Advanced Java

▼ UNIT 1

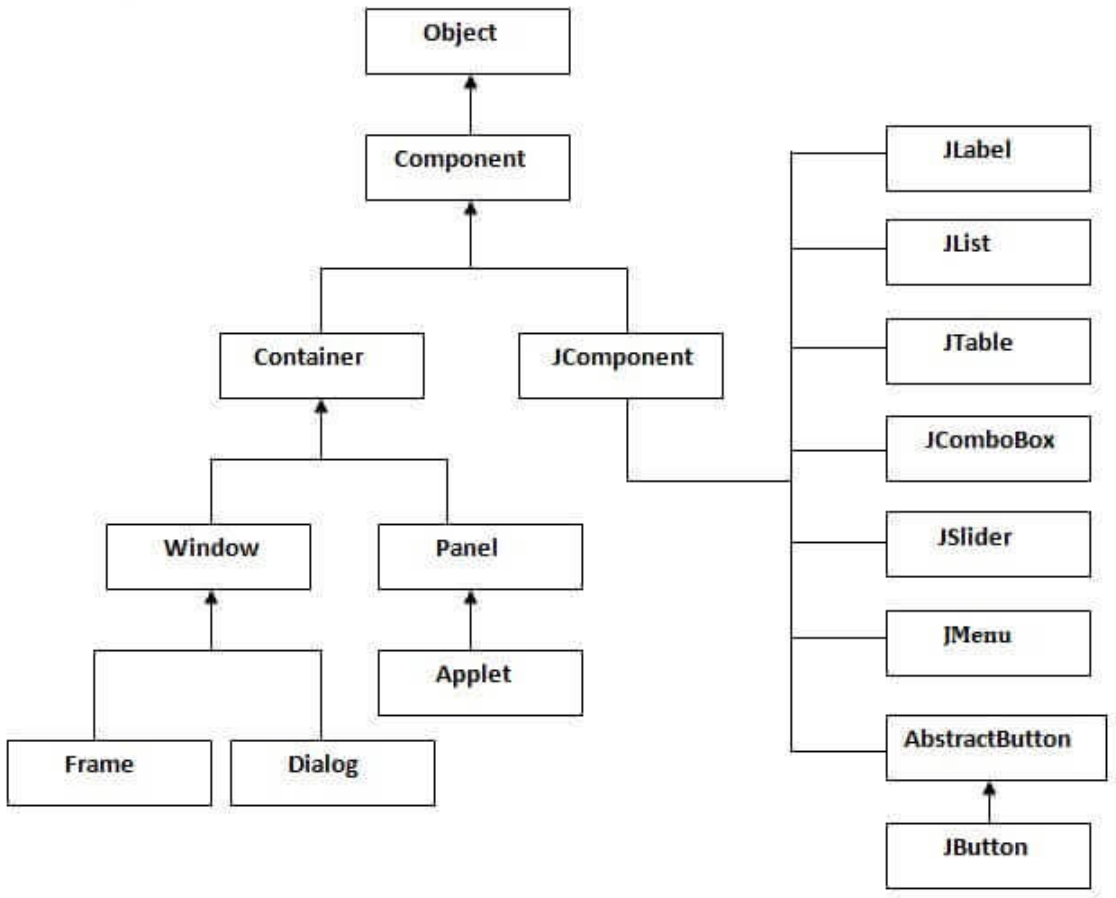
Need for Swing Components:

- **Cross-platform compatibility**: Swing components provide a consistent look and feel across different platforms. This allows developers to write Java GUI applications that can run on various operating systems without significant modifications
- **Rich set of components**: Swing offers a wide range of GUI components such as buttons, text fields, labels, lists, tables, and more. This variety allows developers to create complex and feature-rich user interfaces for their applications
- **Customization**: Swing components are highly customizable. Developers can easily modify the appearance and behavior of Swing components to meet the specific requirements of their applications

Difference between AWT & Swing:

AWT	Swing
Java AWT is an API to develop GUI applications in Java	Swing is a part of Java Foundation Classes and is used to create various applications.
The components of Java AWT are heavy weighted	The components of Java Swing are light weighted.
Java AWT has comparatively less functionality as compared to Swing.	ava Swing has more functionality as compared to AWT.
The execution time of AWT is more than Swing.	The execution time of Swing is less than AWT.
The components of Java AWT are platform dependent.	The components of Java Swing are platform independent.

Components Hierarchy:



Swing Codes:

Java Swing Tutorial - javatpoint

Java Swing Tutorial with example of JButton, Difference between AWT and swing, simple java swing example, example of swing by inheritance, JRadioButton, JTextField, JTextArea, JList, JColorChooser classes that are found in javax.swing package.

https://www.javatpoint.com/java-swing

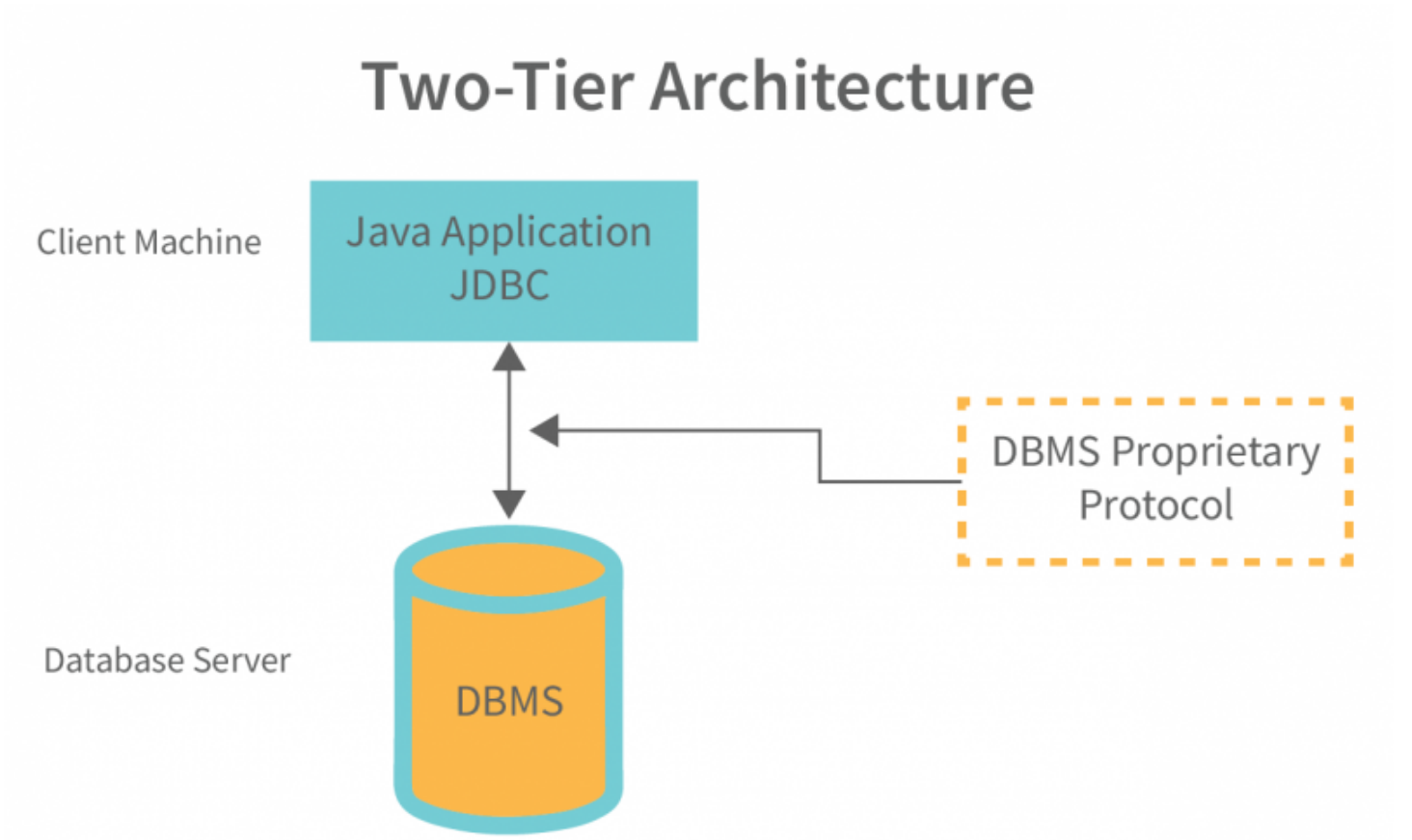
Introduction to JDBC:

- JDBC stands for Java Database Connectivity.
- JDBC is a Java API to connect and execute the query with the database.
- JDBC API uses JDBC drivers to connect with the database.
- There are four types of JDBC drivers:
  - JDBC-ODBC Bridge Driver,
  - Native Driver (partially java driver)
  - Network Protocol Driver (fully java driver)
  - Thin Driver (fully java driver)
- 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.
- 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).

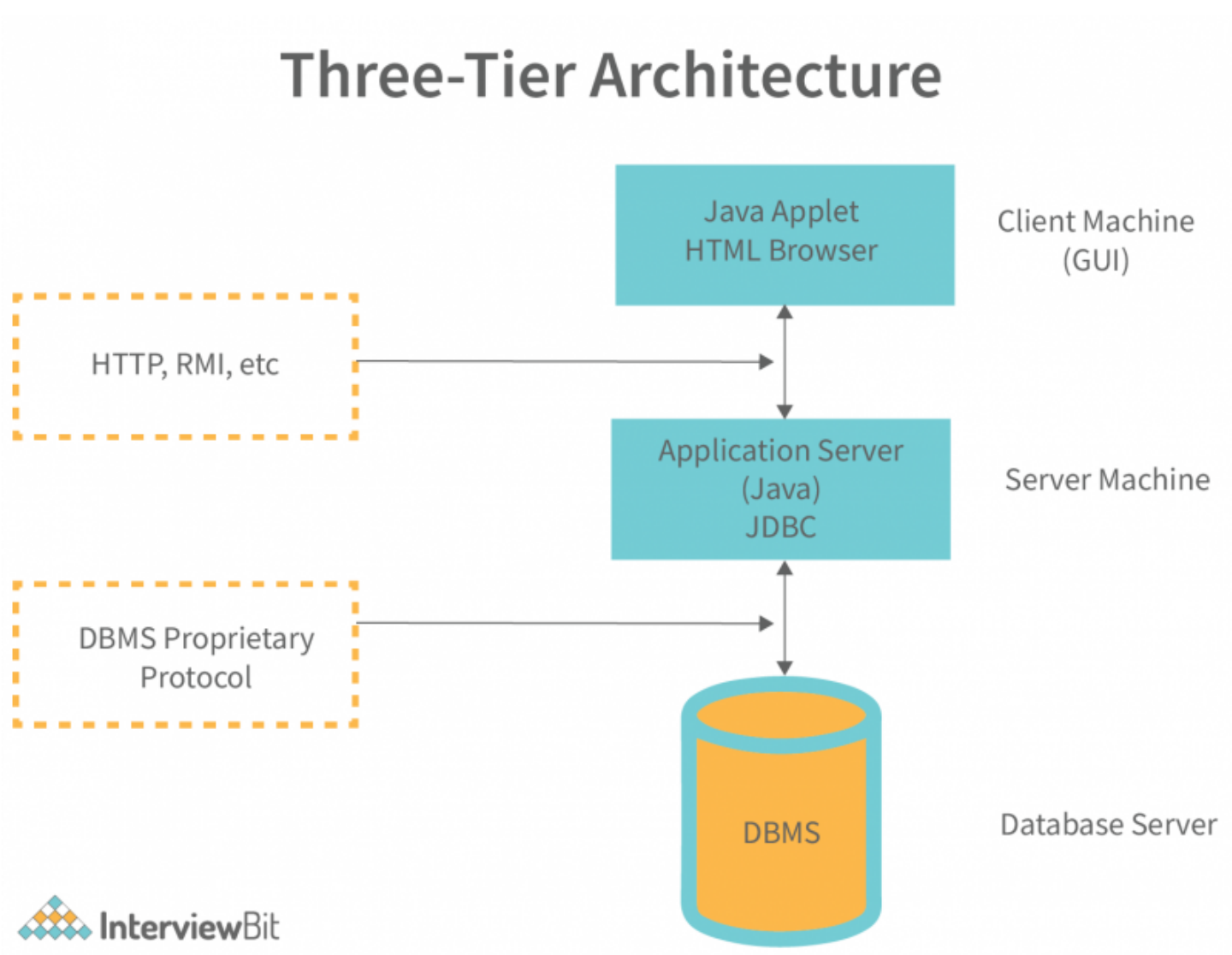
JDBC Architecture:

The architecture of the JDBC consists of two and three tiers model in order to access the given database.

- **Two-tier 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.



- **Three-tier model:** In this model, the queries of the user are being sent to the middle-tier services, from where the commands are sent again to the source of data. The answers to those queries are reverted to the middle tier, and from there, it is again sent to the user.



▼ JDBC Drivers:

JDBC-ODBC bridge driver

The 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.

Advantages:

- easy to use.
- can be easily connected to any database.

Disadvantages:

- Performance degraded because JDBC method call is converted into the ODBC function calls.
- The ODBC driver needs to be installed on the client machine.

Native-API driver

The Native API driver uses the client-side libraries of the database. The driver converts JDBC method calls into native calls of the database API. It is not written entirely in java.

Advantage:

- performance upgraded than JDBC-ODBC bridge driver.

Disadvantage:

- The Native driver needs to be installed on the each client machine.
- The Vendor client library needs to be installed on client machine.

Network Protocol Driver

The Network Protocol driver uses middleware (application server) that converts JDBC calls directly or indirectly into the vendor-specific database protocol. It is fully written in java.

Advantages:

- No client side library is required because of application server that can perform many tasks like auditing, load balancing, logging etc.

Disadvantages:

- Network support is required on client machine.
- Requires database-specific coding to be done in the middle tier.
- Maintenance of Network Protocol driver becomes costly because it requires database-specific coding to be done in the middle tier.

Thin Driver

The thin driver converts JDBC calls directly into the vendor-specific database protocol. That is why it is known as thin driver. It is fully written in Java language.

Advantages:

- Better performance than all other drivers.
- No software is required at client side or server side.

Disadvantages:

- Drivers depend on the Database.

▼ Java Database Connectivity steps:

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

- Register the Driver class:

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

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

- Create connection:

  The **getConnection()** method of DriverManager class is used to establish connection with the database.

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

- Create statement:

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

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

- Execute queries:

  The executeQuery() 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.

  ```
  ResultSet rs=stmt.executeQuery("select * from emp");

  while(rs.next()){
  System.out.println(rs.getInt(1)+" "+rs.getString(2));
  }
  ```

- Close connection:

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

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

Connection interface:

A Connection is a session between a Java application and a database. It helps to establish a connection with the database.

The Connection interface is a factory of Statement, PreparedStatement, and DatabaseMetadata, i.e., an 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(), setAutoCommit(), setTransactionIsolation(), etc.

Commonly used methods of Connection interface:

- **public Statement createStatement():** creates a statement object that can be used to execute SQL queries.
- **public void setAutoCommit(boolean status):** is used to set the commit status. By default, it is true.
- **public void commit():** saves the changes made since the previous commit/rollback is permanent.
- **public void rollback():** Drops all changes made since the previous commit/rollback.
- **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:

- **public ResultSet executeQuery(String sql):** is used to execute SELECT query. It returns the object of ResultSet.
- **public int executeUpdate(String sql):** is used to execute specified query, it may be create, drop, insert, update, delete etc.
- **public boolean execute(String sql):** is used to execute queries that may return multiple results.

ResultSet Interface:

The object of ResultSet maintains a cursor pointing to a row of a table. Initially, cursor points to before the first row.

Commonly used methods of ResultSet interface:

- **public boolean next():** is used to move the cursor to the one row next from the current position.
- **public boolean previous():** is used to move the cursor to the one row previous from the current position.
- **public boolean first():** is used to move the cursor to the first row in result set object.
- **public boolean last():** is used to move the cursor to the last row in result set object.

ResultSet Types:

- **Forward Only (ResultSet.*TYPE_FORWARD_ONLY):***

  This type of ResultSet instance can move only in the forward direction from the first row to the last row. ResultSet can be moved forward one row by calling the next() method.

  ```
  Statement stmt = connection.createStatement(ResultSet.TYPE_FORWARD_ONLY, ResultSet.CONCUR_READ_ONL
  Y);
  ResultSet rs = stmt.executeQuery("select * from tbluser");
  ```

- **Scroll Insensitive (ResultSet.*TYPE_SCROLL_INSENSITIVE*):**

  Scroll Insensitive ResultSet can scroll in both forward and backward directions. It can also be scrolled to an absolute position by calling the absolute() method. But it is not sensitive to data changes. It will only have data when the query was executed and ResultSet was obtained. It will not reflect the changes made to data after it was obtained.

  ```
  Statement stmt = connection.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,          ResultSet.C
  ONCUR_READ_ONLY);
  ResultSet rs = stmt.executeQuery("select * from tbluser");
  ```

- **Scroll Sensitive (ResultSet.*TYPE_SCROLL_SENSITIVE):***

  Scroll Sensitive ResultSet can scroll in both forward and backward directions. It can also be scrolled to an absolute position by calling the absolute() method. But it is sensitive to data changes. It will reflect the changes made to data while it is open.

  ```
  Statement stmt = connection.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,        ResultSet.CONCU
  R_READ_ONLY);
  ResultSet rs = stmt.executeQuery("select * from tbluser");
  ```

- **Read Only (ResultSet.*CONCUR_READ_ONLY):***

  It is the default concurrency model. We can only perform Read-Only operations on ResultSet Instance. No update Operations are allowed.

- **Updatable (ResultSet.*CONCUR_UPDATABLE):***

  In this case, we can perform update operations on ResultSet instance.

PreparedStatement Interface:

The PreparedStatement interface is a sub interface of Statement. It is used to execute parameterized query.

The performance of the application will be faster if you use PreparedStatement interface because query is compiled only once.

Methods for PreparedStatement Interface:

- **public void setInt(int paramIndex, int value):** sets the integer value to the given parameter index.
- **public void setString(int paramIndex, String value):** sets the String value to the given parameter index.

- **public void setFloat(int paramIndex, float value):** sets the float value to the given parameter index.
- **public int executeUpdate():** executes the query. It is used for create, drop, insert, update, delete etc
- **public ResultSet executeQuery():** executes the select query. It returns an instance of ResultSet.

For Inserting:

```
import java.sql.*;
class InsertPrepared{
public static void main(String args[]){
try{
Class.forName("oracle.jdbc.driver.OracleDriver");

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

PreparedStatement stmt=con.prepareStatement("insert into Emp values(?,?)");
stmt.setInt(1,101);//1 specifies the first parameter in the query
stmt.setString(2,"Ratan");

int i=stmt.executeUpdate();
System.out.println(i+" records inserted");

con.close();

}catch(Exception e){ System.out.println(e);}

}
}
```

For Updating:

```
PreparedStatement stmt=con.prepareStatement("update emp set name=? where id=?");
stmt.setString(1,"Sonoo");//1 specifies the first parameter in the query i.e. name
stmt.setInt(2,101);

int i=stmt.executeUpdate();
System.out.println(i+" records updated");
```

For Retrieving:

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

SavePoint:

When you set a save point you define a logical rollback point within a transaction. If an error occurs past a save point, you can use the rollback method to undo either all the changes or only the changes made after the save point.

You can set a save point in a database using the **setSavepoint(String savepointName)** method of the Connection interface, this method accepts a string value representing the name of the save point and returns a Savepoint object.

```
// Java program to demonstrate how to make a save point

import java.io.*;
import java.sql.*;

class GFG {
    public static void main(String[] args)
    {
        // db credentials
        String jdbcEndpoint
```

```
            = "jdbc:mysql://localhost:3000/GEEKSFORGEEKS";
        String userid = "GFG";
        String password = "GEEKSFORGEEKS";

        // create a connection to db
        Connection connection = DriverManager.getConnection(
            jdbcEndpoint, userid, password);

        // construct a query
        Statement deleteStmt = connection.createStatement();
        String deleteQuery
            = "DELETE FROM USER WHERE AGE > 15";

        // Disable auto commit to connection
        connection.setAutoCommit(false);

        /* Table USER
    +--------+---------+------------+
    | USR_ID | NAME | AGE    |
    +--------+---------+------------+
    |    1 | GFG_1   | 10        |
    |    2 | GFG_2 | 20      |
    |    3 | GFG_3 | 25      |
    +--------+---------+------------+
    */

        // Create a savepoint object before executing the
        // deleteQuery
        Savepoint beforeDeleteSavepoint
            = connection.setSavepoint();

        // Executing the deleteQuery
        ResultSet res
            = deleteStmt.executeQuery(deleteQuery);

        /* Table USER after executing deleteQuery
    +--------+---------+------------+
    | USR_ID | NAME | AGE    |
    +--------+---------+------------+
    |    1 | GFG_1   | 10        |
    +--------+---------+------------+
    */

        // Rollback to our beforeDeleteSavepoint
        connection.rollback(beforeDeleteSavepoint);
        connection.commit();

        /* Table USER after rollback
        +--------+---------+------------+
        | USR_ID | NAME | AGE    |
        +--------+---------+------------+
        |    1 | GFG_1 | 10      |
        |    2 | GFG_2 | 20      |
        |    3 | GFG_3 | 25      |
        +--------+---------+------------+
        */
    }
}
```

Batch Processing in JDBC:

Instead of executing a single query, we can execute a batch (group) of queries. It makes the performance fast. It is because when one sends multiple statements of SQL at once to the database, the communication overhead is reduced significantly, as one is not communicating with the database frequently, which in turn results to fast performance.

The java.sql.Statement and java.sql.PreparedStatement interfaces provide methods for batch processing.

Methods of Statement interface:

- **void addBatch(String query):** The addBatch(String query) method of the CallableStatement, PreparedStatement, and Statement is used to single statements to a batch.
- **int[] executeBatch():** The executeBatch() method begins the execution of all the grouped together statements.

```
import java.sql.*;
class FetchRecords{
public static void main(String args[])throws Exception{
Class.forName("oracle.jdbc.driver.OracleDriver");
Connection con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe","system","oracle");
con.setAutoCommit(false);

Statement stmt=con.createStatement();
stmt.addBatch("insert into user420 values(190,'abhi',40000)");
stmt.addBatch("insert into user420 values(191,'umesh',50000)");

stmt.executeBatch();//executing the batch

con.commit();
con.close();
}}
```

```
import java.sql.*;
import java.io.*;
class BP{
public static void main(String args[]){
try{

Class.forName("oracle.jdbc.driver.OracleDriver");
Connection con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe","system","oracle");

PreparedStatement ps=con.prepareStatement("insert into user420 values(?,?,?)");

BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
while(true){

System.out.println("enter id");
String s1=br.readLine();
int id=Integer.parseInt(s1);

System.out.println("enter name");
String name=br.readLine();

System.out.println("enter salary");
String s3=br.readLine();
int salary=Integer.parseInt(s3);

ps.setInt(1,id);
ps.setString(2,name);
ps.setInt(3,salary);

ps.addBatch();
System.out.println("Want to add more records y/n");
String ans=br.readLine();
if(ans.equals("n")){
break;
}
```

```
}
ps.executeBatch();// for executing the batch

System.out.println("record successfully saved");

con.close();
}catch(Exception e){System.out.println(e);}

}}
```

CallableStatement Interface:

CallableStatement interface is used to call the stored procedures and functions.

The prepareCall() method of Connection interface returns the instance of CallableStatement. Syntax is given below:

```
CallableStatement stmt=con.prepareCall("{call myprocedure(?,?)}");
```

To call the stored procedure, you need to create it in the database. Here, we are assuming that stored procedure looks like this:

```
create or replace procedure "INSERTR"
(id IN NUMBER,
name IN VARCHAR2)
is
begin
insert into user420 values(id,name);
end;
```

In this example, we are going to call the stored procedure INSERTR that receives id and name as the parameter and inserts it into the table user420.

```
import java.sql.*;
public class Proc {
public static void main(String[] args) throws Exception{

Class.forName("oracle.jdbc.driver.OracleDriver");
Connection con=DriverManager.getConnection(
"jdbc:oracle:thin:@localhost:1521:xe","system","oracle");

CallableStatement stmt=con.prepareCall("{call insertR(?,?)}");
stmt.setInt(1,1011);
stmt.setString(2,"Amit");
stmt.execute();

System.out.println("success");
}
}
```

▼ Unit 2

▼ Servlets:

- Servlet technology is used to create a web application
- There are many interfaces and classes in the Servlet API such as Servlet, GenericServlet, HttpServlet, ServletRequest, ServletResponse, etc.
- Servlet is an interface that must be implemented for creating any Servlet.
- Servlet is a class that extends the capabilities of the servers and responds to the incoming requests. It can respond to any requests.
- Servlet is a web component that is deployed on the server to create a dynamic web page.