JDBC(Java Database Connectivity)

Table of Contents

What is JDBC

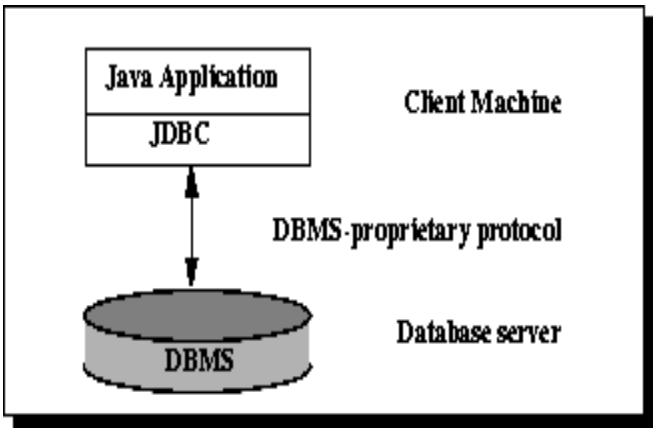
The JDBC is a Java API that provides the facilities to connect java applications to a Relational Database. Using JDBC API's, Java applications can access any kind of tabular data, stored in a Relational Databases.
The JDBC API's works as a bridge between Java applications and Relational Databases.

Activities of JDBC API's:

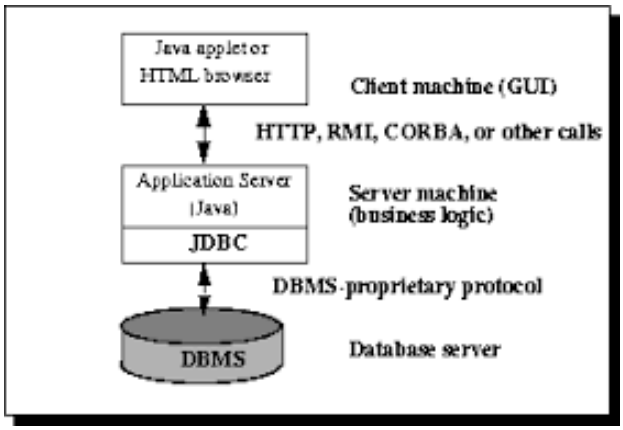
1. Connect to a data source (DBMS, RDBMS).
2. Send queries and update statements to the database.
3. Retrieve and process the results received from the database an answer to your query.

JDBC Architecture

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



Two-tier Architecture for Data Access



Three-tier Architecture for Data Access

In the **two-tier architecture**, a Java application interacts directly to the database. This requires a JDBC driver that can communicate with the particular database being accessed. A Java Application commands are delivered to the database, and the results of those statements are sent back to the application.

In the **three-tier architecture**, commands are sent to a 'midde tier' of services, which then sends the commands to the database. The database processes the commands and sends the results back to the middle tier, which then sends them to the Java Application.

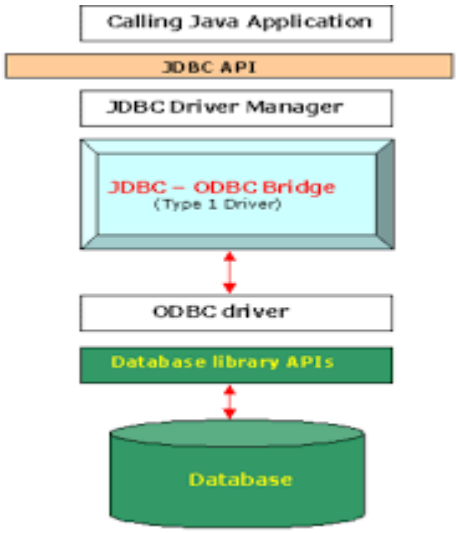
JDBC Drivers

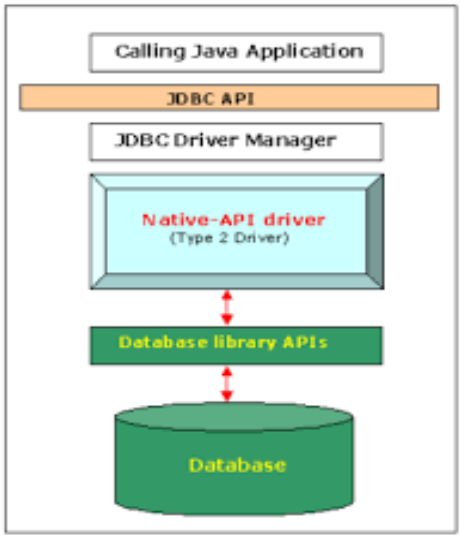
1. JDBC drivers are software components enabling a Java application to interact with a database.
2. To connect with a particular database, JDBC requires drivers for each database.
3. Each Database vendor provides JDBC drivers and these are database dependent.

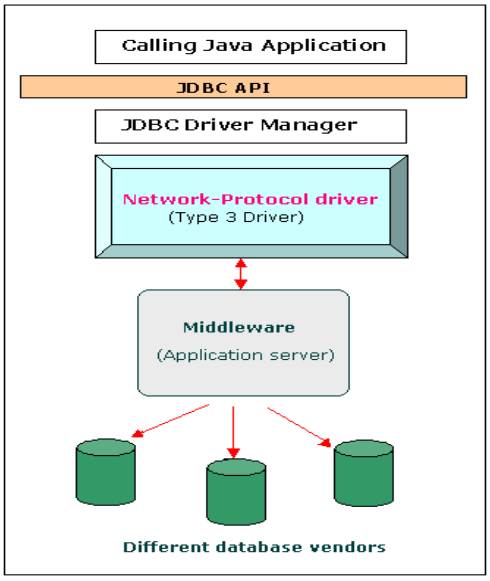
4.Whenever you change the database, You must change database drivers.

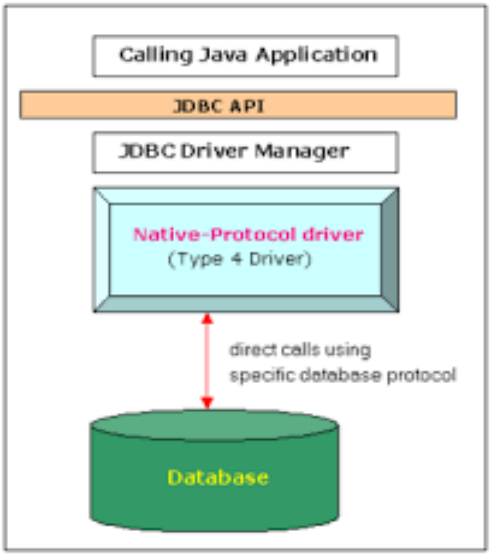
Types of drivers

1. JDBC-ODBC bridge
2. Native-API driver
3. Network Protocol driver
4. Database-Protocol driver (Pure Java driver) or thin driver









1) 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. This is now discouraged because of thin driver.

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.

2) 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.

Advantages:

- Performance upgraded than JDBC-ODBC bridge driver.

Disadvantages:

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

3) 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.

4) 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.

Disadvantages:

- Drivers depends on the Database.
- No software is required at client side or server side.

Connection Interface

A **connection** Interface defines a connection object of Database that provides information about its tables, its supported SQL grammar, its stored procedures, the capabilities of this connection, and so on.

Getting Connection object

To get connection object we need to call **getConnection()** method of **DriverManager** class. It will return connection object associated with database.

Connection con = DriverManager.getConnection(dbUrl,username,password);

Commonly Used Methods of Connection Interface

Methods	Description
Statement createStatement()	Creates a Statement object for sending SQL statements to the database.
PreparedStatement prepareStatement(String sql)	Creates a PreparedStatement object for sending parameterized SQL statements to the database.
CallableStatement prepareCall(String sql)	Creates a CallableStatement object for calling database stored procedures
String nativeSQL(String sql)	Converts the given SQL statement into the system's native SQL grammar
void rollback()	Undoes all changes made in the current transaction and releases any database locks currently held by this Connection object.
void commit()	Makes all changes made since the previous commit/rollback permanent and releases any database looks currently held by this connection object.
void close()	Releases this Connection object of database and JDBC resources immediately instead of waiting for them to be automatically released.

Here are some examples of classes that directly or indirectly implement the java.sql.Connection interface:
oracle.jdbc.OracleConnectionWrapper
com.mysql.jdbc.ConnectionImpl
com.microsoft.sqlserver.jdbc.SQLServerConnection

DriverManager Class

The JDBC **DriverManager** class defines an object which can connect Java application to a JDBC driver. **DriverManager** is a backbone of the JDBC architecture.

- The DriverManager class is responsible to load the driver classes and create the Connection Object.
- By default loads the drivers from system property – **"jdbc.drivers" .** you can also register drivers using **Class.forName() .**
- **DriverManager.getConnection()** returns the connection object.

Requirements for Establishing Connection

Step-1

1. Install MySQL Database or Oracle Database.
2. Create Database in MySQL

   create database j2ee; //Here j2ee is database name.

   use j2ee; // To select the database
3. Create Table in Database

```
create table employees(
        emp_id int not null auto_increment,
        emp_name varchar(50),
        email varchar(50),
        mobile bigint,
        dept varchar(20),
        salary bigint,
        primary key(emp_id)
);
```

4. Insert Record

```
INSERT INTO employees VALUES(
                0,'Bidit vats',Biditvats@gmail.com',9916712669,'IT',50000
);
```

Establishing a Connection

If you want to connect your Java Application with database. First, you need to establish a connection with the data source you want to use. A data source can be a DBMS or RDBMS.

Note: All the JDBC API's are available in java.sql.* and javax.sql.* packages. Both packages use Checked Exception **java.sql.SQLException** , So you must have to **handle** or **declare** this Exception.

Required parameters: (For MySQL Database)

1. String DB_DRIVER = "com.mysql.jdbc.Driver";
2. String DB_URL = "jdbc:mysql://localhost:3306/j2ee";
3. String USERNAME = "root";
4. String PASSWORD = "password";

Steps to establish a Connection

1. **Register Driver Class**

   Class.forName(DB_DRIVER);

2. **Getting Connection Object**

   Connection con = DriverManager.getConnection(DB_URL,USERNAME,PASSWORD);

3. **Creating statement**

   Statement stmt = con.createStatement();

4. **Executing Query**

   String sql = "SELECT * FROM employees";
   ResultSet rs = stmt.executeQuery(sql);

5. **Processing ResultSet**

```
while(rs.next()) {
        System.out.println(rs.getInt(1));
        System.out.println(rs.getString(2));
        System.out.println(rs.getString(3));
       //All Columns
  }
```

Example: JdbcTest.java

```
Import java.sql.*;
public class JdbcTest {
    static final String DB_DRIVER = "com.mysql.jdbc.Driver";
    static final String DB_URL = "jdbc:mysql://localhost:3306/j2ee";
    static final String USERNAME = "root";
    static final String PASSWORD = "password";

    static Connection con = null;
```

```
    public static void main(String[] args) throws Exception {
        String sql = "SELECT * FROM employees";
        Statement stmt = null;
        try {
            Class.forName(DB_DRIVER);
            Connection con =  DriverManager.getConnection(DB_URL,USERNAME,PASSWORD);
            stmt = con.createStatement();
            Resultset rs = stmt.executeQuery(sql);

            while(rs.next()) {
                System.out.println(rs.getInt(1));
                System.out.println(rs.getString(2));
                System.out.println(rs.getString(3));
                System.out.println(rs.getLong(4));
                System.out.println(rs.getString(5));
                System.out.println(rs.getLong(6));
            }
        }catch(Exception ex) {
            ex.printstackTrace();
        }finally {
            if(stmt != null)
                stmt.close();
        }
    }
}
```

Statements

JDBC API's define three types of statements to execute the SQL statements.

1. **Statement**
2. **PreparedStatement**
3. **CallableStatement**

Statement

The Statement interface defines an Object that is used for executing a static SQL statement and returning the results that produces by SQL statement.

By default, only one **ResultSet** object per **Statement** object can be open at the same time.

Creating Statement

To create *Statement* object, we call *createStatement()* method of *Connection* object. That creates the object of *Statement.*

```
//Getting Connection Object
Connection con = DriverManager.getConnection(dburl, username, password);
//Creating Statement using Connection
Statement stmt = con.createStatement();
```

Commonly Used methods of Statement

- **boolean execute (String SQL)**: Executes the given SQL statement, which may return multiple results.
- **int executeUpdate (String SQL)**: Returns the number of rows affected by the execution of the SQL statement. Use this method to execute SQL statements for which you expect to get a number of rows affected - for example, an INSERT, UPDATE, or DELETE statement.
- **ResultSet executeQuery (String SQL)**: Returns a ResultSet object. Use this method when you expect to get a result set. It is used to execute SELECT query.
- **void addBatch(String sql)**: Adds the given SQL command to the current list of commands for this Statement object.
- **Int[ ] executeBatch():** submits a batch of commands to the database for execution and if all commands execute successfully, returns an array of update counts.
- **ResultSet getGeneratedKeys()**: Retrieves any auto-generated keys created as a result of executing this Statement object.
- **Void close()**: releases this Connection object of database and JDBC resources immediately instead of waiting for them to be automatically released.

PreparedStatement

PreparedStatement is an object that reperesents a precompiled SQL statement. A SQL statement is precompiled and stored in a PreparedStatement object. Then you can execute this statement multiple times.

PreparedStatement allows parameterized query and define several methods to set the parameters.

Let's see the example of parameterized query:

1. String sql="insert into emp values(?,?,?)";

As you can see, we are passing parameter (?) for the values. Its value will be set by calling the setter methods of PreparedStatement.

- **Why use PreparedStatement?**

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

- **Creating PreparedStatement**

To create *PreparedStatement* object, we call *prepareStatement(String sql)* method of *Connection* object by passing SQL Query.

```
//Getting Connection Object
Connection con = DriverManager.getConnection(dburl, username, password);
//Creating PreparedStatement using Connection
String sql = "SELECT * FROM employees";
PreparedStatement pstmt = con.prepareStatement(sql);
ResultSet rs = pstmt.executeQuery();
```

Commonly Used methods of PreparedStatement :

The important methods of PreparedStatement interface are given below:

Method	Description
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.
boolean execute()	Executes the SQL statement in this PreparedStatement object, which may be any kind of SQL statement.
public int executeUpdate()	Executes the SQL statement in this PreparedStatement object, which must be an SQL Data Manipulation language(DML) statement, suach as INSERT, UPDATE, or DELETE; or an SQL statement that returns nothing, such as a DDL statement.
public ResultSet executeQuery()	Executes the SQL query in this PreparedStatement object and returns the ResultSet object generated by the query.

Example PreparedStatement :

```
Import java.sql.*;
public class JdbcPStatement {
        String DB_DRIVER = "com.mysql.jdbc.Driver";
        String DB_URL = "jdbc:mysql://localhost:3306/j2ee";
        String USERNAME = "root";
        String PASSWORD = "password";

        Connection con = null;

    public static void main(String[] args) throws Exception {
            try {
                    Class.forName(DB_DRIVER);
                    Connection con =  DriverManager.getConnection(DB_URL,USERNAME,PASSWORD);
                     String sql ="INSERT INTO employees.VALUES(?,?,?,?,?,?)";
                     PreparedStatement pstmt = con.prepareStatement(sql);
                     pstmt.setInt(1,0);
                     pstmt.setString(2,"Bidit");
                     pstmt.setString(3,"Biditvats@gmail.com");
                     pstmt.setInt(4,889255);
                     pstmt.setString(5,"DEVELOPMENT");
                     pstmt.setInt(6,60000);

                     int i = pstmt.executeUpdate( );
                     if(i > 0)
                            System.out.println("Record have been saved successfully");

            }catch(Exception ex) {
                    ex.printstackTrace();
            }
        }
}
```

CallableStatement

Just as a Connection object creates the Statement and PreparedStatement objects, it also creates the CallableStatement object, which would be used to execute a call to a database stored procedure.

- Callablestatement is used to execute SQL stored procedures. The JDBC API provides a stored procedure SQL syntax that allows stored procedures to be called in a standard way for all RDBMS.

- We can have business logic on the database by the use of stored procedures and functions that will make the performance better because these are precompiled.
- Suppose you need the get the age of the employee based on the date of birth, you may create a function that receives date as the input and returns age of the employee as the output.

Q. What is the difference between stored procedures and functions.?

The differences between stored procedures and functions are given below:

Stored Procedure	Function
is used to perform business logic.	is used to perform calculation.
must not have the return type.	must have the return type.
may return 0 or more values.	may return only one values.
We can call functions from the procedure.	Procedure cannot be called from function.
Procedure supports input and output parameters.	Function supports only input parameter.
Exception handling using try/catch block can be used in stored procedures.	Exception handling using try/catch can't be used in user defined functions.

- let us write same stored procedure for MySQL as follows to create it in EMP database –

```
DELIMITER $$


DROP PROCEDURE IF EXISTS `EMP`.`getEmpName` $$
CREATE PROCEDURE `EMP`.`getEmpName`
  (IN EMP_ID INT, OUT EMP_FIRST VARCHAR(255))
```

```
BEGIN
   SELECT first INTO EMP_FIRST
   FROM Employees
   WHERE ID = EMP_ID;
END $$


DELIMITER ;
```

Three types of parameters exist: IN, OUT, and INOUT. The PreparedStatement object only uses the IN parameter. The CallableStatement object can use all the three.

Here are the definitions of each −

Parameter	Description
IN	A parameter whose value is unknown when the SQL statement is created. You bind values to IN parameters with the setXXX() methods.
OUT	A parameter whose value is supplied by the SQL statement it returns. You retrieve values from theOUT parameters with the getXXX() methods.
INOUT	A parameter that provides both input and output values. You bind variables with the setXXX() methods and retrieve values with the getXXX() methods.

The following code snippet shows how to employ the **Connection.prepareCall()** method to instantiate a **CallableStatement**object based on the preceding stored procedure −

```
CallableStatement cstmt = null;
try {
   String SQL = "{call getEmpName (?, ?)}";
   cstmt = conn.prepareCall (SQL);
   . . .
}
```

```
catch (SQLException e) {
  . . .
}
finally {
  . . .
}
```

The String variable SQL, represents the stored procedure, with parameter placeholders.

Using the CallableStatement objects is much like using the PreparedStatement objects. You must bind values to all the parameters before executing the statement, or you will receive an SQLException.

If you have IN parameters, just follow the same rules and techniques that apply to a PreparedStatement object; use the setXXX() method that corresponds to the Java data type you are binding.

When you use OUT and INOUT parameters you must employ an additional CallableStatement method, registerOutParameter(). The registerOutParameter() method binds the JDBC data type, to the data type that the stored procedure is expected to return.

Once you call your stored procedure, you retrieve the value from the OUT parameter with the appropriate getXXX() method. This method casts the retrieved value of SQL type to a Java data type.

Example CallableStatement :

```
//STEP 1. Import required packages
import java.sql.*;


public class JDBCExample {
  // JDBC driver name and database URL
  static final String JDBC_DRIVER = "com.mysql.jdbc.Driver";
  static final String DB_URL = "jdbc:mysql://localhost/EMP";


  //  Database credentials
  static final String USER = "root";
  static final String PASS = "password";
```

```
public static void main(String[] args) {
Connection conn = null;
CallableStatement stmt = null;

  //STEP 2: Register JDBC driver
  Class.forName("com.mysql.jdbc.Driver");

  //STEP 3: Open a connection
  System.out.println("Connecting to database...");
  conn = DriverManager.getConnection(DB_URL,USER,PASS);

  //STEP 4: Execute a query
  System.out.println("Creating statement...");
  String sql = "{call getEmpName (?, ?)}";
  stmt = conn.prepareCall(sql);

  //Bind IN parameter first, then bind OUT parameter
  int empID = 102;
  stmt.setInt(1, empID); // This would set ID as 102
  // Because second parameter is OUT so register it
  stmt.registerOutParameter(2, java.sql.Types.VARCHAR);

  //Use execute method to run stored procedure.
  System.out.println("Executing stored procedure..." );
  stmt.execute();
```

```
    //Retrieve employee name with getXXX method
    String empName = stmt.getString(2);
    System.out.println("Emp Name with ID:" +
            empID + " is " + empName);
    stmt.close();
    conn.close();
  System.out.println("Goodbye!");
}//end main
}//end JDBCExample
```

Now let us compile the above example as follows −

```
C:\>javac JDBCExample.java
C:\>
```

When you run **JDBCExample**, it produces the following result −

```
C:\>java JDBCExample
Connecting to database...
Creating statement...
Executing stored procedure...
Emp Name with ID:102 is Virat
Goodbye!
C:\>
```

Processing ResultSet

ResultSet is an object that represent a table of data, which is generated by executing a statement that queries the database.

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

The *next()* method moves the cursor to the next row, and it returns false when there are no more rows in the *ResultSet* object, it can be used in a while loop to iterate through the result set.

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 int getInt(int columnIndex):	is used to return the data of specified column index of the current row as int.
6) public int getInt(String columnName):	is used to return the data of specified column name of the current row as int.
7) public String getString(int columnIndex):	is used to return the data of specified column index of the current row as String.
8) public String getString(String columnName):	is used to return the data of specified column name of the current row as String.

```
String sql = “SELECT * FROM employees”;
ResultSet rs = stmt.executeQuery();

while(rs.next()) {
        System.out.println(rs.getInt(1));
        System.out.println(rs.getstring(2));
        System.out.println(rs.getString(3));
        System.out.println(rs.getLong(4));
        System.out.println(rs.getString(5));
        System.out.println(rs.getLong(6));
}
```

CRUD Operations

```
package com.kalibermind.jdbc;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;

public class CRUDOperations  {
String driver="com.mysql.jdbc.Driver";
String dburl="jdbc:mysql://localhost:3306/j2ee";
String username="root";
String password="password";
Connection con=null;

public CRUDOperation()
{
	try{
	Class.forName(driver);
	con=DriverManager.getConnection(dburl,username,password);
	}
	catch(Exception e)
	{
		e.printStackTrace();
	}
	}

public void createEmployee(String empName,String email,long mobile,String dept,long salary)
{
	String sql="INSERT INTO
employees(emp_name,emp_email,mobile,dept,salary)"+"VALUES(?,?,?,?,?)";
	try{
		PreparedStatement psmt=con.prepareStatement(sql);
		psmt.setString(1, empName);
		psmt.setString(2, email);
		psmt.setLong(3, mobile);
		psmt.setString(4, dept);
		psmt.setLong(5, salary);
		int i=psmt.executeUpdate();
		if(i>0)
```

```
                System.out.println("Record has been saved successfully!");
            }
        catch(Exception ex)
            {
                ex.printStackTrace();
            }
        }
public void getEmployee(int empId)
{
        String sql="SELECT * FROM employee WHERE emp_id=?";
        try{
                PreparedStatement psmt=con.prepareStatement(sql);
                psmt.setInt(1, empId);
                ResultSet rs=psmt.executeQuery();
                while(rs.next())
                {
                        System.out.println(rs.getString(2));
                        System.out.println(rs.getString(3));
                        System.out.println(rs.getLong(4));
                        System.out.println(rs.getString(5));
                        System.out.println(rs.getLong(6));
                }
        }
        catch(Exception ex)
        {
                ex.printStackTrace();
        }
}
public void updateEmployee(int empId,long mobile,long salary)
{
        String s="UPDATE employee SET mobile=?,salary=? WHERE emp_id=?";
        try{
                PreparedStatement psmt=con.prepareStatement(s);
                psmt.setLong(1, mobile);
                psmt.setLong(2, salary);
                psmt.setInt(3, empId);
                int i=psmt.executeUpdate();
                if(i>0)
                        System.out.println("Record has been updated successfully!");
        }
```

```
        catch(Exception ex)
        {
                ex.printStackTrace();
        }
}
public void deleteEmployee(int empId)
{
        String sql="DELETE FROM employee WHERE emp_id=?";
        try{
                PreparedStatement psmt=con.prepareStatement(sql);
                psmt.setInt(1, empId);
                int i=psmt.executeUpdate();
                if(i>0)
                        System.out.println("Record has been deleted successfully!");
        }
        catch(Exception e)
        {
                e.printStackTrace();
        }
        }
public static void main(String[] args)
{
        CRUDOperations  crud=new  CRUDOperations();
        crud.createEmployee("CP verma", "cpverma2020@yahoo.com",975463729L, "CSE", 90000);
        crud.createEmployee("Bidit", "biditvats@gmail.com", 9916712669L, "CSE", 40000);
        crud.getEmployee(1);
        crud.updateEmployee(3, 8892550034L, 50000);
        crud.deleteEmployee(2);
}
}
```