. **CallableStatement**: For executing stored procedures.
4. **ResultSet**: This interface represents the result set of a query. It provides methods to navigate through the data and retrieve values.
5. **Transaction Management**: JDBC allows applications to manage transactions, including commit and rollback operations, ensuring data integrity.

Basic Workflow

1. **Load the Driver**: Use Class.forName() to load the JDBC driver.
2. **Establish a Connection**: Use DriverManager.getConnection() to connect to the database.
3. **Create a Statement**: Use the connection to create a Statement, PreparedStatement, or CallableStatement.
4. **Execute Queries**: Use the statement to execute SQL queries and retrieve results.
5. **Process Results**: Iterate through the ResultSet to process the results of the query.
6. **Close Connections**: Close the ResultSet, Statement, and Connection objects to free up resources.

Example:

```
//Java program to implement a simple JDBC application
import java.sql.*;
public class JDBCDemo {
public static void main(String args[]){
String driverClassName= "sun.jdbc.odbc.JdbcOdbcDriver";
String url = "jdbc:odbc:XE";
String username = "scott";
String password = "tiger";
String query= "insert into students values(109, 'bhatt')";
// Load driver class
Class.forName(driverClassName);
```

```
// Establish a connection
Connection con = DriverManager.getConnection(url, username, password);
// Create a statement
Statement st = con.createStatement();
// Execute the query
int count = st.executeUpdate(query);
System.out.println("number of rows affected by this query= "+ count);
// Closing the connection as per the requirement with connection is completed
con.close();
}
} // class
```

Advantages:

- **Database Independence**: JDBC provides a uniform interface for different databases.
- **Support for Object-Oriented Programming**: Being part of Java, it integrates well with Java's object-oriented features.
- **Robust Error Handling**: Exception handling is built-in, allowing developers to manage database errors effectively.

❖ **Steps for JDBC Implementation:**

1. **Load the JDBC Driver**: You can load the driver using Class.forName() method. However, this step is often not necessary with newer JDBC versions that support service provider loading.
2. **Establish a Connection**: Use DriverManager.getConnection() to establish a connection to your database.
3. **Create a Statement**:
   Use Connection.createStatement() or Connection.prepareStatement() to create a SQL statement.
4. **Execute the Query**: Use Statement.executeQuery() for SELECT statements or Statement.executeUpdate() for INSERT, UPDATE, DELETE.
5. **Process the Result Set**: If you executed a query, you will receive a Result Set which you can iterate through to get the results.

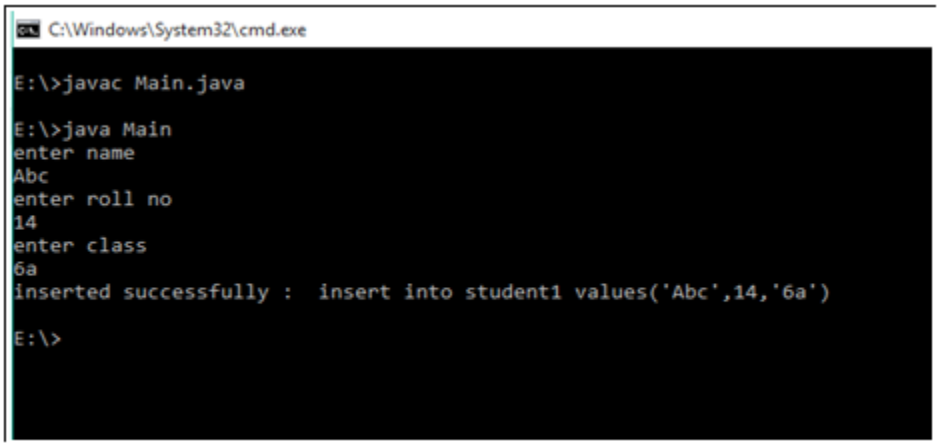
6. **Close the Connections**: Always close your ResultSet, Statement, and Connection to free up resources.

Example :

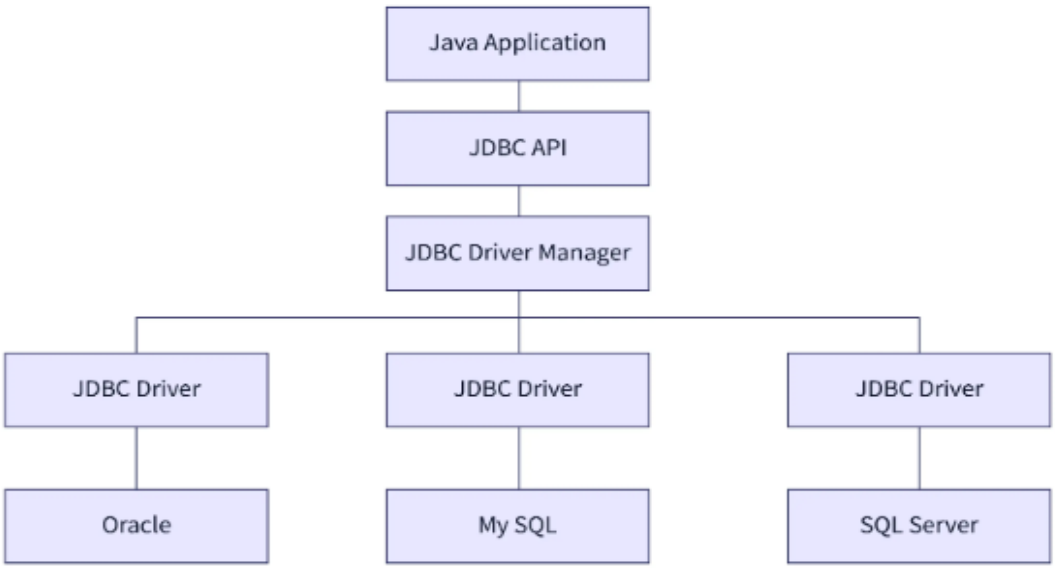
```
import java.sql.*;
import java.util.*;
class Main {
public static void main(String a[]){
String url = "jdbc:oracle:thin:@localhost:1521:xe";
String user = "system";
String pass = "12345";
Scanner k = new Scanner(System.in);
System.out.println("enter name");
String name = k.next();
System.out.println("enter roll no");
int roll = k.nextInt();
System.out.println("enter class");
String cls = k.next();
String sql = "insert into student1 values('" + name+ "'," + roll + ",'" + cls + "')";
Connection con = null;
try {
DriverManager.registerDriver(new oracle.jdbc.OracleDriver());
con = DriverManager.getConnection(url, user, pass);
Statement st = con.createStatement();
int m = st.executeUpdate(sql);
if (m == 1)
System.out.println("inserted successfully : " + sql);
else
System.out.println("insertion failed");
con.close();
}catch (Exception ex) {
System.err.println(ex);
```

}}}

Output:



Architecture of JDBC:



As we can see in the above image the major components of JDBC architecture are as follows:

1. Application
2. The JDBC API
3. Driver Manager
4. JDBC Drivers

5. Data Sources

1. Application:

Applications in JDBC architecture are java applications like applets or servlet that communicates with databases.

2. JDBC API:

The JDBC API is an Application Programming Interface used to create Databases. JDBC API uses classes and interfaces to connect with databases

3. Driver Manager:

Driver Manager class in the JDBC architecture is used to establish a connection between Java applications and databases. Using the get Connection method of this class a connection is established between the Java application and data sources.

4. JDBC Drivers:

JDBC drivers are used to connecting with data sources. All databases like Oracle, MYSQL, MYSQL, etc. Class is a java class used to load drivers. **Class.forName()** method is used to load drivers in JDBC architecture.

5. Data Sources:

Data Sources in the JDBC architecture are the databases that we can connect using this API. These are the sources where data is stored and used by Java applications. JDBC API helps to connect various databases like Oracle, MYSQL, MSSQL, PostgreSQL, etc.

Types of JDBC Architecture

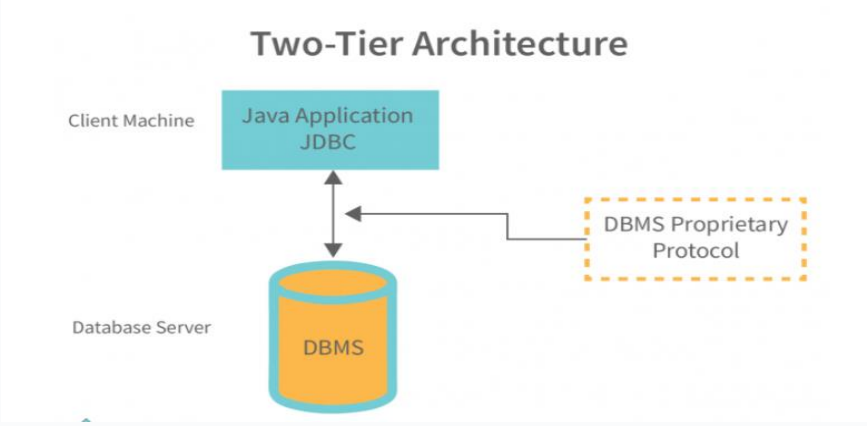
The JDBC Architecture can be of two types based on the processing models it uses. These models are

1. 2-tier model
2. 3-tier model

1.Two-tier model:

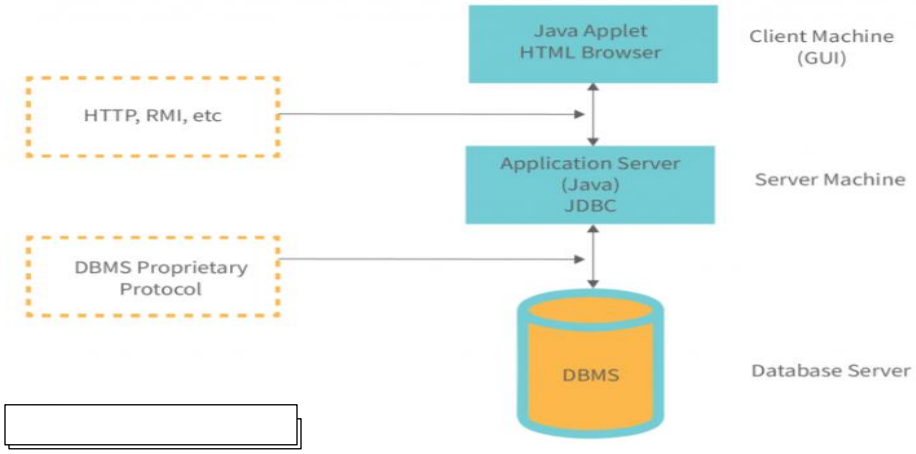
- Two –tier model JDBC architecture model is a **basic model or simple model**.
- In this model, the application interacts directly with the source of data. The JDBC driver establishes the interaction between the data source and the application.
- When a query is sent by the user to the data source, the reply of those sent queries is sent directly to the user.

- In this model, the data source can be located on a different machine connected to the same network the user is connected to.
- This model is also known as a **client/server** configuration. Here user's machine acts as a client and the machine on which the database is located acts as a server.



Three-tier model:

- 3-tier model is a **complex and more secure model** of JDBC architecture in java.
- In this model the user queries are sent to the middle tier and then they are executed on the data source.
- Here, the java application is considered as one tier connected to the data source (3rd tier) using middle-tier services.
- In this model user queries are sent to the data source using middle-tier services, from where the commands are again sent to databases for execution.
- The results obtained on the database are again sent to the middle-tier and then to the user/application.



❖ **JDBC Implementation:**

Java Database Connectivity (JDBC) is a Java API that allows Java applications to interact with relational databases. It provides a standard interface for connecting to databases and executing SQL queries.

Steps for JDBC Implementation:

1. **Import JDBC Packages:** First, you need to import the necessary JDBC classes.

```
import java. Sql.*;
```

2. **Load the JDBC Driver:** This step is required to establish a connection with the database. For modern versions of JDBC, you don't always need to manually load the driver, but in some cases, you may still need to do it.

Example:

```
try {
   Class.forName("com.mysql.cj.jdbc.Driver");
} catch (ClassNotFoundException e) {
   e.printStackTrace();
}
```

3. Establish Connection: Use the Driver Manager class to establish a connection to the database.

```
String url = "jdbc:mysql://localhost:3306/yourdatabase";
String username = "yourusername";
String password = "yourpassword";
Connection connection = null;
```

```
try {
    connection = DriverManager.getConnection(url, username, password);
    System.out.println("Connection successful.");
} catch (SQLException e) {
    e.printStackTrace();
}
```

4. **Create a Statement:** Once a connection is established, you can create a Statement object to execute SQL queries.

```
Statement statement = null;
try {
    statement = connection.createStatement();
} catch (SQLException e) {
    e.printStackTrace();
}
```

5. **Execute Queries:** You can execute different types of SQL queries, such as SELECT, INSERT, UPDATE, and DELETE. For example:

- **Execute a SELECT Query (Retrieve Data):**

```
String sql = "SELECT * FROM employees";
ResultSet resultSet = null;
try {
    resultSet = statement.executeQuery(sql);
    while (resultSet.next()) {
        int id = resultSet.getInt("id");
        String name = resultSet.getString("name");
        System.out.println(id + ": " + name);
    }
} catch (SQLException e) {
    e.printStackTrace();
}
```

- **Execute an INSERT Query (Insert Data):**

```
String sql = "INSERT INTO employees (name, age) VALUES ('John Doe', 30)";
```

```
try {
    int rowsAffected = statement.executeUpdate(sql);
    System.out.println("Rows affected: " + rowsAffected);
} catch (SQLException e) {
    e.printStackTrace();
}
```

6. **Close the Connection:** Always close the Connection, Statement, and Result Set objects to avoid memory leaks and database connection issues.

```
try {
    if (resultSet != null) {
        resultSet.close();
    }
    if (statement != null) {
        statement.close();
    }
    if (connection != null) {
        connection.close();
    }
} catch (SQLException e) {
    e.printStackTrace();
}
```

Example of JDBC Implementation:

Here's a simple example that connects to a MySQL database and performs a SELECT query:

```
import java.sql.*;
public class JDBCExample {
public static void main(String[] args) {
// JDBC variables
String url = "jdbc:mysql://localhost:3306/yourdatabase";
String username = "your username";
String password = "your password";
// Initialize connection objects
```

```
Connection connection = null;
Statement statement = null;
ResultSet resultSet = null;
try {
    // Load the JDBC driver
    Class.forName("com.mysql.cj.jdbc.Driver");
    // Establish a connection
    connection = DriverManager.getConnection(url, username, password);
    System.out.println("Connection successful.");
    // Create a statement
    statement = connection.createStatement();
    // Execute a SELECT query
    String sql = "SELECT * FROM employees";
    resultSet = statement.executeQuery(sql);
    // Process the result set
    while (resultSet.next()) {
        int id = resultSet.getInt("id");
        String name = resultSet.getString("name");
        System.out.println(id + ": " + name);
    }
} catch (SQLException | ClassNotFoundException e) {
    e.printStackTrace();
} finally {
    // Close resources
    try {
        if (resultSet != null) resultSet.close();
        if (statement != null) statement.close();
        if (connection != null) connection.close();
    } catch (SQLException e) {
        e.printStackTrace();
    }    }    }   }
```

❖ **Connection class:**

In JDBC, the Connection class is part of the java.sql package, and it provides methods to establish a connection with a database. Once a connection is established, you can create Statement, Prepared Statement, and Callable Statement objects to execute SQL queries. It also manages transaction control (commit and rollback) and connection management (close, set auto-commit mode, etc.).

Key Methods of the Connection Interface:

Here are some important methods provided by the Connection class:

1. **Creating a Connection:**

- **Connection get Connection(String url, String user, String password)**

  Establishes a connection to the database using the given URL, username, and password.

- **Connection get Connection(String url)**

  Establishes a connection using just the URL. This method is typically used for single-user or embedded databases.

2. **Creating Statements:**

1. **Statement create Statement ():** Creates a Statement object for executing SQL queries without parameters.
2. **Prepared Statement prepare Statement (String sql)**: Creates a Prepared Statement object for executing SQL queries with parameters.
3. **Callable Statement prepare Call (String sql):** Creates a Callable Statement for executing stored procedures.

3. **Managing Transactions:**

1. **void setAutoCommit(Boolean auto Commit)**

   Enables or disables auto-commit mode. In auto-commit mode, each statement is executed as a transaction. Disabling auto-commit allows you to manage transactions manually.

2. **Boolean getAutoCommit()**

   It returns whether auto-commit mode is enabled or not.

3. **void commit()**

   It commits the current transaction.

4. **void rollback()**

   It rolls back the current transaction.

4. **Managing Database Metadata:**

Database Metadata get Metadata()

It returns a Database Metadata object that can be used to get information about the database (such as supported features, tables, columns, etc.).

5.**Managing Connections and Resources:**

- **void close()**

  Closes the Connection object and releases any database resources associated with it.

- **Boolean is Closed()**

  Returns true if the connection is closed, false otherwise.

6. **Handling SQL Exceptions:**

- **SQLWarning get Warnings()**

  Returns any warnings generated by the database during the execution of SQL commands.

7. **Set and Retrieve Client Info:**

- **void setClientInfo(String name, String value)**

  Sets a custom client info property.

- **String getClientInfo(String name)**

  Retrieves the value of a custom client info property.

Example:

```
import java. Sql.*;
public class Connection Example {
   public static void main(String[] args) {
      String url = "jdbc:mysql://localhost:3306/your database";
      String username = "your username";
      String password = "your password";
      Connection connection = null;
      try {
         Class.forName("com.mysql.cj.jdbc.Driver");
         connection = DriverManager.getConnection(url, username, password);
         System.out.println("Connection established successfully.");
         connection. SetAutoCommit(false);
         Statement stmt = connection.createStatement();
```

```
            String sql = "SELECT * FROM employees";
            ResultSet resultSet = stmt.executeQuery(sql);
             while (resultSet. Next()) {
                int id = resultSet.getInt("id");
                String name = resultSet.getString("name");
                System.out.println("ID: " + id + ", Name: " + name);
            }
            connection.commit();
        } catch (SQLException | ClassNotFoundException e) {
            e.printStackTrace();
            try {
                if (connection != null) {
                    connection. Rollback();
                }
            } catch (SQLException rollbackEx) {
                rollbackEx.printStackTrace();
            }
        } finally {
                    try {
                if (connection != null && !connection.isClosed()) {
                    connection. Close();
                    System.out.println("Connection closed.");
                }
            } catch (SQLException e) {
                e.printStackTrace();
            } } } }
```

- **Statements:**

The **Statement** interface in JDBC is used to create SQL statements in Java and execute queries with the database. There are different types of statements used in JDBC:

1. Create Statement
2. Prepared Statement

3. Callable Statement

1. Create a Statement:

A Statement object is used for general-purpose access to databases and is useful for executing static SQL statements at runtime.

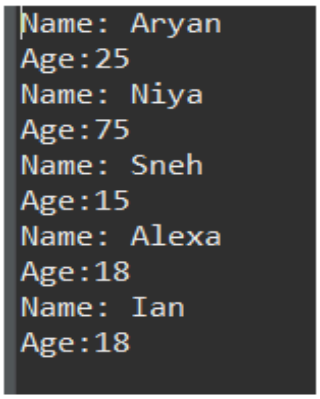
Syntax:

Statement statement = connection.createStatement();

Example:

```
import java.sql.*;
class GFG {
   public static void main(String[] args){
      try {
         Class.forName("com.mysql.cj.jdbc.Driver");
         Connection con = DriverManager.getConnection("jdbc:mysql:///world", "root",
"12345");
         Statement statement = con.createStatement();
         String sql = "select * from people";
         ResultSet result = statement.executeQuery(sql);
         while (result. Next()) {
            System.out.println("Name: " + result.getString("name"));
            System.out.println ( "Age:" + result.getString("age"));
         }
      }
      catch (SQLException e) {
         System.out.println(e);
      }
      catch (ClassNotFoundException e) {
         e.printStackTrace();
} } }
```

Output:



2. Prepared Statement:

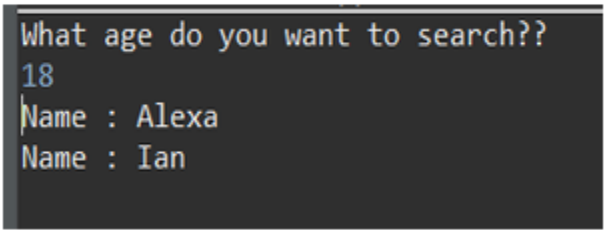
A Prepared Statement represents a precompiled SQL statement that can be executed multiple times. It accepts parameterized SQL queries, with ? as placeholders for parameters, which can be set dynamically.

Example:

```
import java.sql.*;
import java.util.Scanner;
class GFG {
   public static void main(String[] args){
      try {
         Class.forName("com.mysql.cj.jdbc.Driver");
         Scanner sc = new Scanner(System.in);
         System.out.printl ("What age do you want to search?? ");
         int age = sc.nextInt();
         Connection con = DriverManager.getConnection("jdbc:mysql:///world", "root",
"12345");
         Preparedstatement ps = con.prepareStatement ("select name from world. People
where age = ?");
         ps.setInt(1, age);
         ResultSet result = ps.executeQuery();
         while (result. Next()) {
            System.out.println("Name : + result.getString(1));
```

```
        }
    }
    catch (SQLException e) {
        System.out.println(e);
    }
    catch (ClassNotFoundException e) {
        e.printStackTrace();
    } } }
```

Output:



3. Callable Statement:

A Callable Statement is used to execute stored procedures in the database. Stored procedures are precompiled SQL statements that can be called with parameters. They are useful for executing complex operations that involve multiple SQL statements.

Example:

```
import java.sql.*;
public class GFG {
    public static void main(String[] args) {
        try {
            Class.forName("com.mysql.cj.jdbc.Driver");
            Connection con = DriverManager.getConnection("jdbc:mysql:///world", "root",
"12345");
            CallableStatement cs = con.prepareCall("{call GetPeopleInfo()}");
            ResultSet result = cs.executeQuery();
            while (result. Next()) {
                System.out.println("Name : " + result.getString("name"));
```

```
            System.out.println("Age : " + result.getInt("age"));
        }
        result.close();
        cs.close();
        con.close();
    }
    catch (SQLException e) {
        e.printStackTrace();
    }
            catch (ClassNotFoundException e) {
        e.printStackTrace();
    } } }
```

Output:

```
Name : Aryan
Age : 25
Name : Niya
Age : 75
Name : Sneh
Age : 15
Name : Alexa
Age : 18
Name : Ian
Age : 18
Name : Bob
Age : 64
```

❖ **Catching Database Results:**

- Java Database Connectivity is Java-based technology and that provides a standard API for accessing databases in Java applications.
- The Key Component of Java Database Connectivity is the Result Set. JDBC driver allows developers to read and manipulate data from the database.
- In JDBC catching database results typically refers to retrieving and processing the results of a query.
- The process of executing queries and handling the results involves several steps, including creating a connection, preparing a statement, executing the query, and then extracting and processing the results. Here's how you can handle the results from a database in JDBC.

1. Establish a Connection

First, you need to create a connection to your database using the Connection object. Typically, you'll use DriverManager.getConnection or DataSource.getConnection to establish a connection.

2. **Execute the Query**

You can execute a query using a Statement or Prepared Statement. After the query is executed, a Result Set object is returned, which contains the data returned from the database.

3. Processing the Result Set

The Result Set contains the rows returned by the query. You can iterate through this Result Set and retrieve the data column by column.

Example:

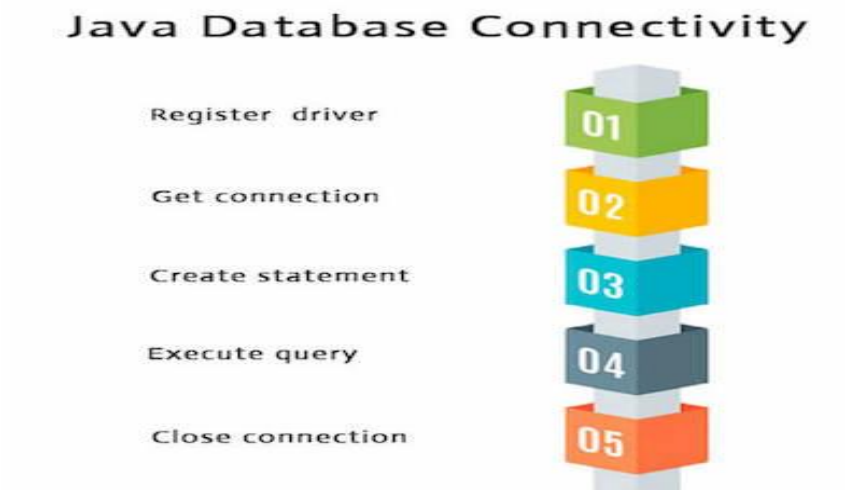
```
import java.sql. *;
public class JDBCExample {
public static void main(String[] args) {
   String url = "jdbc:mysql://localhost:3306/your database";
   String username = "your username";
   String password = "your password";
   String query = "SELECT id, name, email FROM users";
   Connection connection = null;
   Statement statement = null;
   ResultSet resultSet = null;
   try {
      connection = DriverManager.getConnection(url, username, password);
      statement = connection.createStatement();
      resultSet = statement.executeQuery(query);
      while (resultSet.next()) {
         int id = resultSet.getInt("id");
         String name = resultSet.getString("name");
         String email = resultSet.getString("email");
         System.out.println("ID: " + id + ", Name: " + name + ", Email: " + email);
```

```
    }
} catch (SQLException e) {
    e.printStackTrace();
} finally {
    try {
        if (resultSet != null) resultSet.close();

        if (statement != null) statement.close();

        if (connection != null) connection. Close();

    } catch (SQLException e) {

        e.printStackTrace();
    }   }   }  }
```

❖ **Handling Database Queries:**

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

1. Register the Driver class
2. Create connection
3. Create statement
4. Execute queries
5. Close connection



1) Register the driver class

The for Name() method of Class class is used to register the driver class. This method is used to dynamically load the driver class.

Syntax :

public static void forName(String className)throws ClassNotFoundException

Example:

Class.forName("oracle.jdbc.driver.OracleDriver");

2) Create the connection object:

The get Connection() method of Driver Manager class is used to establish connection with the database.

Syntax :

1) public static Connection get Connection(String url)throws SQLException

2) public static Connection get Connection(String url, String name, String password) throws SQLException

Example:

Connectioncon=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe","system","password");

3) Create the Statement object:

The create Statement() method of Connection interface is used to create statement. The object of statement is responsible to execute queries with the database.

Syntax:

public Statement create Statement()throws SQLException

Example:

Statement stmt=con.createStatement();

4) Execute the query:

The execute Query() method of Statement interface is used to execute queries to the database. This method returns the object of ResultSet that can be used to get all the records of a table.

Syntax:

public ResultSet executeQuery(String sql)throws SQLException

Example:

```
ResultSet rs=stmt.executeQuery("select * from emp");
while(rs.next()){
System.out.println(rs.getInt(1)+" "+rs.getString(2));
}
```

5) Close the connection object:

By closing connection object statement and ResultSet will be closed automatically. The close() method of Connection interface is used to close the connection.

Syntax:

```
public void close()throws SQLException
```

Example:

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

- **Inet Address class:**

Java Inet Address class represents an IP address. The java.net.InetAddress class provides methods to get the IP of any host name.

Example:www.javatpoint.com, www.google.com, www.facebook.com, etc.

- An IP address is represented by 32-bit or 128-bit unsigned number. An instance of Inet Address represents the IP address with its corresponding host name.
- There are two types of addresses: Unicast and Multicast.
- The Unicast is an identifier for a single interface whereas Multicast is an identifier for a set of interfaces.

IP Address:

- An IP address helps to identify a specific resource on the network using a numerical representation.
- Most networks combine IP with TCP (Transmission Control Protocol). It builds a virtual bridge among the destination and the source.

There are two versions of IP address:

1. IPv4:

- IPv4 is the primary Internet protocol. It is the first version of IP deployed for production in the ARAPNET in 1983.

- It is a widely used IP version to differentiate devices on network using an addressing scheme.
- A 32-bit addressing scheme is used to store 2^{32} addresses that is more than 4 million addresses.

Features of IPv4:

- It is a connectionless protocol.
- It utilizes less memory and the addresses can be remembered easily with the class based addressing scheme.
- It also offers video conferencing and libraries.

2. IPv6:

- IPv6 is the latest version of Internet protocol. It aims at fulfilling the need of more internet addresses.
- It provides solutions for the problems present in IPv4. It provides 128-bit address space that can be used to form a network of 340 undecillion unique IP addresses.
- IPv6 is also identified with a name IPng (Internet Protocol next generation).

Features of IPv6:

- It has a stateful and stateless both configurations.
- It provides support for quality of service (QoS).
- It has a hierarchical addressing and routing infrastructure.
- TCP/IP is a communication protocol model used connect devices over a network via internet.
- TCP/IP helps in the process of addressing, transmitting, routing and receiving the data packets over the internet.
- The two main protocols used in this communication model are:
    - TCP i.e. Transmission Control Protocol. TCP provides the way to create a communication channel across the network. It also helps in transmission of packets at sender end as well as receiver end.
    - IP i.e. Internet Protocol. IP provides the address to the nodes connected on the internet. It uses a gateway computer to check whether the IP address is correct and the message is forwarded correctly or not.

Java Inet Address Class Methods:

Method	Description
public static InetAddress get Byname(String host) throws UnknownHostException	It returns the instance of InetAddress containing Localhost IP and name.
public static InetAddress get Localhost() throws UnknownHostException	It returns the instance of InetAddress containing local host name and address.
public String get Hostname()	It returns the host name of the IP address.
public String getHostAddress()	It returns the IP address in string format.

Example:

```
import java.io.*;
import java.net.*;
public class InetDemo{
public static void main(String[] args){
try{
InetAddress ip=InetAddress.getByName("www.javatpoint.com");
System.out.println("Host Name: "+ip.getHostName());
System.out.println("IP Address: "+ip.getHostAddress());
}catch(Exception e){System.out.println(e);}
}  }
```

Output:

```
Host Name: www.javatpoint.com
IP Address: 172.67.196.82
```

❖ **URL Class:**

The Java URL class represents an URL. URL is an acronym for Uniform Resource Locator. It points to a resource on the World Wide Web.

Example: https://www.javatpoint.com/java-tutorial

A URL contains many information:

1. **Protocol:** In this case, http is the protocol.
2. **Server name or IP Address:** In this case, www.javatpoint.com is the server name.
3. **Port Number:**

   It is an optional attribute. If we write http//ww.javatpoint.com:80/sonoojaiswal/ , 80 is the port number. If port number is not mentioned in the URL, it returns -1.
4. **File Name or directory name:**

   In this case, index.jsp is the file name.

Constructors of Java URL class:

URL(String spec):

Creates an instance of a URL from the String representation.

URL(String protocol, String host, int port, String file)

Creates an instance of a URL from the given protocol, host, port number, and file.

URL(String protocol, String host, int port, String file, URLStreamHandler handler)

Creates an instance of a URL from the given protocol, host, port number, file, and handler.

URL(String protocol, String host, String file)

Creates an instance of a URL from the given protocol name, host name, and file name.

URL(URL context, String spec)

Creates an instance of a URL by parsing the given spec within a specified context.

URL(URL context, String spec, URLStreamHandler handler)

Creates an instance of a URL by parsing the given spec with the specified handler within a given context.

Commonly used methods of Java URL class:

The java.net.URL class provides many methods. The important methods of URL class are given below.

Method	Description
public String getProtocol()	it returns the protocol of the URL.
public String getHost()	it returns the host name of the URL.
public String getPort()	it returns the Port Number of the URL.
public String getFile()	it returns the file name of the URL.
public String getAuthority()	it returns the authority of the URL.
public String toString()	it returns the string representation of the URL.
public String getQuery()	it returns the query string of the URL.
public String getDefaultPort()	it returns the default port of the URL.
public URLConnection openConnection()	it returns the instance of URLConnection i.e. associated with this URL.
public boolean equals(Object obj)	it compares the URL with the given object.
public Object getContent()	it returns the content of the URL.
public String getRef()	it returns the anchor or reference of the URL.
public URI toURI()	it returns a URI of the URL.

Example :

```
import java.net.*;
public class URLDemo{
public static void main(String[] args){
try{
URL url=new URL("http://www.javatpoint.com/java-tutorial");
System.out.println("Protocol: "+url.getProtocol());
System.out.println("Host Name: "+url.getHost());
System.out.println("Port Number: "+url.getPort());
System.out.println("File Name: "+url.getFile());
}
catch(Exception e){
System.out.println(e);
} }  }
```

Output:

Protocol: http

Host Name: www.javatpoint.com

- **TCP Sockets:**

TCP is the foundation of most web protocols, including HTTP, HTTPS, FTP, and more. When you access a website in a browser, your browser uses the HTTP/HTTPS protocol over a TCP connection to communicate with the web server.

- **HTTP/HTTPS**: These protocols are built on top of TCP to provide reliable, ordered communication between the client (e.g., your browser) and the server.
- **Web Sockets**: Web Sockets provide a full-duplex communication channel over a single TCP connection, often used for real-time applications (e.g., chat apps, live notifications).

How TCP works in Web Programming:

1. **Client-Server Model**: The client (e.g., a browser) connects to the server using a TCP connection. The client sends a request, and the server sends a response.
2. **Three-Way Handshake**: TCP establishes a connection via a handshake process. This ensures both parties are ready to communicate.

3. **Data Transmission**: After the connection is established, data is transmitted reliably, ensuring it arrives in order and is error-free.

Example of TCP Socket Server (Web Server)

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
public class JDBCExample {
public static void main(String[] args) {
String url = "jdbc:mysql://localhost:3306/mydatabase";
String user = "root";
 String password = "password";
 Connection conn = null;
 try {
          conn = DriverManager.getConnection(url, user, password);
          System.out.println("Connected to the database successfully!");
       } catch (SQLException e) {
          e.printStackTrace();
       }
      finally {
          if (conn != null) {
             try {
                conn.close();
             } catch (SQLException e) {
                e.printStackTrace();
          } } }
         } }
```

UDP Sockets :

UDP is less common in traditional web programming but is used in certain specific cases where real-time communication is essential, and slight data loss or packet reordering is acceptable. It is ideal for applications like live video streaming, VoIP (Voice over IP), and online gaming, where the timeliness of data transmission is more critical than guaranteed delivery.

- **DNS (Domain Name System)**: UDP is used by DNS servers to handle queries because DNS requires fast responses and can tolerate occasional packet loss.
- **Media Streaming**: UDP is often used for real-time video and audio streaming protocols (e.g., RTP - Real-Time Protocol) since it minimizes latency.
- **VoIP**: In voice communication, low latency is crucial, and slight packet loss (which may result in minor sound glitches) is less of a concern.

Example of UDP Socket :

```
import socket
server_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
server_socket.bind(('localhost', 8080))
print("UDP server listening on port 8080...")
while True:
    message, client_address = server_socket.recvfrom(4096)
    print(f"Received message from {client_address}: {message.decode()}")
  server_socket.sendto(b"Message received", client_address)
```

UDP Client Example:

```
import socket
client_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
server_address = ('localhost', 8080)
message = b"Hello, UDP Server!"
client_socket.sendto(message, server_address)
data, server = client_socket.recvfrom(4096)
print(f"Received response: {data.decode()}")
client_socket.close()
```

Difference between TCP and UDP Socket:

Feature	TCP	UDP
Connection	Connection-oriented (reliable)	Connectionless (unreliable)
Common Protocols	HTTP, HTTPS, FTP, WebSockets, SMTP	DNS, VoIP, RTP (for streaming)
Use Cases	Web apps, file transfer, email, real-time chat	Real-time applications (video, gaming)
Reliability	Guarantees delivery and order	No delivery guarantees, fast but less reliable
Performance	Slower (due to overhead)	Faster (due to minimal overhead)
Data Integrity	Ensures data integrity	No inherent data integrity check

❖ **Java Beans:**

JavaBeans are classes that encapsulate multiple objects into a single object. It also helps in accessing these object from multiple places. It is a reusable software component. Moreover, it provides easy maintenance. It contains several elements like Constructors, getter and setter methods and much more.

A JavaBean is a Java class that should follow the following conventions:

Class Definition: They are defined as public classes.

Serializability (Optional): While not mandatory, they often implement the Serializable interface to allow object data to be converted into a stream of bytes for storage or transmission.

No-argument Constructor: JavaBeans must have a public constructor that takes no arguments (often called a no-arg constructor). It simplifies object creation.

Data Encapsulation: They encapsulate data using private member variables (fields) to store the object's state.

Getter and Setter Methods: For each private field, there should be a corresponding public getter and setter method:

- Getter methods (usually prefixed with get or is for Booleans) retrieve the value of a field.
- Setter methods (usually prefixed with set) modify the value of a field.

JavaBean Properties:

- A JavaBean property is a named feature that can be accessed by the user of the object.
- The feature can be of any Java data type, containing the classes that you define.
- A JavaBean property may be read, write, read-only, or write-only.

JavaBean features are accessed through two methods in the JavaBean's implementation class:

Properties of setter Methods:

1. It must be public in nature.
2. The return type a should be void.
3. The setter method uses prefix set.
4. It should take some argument.

Example:

```
public class TestBean {
private String name;
public void setName(String name) {
this.name = name;
}
public String getName() {
return name;
 }  }
```

Properties of getter Methods:

1. It must be public in nature.
2. The return type should not be void.
3. The getter method uses prefix get.
4. It does not take any argument.

Example:

```
public class Test {
private boolean empty;
 public boolean getName(){
 return empty;
 }
 public boolean isempty(){
 return empty;
 }  }
```

1. GetPropertyName()

If the property name is first Name, the method name would be getFirstName() to read that property. This method is called the accessor.

Example:

```
package geeks;
public class Student implements java.io.Serializable {
 private int id;
 private String name;
 public Student() {}
 public void setId(int id) {
 this.id = id;
 }
 public int getId() {
  return id;
 }
 public void setName(String name) {
 this.name = name;
  }
 public String getName() {
  return name;
 }}
```

Output:

```
Id
Name
```

2. setPropertyName():

If the property name is first Name, the method name would be setFirstName() to write that property. This method is called the mutator.

Example:

```
package geeks;
public class Test {
public static void main(String args[]){
```

```
Student s = new Student();
s.setName("ANU");
System.out.println(s.getName());
} }
```

Output:

ANU

Advantages:

The following are the advantages of JavaBean:

- It is portable, compact, and easy.
- The JavaBean properties and methods can be exposed to another application.
- It provides an easiness to reuse the software components.

Disadvantages:

The following are the disadvantages of JavaBean:

- JavaBeans are mutable. So, it cannot take advantages of immutable objects.
- Creating the setter and getter method for each property separately may lead to the boilerplate code.

❖ **RMI(Remote Method Invocation):**

- RMI is a Java-based technology that allows for the invocation of methods on an object located in a different Java Virtual Machine (JVM), potentially running on a different physical machine.
- RMI is primarily used for distributed applications where the client and server are in different locations, and it abstracts the complexities of communication between them.
- RMI is not as commonly used today as it once was, primarily because of the rise of more lightweight technologies such as RESTful APIs and Web Sockets.

RMI Works:

RMI allows you to invoke methods on objects located remotely, and it involves several components:

1. **RMI Server**: This is the server-side component where the object resides. The server object implements a remote interface and provides the implementation of the methods that clients can call.

2. **RMI Client**: This is the client-side component that calls methods on the remote object.
3. **Remote Interface**: A Java interface that defines the methods that can be called remotely. This interface extends java.rmi.Remote.
4. **Stub**: On the client side, the stub acts as a proxy that forwards the method call to the remote object on the server.
5. **Skeleton**: On the server side, the skeleton (now often part of the RMI runtime) receives the method invocation and forwards it to the actual remote object.
6. **RMI Registry**: The registry holds references to remote objects and allows clients to look them up by name.

RMI Example:

Let's walk through a simple RMI example where a client calls a method on a server running in a different JVM.

1. Define the Remote Interface:

First, define the remote interface that extends java.rmi.Remote. This interface specifies the methods that can be invoked remotely.

Example:

```
import java.rmi.Remote;
import java.rmi.RemoteException
public interface Hello extends Remote {
String sayHello() throws RemoteException;
}
```

2. Implement the Remote Interface:

Next, implement the remote interface. This class will be the actual remote object that clients will interact with.

Example:

```
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
public class HelloImpl extends UnicastRemoteObject implements Hello {
public HelloImpl() throws RemoteException {
  super();
    }
```

```
public String sayHello() throws RemoteException {
    return "Hello, RMI!";
  }  }
```

3. Create the RMI Server:

The RMI server creates an instance of the remote object and binds it to the RMI registry. The server exposes the remote object so clients can look it up.

Example:

```
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
public class RMIServer {
public static void main(String[] args) {
    try {
        HelloImpl obj = new HelloImpl();
        Registry registry = LocateRegistry.createRegistry(1099);
        registry.rebind("HelloService", obj);
        System.out.println("HelloService bound to registry");
    } catch (Exception e) {
        e.printStackTrace();
    }  }  }
```

4. Create the RMI Client:

The client looks up the remote object in the RMI registry and calls methods on it as if it were a local object.

Example:

```
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
public class RMIClient {
 public static void main(String[] args) {
    try {
        Registry registry = LocateRegistry.getRegistry("localhost", 1099);
        Hello stub = (Hello) registry.lookup("HelloService");
        System.out.println(stub.sayHello());
    } catch (Exception e) {
```

```
        e.printStackTrace();
    } } }
```

Advantages of RMI:

1. **Java-centric**:

   Since RMI is built into Java, it allows easy integration between Java applications and is fully supported in the Java ecosystem.

2. **Distributed Computing**:

   RMI abstracts the complexities of network programming, making it easier to develop distributed applications where the communication between components is transparent.

3. **No Need for Manual Sockets**:

   RMI abstracts socket programming, so developers don't need to manually handle network connections or the details of serialization/deserialization of objects.

Disadvantages of RMI:

1. **Java-specific**:

   RMI is a Java-specific technology and may not work as well in environments that involve non-Java clients. For cross-platform, cross-language communication**,** web services (e.g., REST or SOAP) are often preferred.

2. **Complexity**:

   Setting up RMI can be more complex than using other web-based technologies like REST, especially when dealing with security.

3. **Firewall Issues**:

   RMI typically requires ports to be open in firewalls, which can complicate deployment, especially when dealing with enterprise environments or distributed systems across different networks.