JDBC

JDBC- Java Database Connectivity

JDBC is an API (Application Programming Interface) in Java that allows Java applications to interact with databases like MySQL, Oracle, PostgreSQL, etc. It provides a set of classes and interfaces to connect, execute queries, and retrieve data from a database.

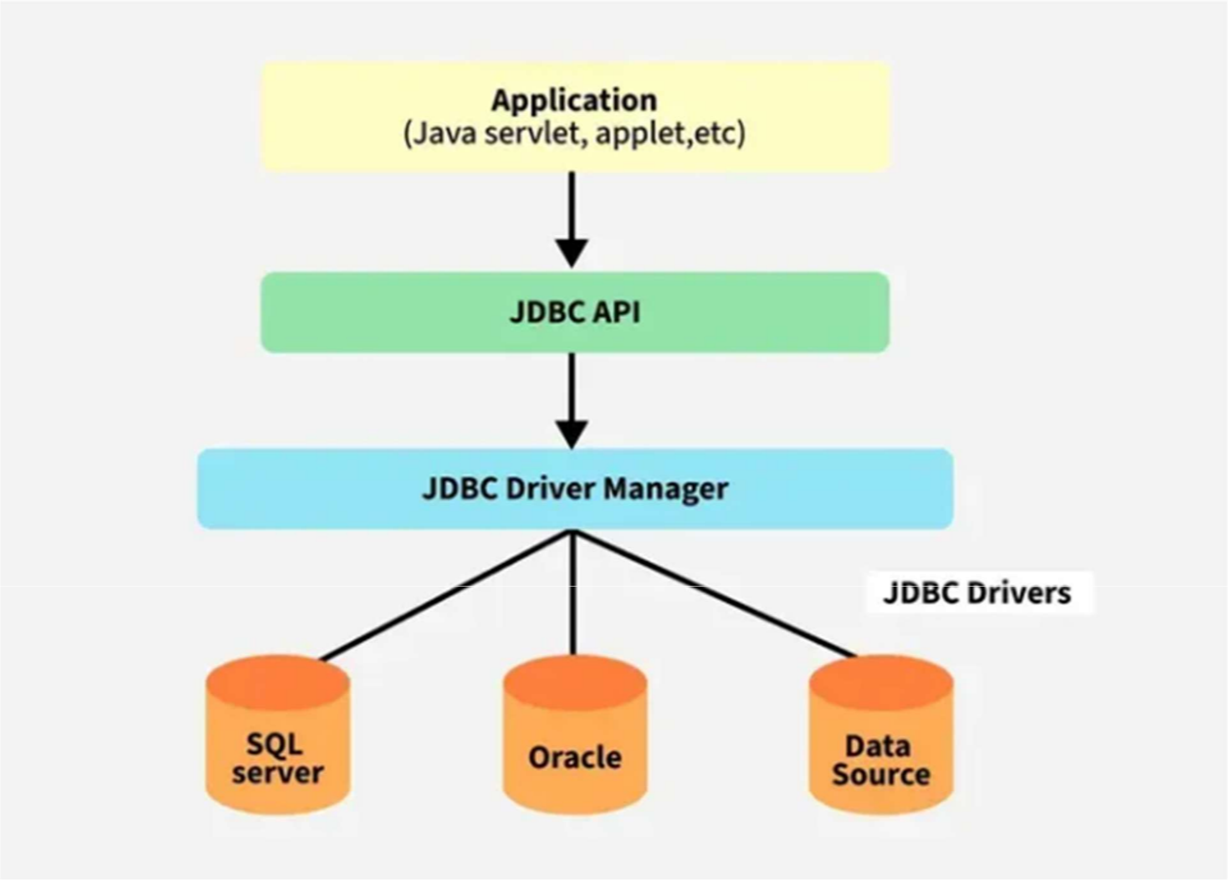
(JDBC)
JAVA APPLICATION
DATABASE

Key Features of JDBC:

1. **Connectivity:** Establishes a connection between Java applications and databases.
2. **Query Execution:** Allows execution of SQL queries (SELECT, INSERT, UPDATE, DELETE).
3. **Data Retrieval:** Retrieves and processes data from databases.
4. **Database Independence:** Works with different databases using different JDBC drivers.
5. **Transaction Management:** Supports transactions (commit, rollback).

Architecture of JDBC

JDBC Architecture



Components of JDBC Architecture:

1. **JDBC API:**
   - Provides methods to connect to a database, execute queries, and handle results.
   - Key Interfaces: Connection, Statement, PreparedStatement, ResultSet, DriverManager.
2. **JDBC Driver Manager:**
   - Manages different types of JDBC drivers and establishes database connections.
   - Loads the appropriate driver based on the connection request.

3. **JDBC Drivers:**
    - Enables Java applications to communicate with different databases.
    - Types of JDBC Drivers:
        - **Type-1:** JDBC-ODBC Bridge Driver (Deprecated)
        - **Type-2:** Native-API Driver (Database vendor-specific)
        - **Type-3:** Network Protocol Driver (Middleware-based)
        - **Type-4:** Thin Driver (Direct database communication, most commonly used)
4. **Database:**
    - Stores the actual data in a structured format using tables.
    - Supports SQL queries for data manipulation.
5. **Application:** It is a Java applet or a servlet that communicates with a data source.

JDBC Architecture: Two-Tier and Three-Tier Models

JDBC architecture follows **two-tier and three-tier processing models** to access a database efficiently.

1. Two-Tier Architecture (Client-Server Model)

In this model, the **Java application directly communicates with the database** using a JDBC driver. Queries are sent, and results are returned without any intermediate processing.

JDBC (Java Database Connectivity) follows a **two-layer architecture**, which includes:

1. **JDBC API Layer** – The interface between Java applications and JDBC drivers.

2. **JDBC Driver Layer** – The communication bridge between the Java application and the database.

Structure:

Client Application (Java) → JDBC Driver → Database

✅ **Pros:**

- Simple and easy to implement.
- Faster communication since there is no intermediate layer.

❌ **Cons:**

- Less secure, as the database is directly exposed to the client.
- Not scalable for large applications.

2. Three-Tier Architecture (Client-Middleware-Database)

In this model, queries from the client application go through a **middleware (Application Server)** before reaching the database. The middleware processes the request, interacts with the database, and sends the results back to the client.

Structure:

Client Application → Application Server → JDBC Driver → Database

✅ **Pros:**

- **More secure** (database is hidden behind an application server).
- **Scalable** (handles multiple clients efficiently).
- **Supports business logic** processing before sending data to the client.

❌ **Cons:**

- Slightly slower due to an additional layer.
- More complex implementation.

Key Points:

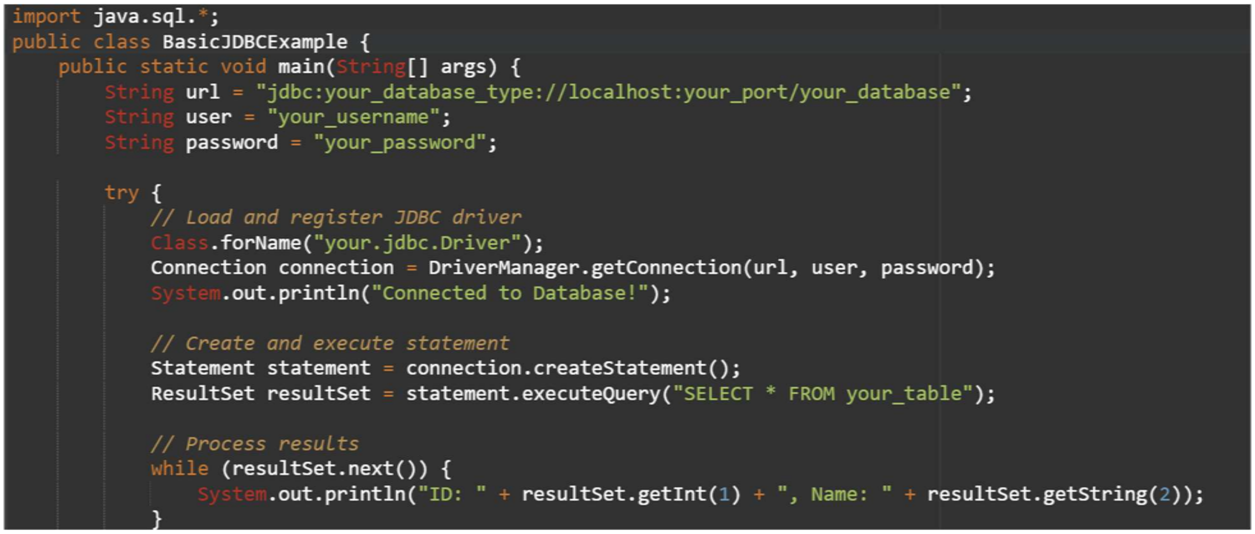
- **Two-Tier Architecture** is ideal for small applications requiring direct database access.
- **Three-Tier Architecture** is preferred for large, enterprise-level applications requiring **security, scalability, and business logic processing.**

Steps to connect Java application with a Database using JDBC

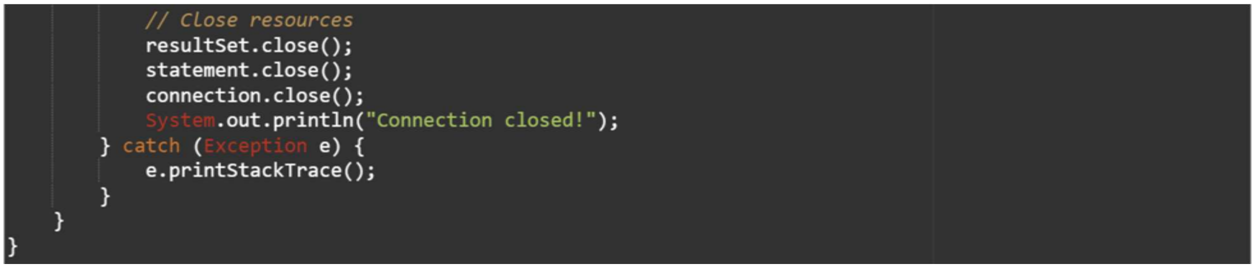
1) Import Packages
2) Load Driver
3) Register Driver
4) Create Connection
5) Create Statement
6) Execute Statement
7) Close

After JDBC 4.0 i.e. introduced in java 6 jdbc registration and loading is not compulsory.

SAMPLE CODE: For interaction with any database

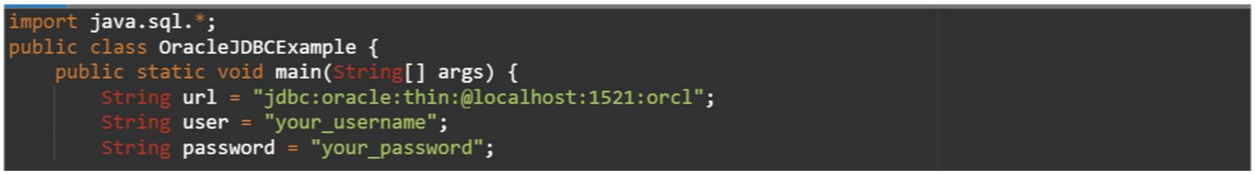


```java
import java.sql.*;
public class BasicJDBCExample {
    public static void main(String[] args) {
        String url = "jdbc:your_database_type://localhost:your_port/your_database";
        String user = "your_username";
        String password = "your_password";

        try {
            // Load and register JDBC driver
            Class.forName("your.jdbc.Driver");
            Connection connection = DriverManager.getConnection(url, user, password);
            System.out.println("Connected to Database!");

            // Create and execute statement
            Statement statement = connection.createStatement();
            ResultSet resultSet = statement.executeQuery("SELECT * FROM your_table");

            // Process results
            while (resultSet.next()) {
                System.out.println("ID: " + resultSet.getInt(1) + ", Name: " + resultSet.getString(2));
            }
```

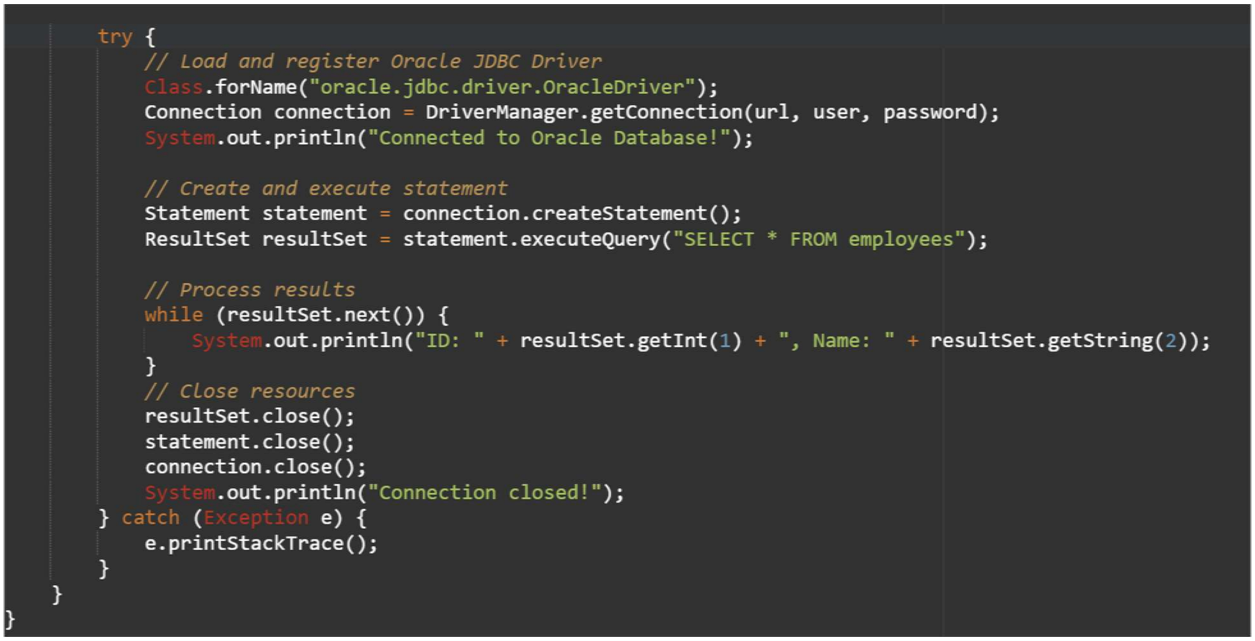


```
            // Close resources
            resultSet.close();
            statement.close();
            connection.close();
            System.out.println("Connection closed!");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

CODE: For interaction with oracle(sql) database



```
import java.sql.*;
public class OracleJDBCExample {
    public static void main(String[] args) {
        String url = "jdbc:oracle:thin:@localhost:1521:orcl";
        String user = "your_username";
        String password = "your_password";
```



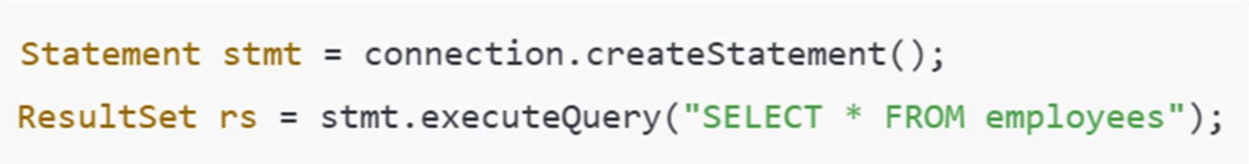
```
        try {
            // Load and register Oracle JDBC Driver
            Class.forName("oracle.jdbc.driver.OracleDriver");
            Connection connection = DriverManager.getConnection(url, user, password);
            System.out.println("Connected to Oracle Database!");

            // Create and execute statement
            Statement statement = connection.createStatement();
            ResultSet resultSet = statement.executeQuery("SELECT * FROM employees");

            // Process results
            while (resultSet.next()) {
                System.out.println("ID: " + resultSet.getInt(1) + ", Name: " + resultSet.getString(2));
            }
            // Close resources
            resultSet.close();
            statement.close();
            connection.close();
            System.out.println("Connection closed!");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

When to use simple Statement & when to use preparedStatement

Use Statement when:

1. **Static Queries** – If the SQL query is fixed and doesn't change dynamically.
2. **Simple Execution** – For one-time query execution without parameters.

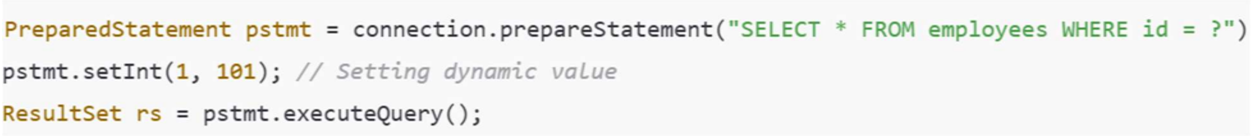
3. **DDL Queries** – Useful for CREATE, DROP, or ALTER statements.
4. **Performance is not a concern** – Suitable when query execution happens infrequently.
5. **No User Input** – Use Statement when queries don't involve user inputs to avoid SQL injection risks.



```
Statement stmt = connection.createStatement();
ResultSet rs = stmt.executeQuery("SELECT * FROM employees");
```

Use PreparedStatement when:

1. **Parameterized Queries** – When the query has variables that change dynamically (e.g., ? placeholders).
2. **Preventing SQL Injection** – Prevents malicious injections by automatically handling special characters.
3. **Performance Optimization** – Compiled once, executed multiple times efficiently.
4. **Batch Execution** – Supports executing multiple queries in a batch for efficiency.
5. **Frequent Query Execution** – If the same query is run multiple times with different values.



```
PreparedStatement pstmt = connection.prepareStatement("SELECT * FROM employees WHERE id = ?")
pstmt.setInt(1, 101); // Setting dynamic value
ResultSet rs = pstmt.executeQuery();
```

Here are some **important methods** used in **JDBC** and **SQL**:

---

1. JDBC Important Methods

a) Connection Interface (Managing Database Connection)

Method	Description
createStatement()	Creates a simple SQL statement.
prepareStatement(String sql)	Creates a precompiled SQL statement.
commit()	Commits a transaction.
rollback()	Rolls back a transaction.
setAutoCommit(boolean status)	Enables/disables auto-commit mode.
close()	Closes the database connection.

b) Statement Interface (Executing SQL)

Method	Description
executeQuery(String sql)	Executes SELECT queries and returns a ResultSet.
executeUpdate(String sql)	Executes INSERT, UPDATE, DELETE queries and returns affected row count.
execute(String sql)	Can execute both SELECT and DML statements.
close()	Closes the statement object.

c) PreparedStatement Interface (For Parameterized Queries)

Method	Description
setInt(int index, int value)	Sets an integer parameter.
setString(int index, String value)	Sets a string parameter.

Method	Description
setDouble(int index, double value)	Sets a double parameter.
executeQuery()	Executes a SELECT query.
executeUpdate()	Executes an INSERT, UPDATE, DELETE query.

---

d) ResultSet Interface (Processing Query Results)

Method	Description
next()	Moves to the next row.
getInt(String columnLabel)	Retrieves an integer value.
getString(String columnLabel)	Retrieves a string value.
getDouble(String columnLabel)	Retrieves a double value.
close()	Closes the ResultSet.

---

2. SQL Important Methods

a) Data Manipulation Language (DML)

Method	Description
SELECT	Retrieves data from a table.
INSERT INTO	Inserts new data into a table.
UPDATE	Modifies existing data.

Method	**Description**
DELETE	Removes data from a table.

---

b) Data Definition Language (DDL)

Method	**Description**
CREATE TABLE	Creates a new table.
ALTER TABLE	Modifies an existing table.
DROP TABLE	Deletes a table.
TRUNCATE TABLE	Removes all records from a table without logging.

---

c) Data Control Language (DCL)

Method	**Description**
GRANT	Gives user permissions.
REVOKE	Removes user permissions.

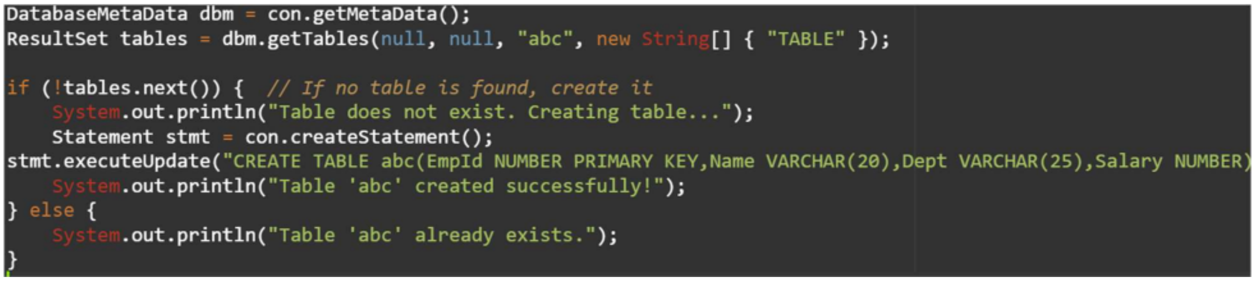
---

d) Transaction Control Language (TCL)

Method	**Description**
COMMIT	Saves all changes.
ROLLBACK	Reverts changes since the last COMMIT.
SAVEPOINT	Creates a rollback checkpoint.

These methods help efficiently interact with **JDBC** and **SQL** databases.

If table is already present then don’t create new table ,if not present then create



```
DatabaseMetaData dbm = con.getMetaData();
ResultSet tables = dbm.getTables(null, null, "abc", new String[] { "TABLE" });

if (!tables.next()) {  // If no table is found, create it
    System.out.println("Table does not exist. Creating table...");
    Statement stmt = con.createStatement();
stmt.executeUpdate("CREATE TABLE abc(EmpId NUMBER PRIMARY KEY,Name VARCHAR(20),Dept VARCHAR(25),Salary NUMBER)
    System.out.println("Table 'abc' created successfully!");
} else {
    System.out.println("Table 'abc' already exists.");
}
```

Example-

Problem Statement:

Write a java code that manages employee payroll information stored in an Oracle database.It has following features:

1. **Add** new employee records (employee ID, name, department, salary).
2. **View** all employee records.
3. **Update** an employee's salary.
4. **Delete** employee records

JdbcConnection.java

```
import java.sql.*;

public class JdbcConnection {

   public static Connection getConnection() {

      Connection con = null;

      try {

         // Load Oracle JDBC Driver

         Class.forName("oracle.jdbc.driver.OracleDriver");
         // Establish connection
         con =
DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:XE",
"system", "anupam");
         System.out.println("Database connected successfully!");
         // Check if table 'abc' exists
         DatabaseMetaData dbm = con.getMetaData();
         ResultSet tables = dbm.getTables(null, null, "ABC", new String[] {
"TABLE" });


         if (!tables.next()) { // Table does not exist
           String createTableQuery = "CREATE TABLE abc (EmpId NUMBER
PRIMARY KEY, Name VARCHAR(20), Dept VARCHAR(25), Salary NUMBER)";
           Statement stmt = con.createStatement();
           stmt.executeUpdate(createTableQuery);
           System.out.println("Table 'abc' created successfully!");
         } else {
           System.out.println("Table 'abc' already exists.");
         }
      } catch (Exception e) {
         System.out.println("Error: " + e.getMessage());
      }
      return con;
   }
```

```
}
```

MainOperation.java

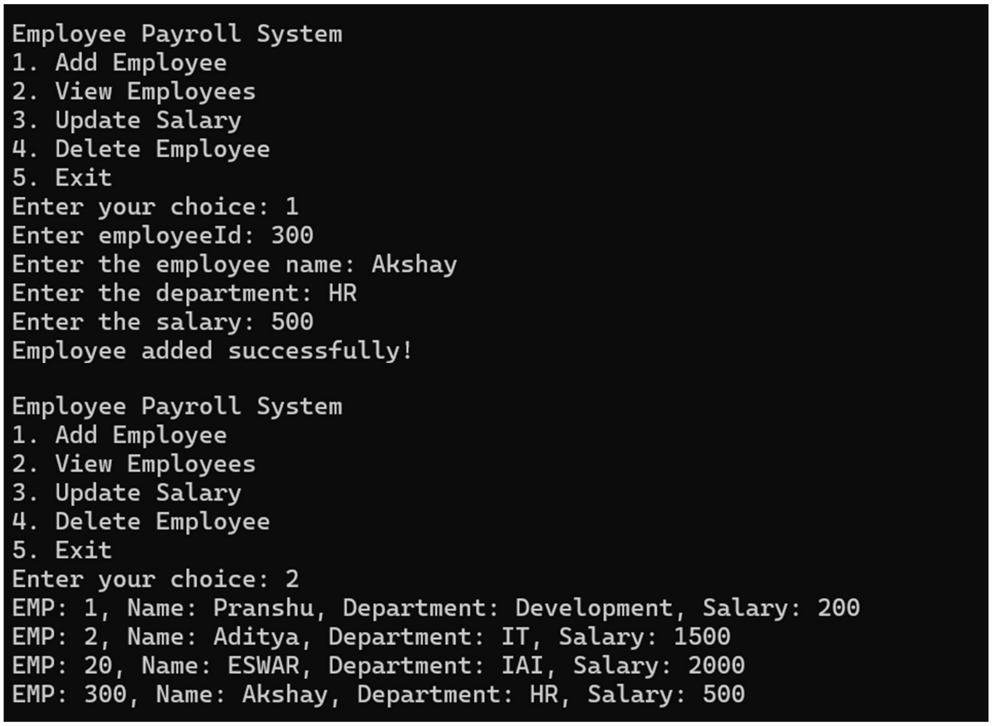
```
import java.sql.*;

import java.util.*;


public class MainOperation {

    static Scanner sc = new Scanner(System.in);

    static Connection con = JdbcConnection.getConnection();


    public static void addEmployee() {

        try {

            String s = "INSERT INTO abc VALUES(?,?,?,?)";

            PreparedStatement q = con.prepareStatement(s);


            System.out.print("Enter employeeId: ");

            int empId = sc.nextInt();

            sc.nextLine();


            System.out.print("Enter the employee name: ");

            String name = sc.nextLine();


            System.out.print("Enter the department: ");
            String dept = sc.nextLine();

            System.out.print("Enter the salary: ");
            int salary = sc.nextInt();

            q.setInt(1, empId);
            q.setString(2, name);
            q.setString(3, dept);
            q.setInt(4, salary);
            q.executeUpdate();

            System.out.println("Employee added successfully!");
        } catch (Exception e) {
            System.out.println("Error: " + e.getMessage());
        }
    }

    public static void viewEmployee() {
        try {
            String sql = "SELECT * FROM abc";
            Statement s = con.createStatement();
            ResultSet rs = s.executeQuery(sql);
```

```
            while (rs.next()) {
                System.out.println("EMP: " + rs.getInt(1) + ", Name: " + rs.getString(2) + ", Department: " + rs.getString(3) + ", Salary: " + rs.getInt(4));
            }
        } catch (Exception e) {
            System.out.println("Error: " + e.getMessage());
        }
    }

    public static void updateSalary() {
        try {
            System.out.print("Enter the employee name whose salary is to be updated: ");
            int empId = sc.next();

            System.out.print("Enter the new salary: ");
            int salary = sc.nextInt();

            String sql = "UPDATE abc SET Salary=? WHERE EmpId=?";
            PreparedStatement ps = con.prepareStatement(sql);
            ps.setInt(1, salary);
            ps.setInt(2, empId);
            ps.executeUpdate();
        int rowsAffected = ps.executeUpdate(); // Returns the number of rows updated


      if (rowsAffected > 0) {
        System.out.println("Employee salary updated successfully!");
      } else {
        System.out.println("Error: No employee found with ID " + empId);
      }     } catch (Exception e) {
        System.out.println("Error: " + e.getMessage());
      }
    }


    public static void deleteEmployee() {
      try {
        System.out.print("Enter 1 to delete complete table or 2 to delete by employee ID: ");
        int n = sc.nextInt();


        if (n == 1) {
          String sql = "DROP TABLE abc";
          Statement s = con.createStatement();
          s.execute(sql);
          System.out.println("Table deleted successfully!");
        } else {
```

```
      System.out.print("Enter the employee name to delete: ");
      int empId = sc.next();

      String sql = "DELETE FROM abc WHERE EmpId=?";
      PreparedStatement ps = con.prepareStatement(sql);
      ps.setInt(1, empId);
      ps.executeUpdate();

      System.out.println("Employee record deleted successfully!");
    }
  } catch (Exception e) {
    System.out.println("Error: " + e.getMessage());
  }
}

public static void main(String[] args) {
  while (true) {
    System.out.println("\nEmployee Payroll System");
    System.out.println("1. Add Employee");
    System.out.println("2. View Employees");
    System.out.println("3. Update Salary");
    System.out.println("4. Delete Employee");
    System.out.println("5. Exit");
    System.out.print("Enter your choice: ");

    int choice = sc.nextInt();
    switch (choice) {
      case 1:
        addEmployee();
        break;
      case 2:
        viewEmployees();
        break;
      case 3:
        updateSalary();
        break;
      case 4:
        deleteEmployee();
        break;
      case 5:
        System.out.println("Exiting...");
        return;
      default:
        System.out.println("Invalid choice. Try again.");
    }
  }
```

```
    }
}
```

Output



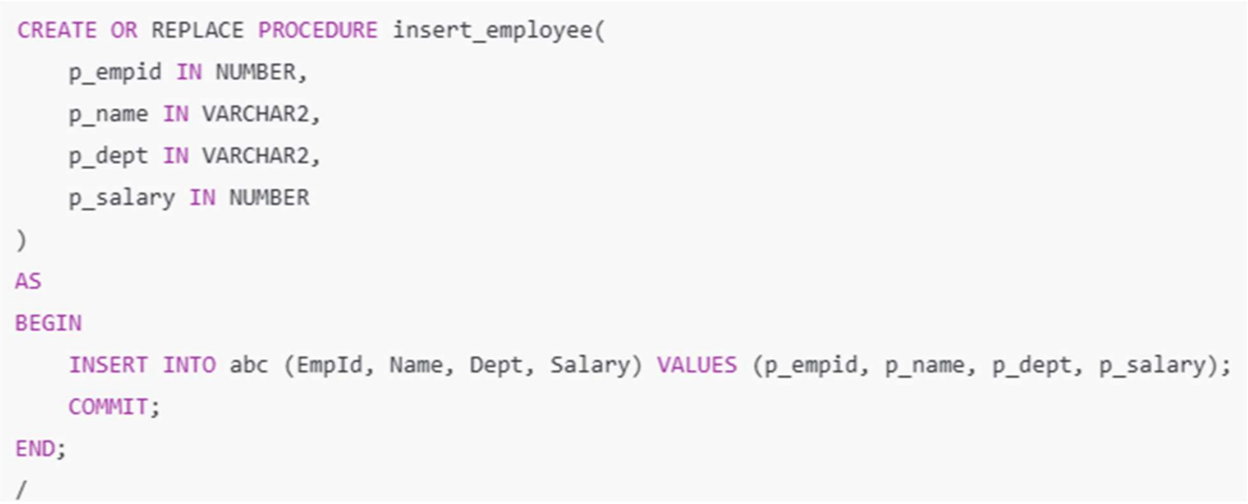
```
Employee Payroll System
1. Add Employee
2. View Employees
3. Update Salary
4. Delete Employee
5. Exit
Enter your choice: 1
Enter employeeId: 300
Enter the employee name: Akshay
Enter the department: HR
Enter the salary: 500
Employee added successfully!

Employee Payroll System
1. Add Employee
2. View Employees
3. Update Salary
4. Delete Employee
5. Exit
Enter your choice: 2
EMP: 1, Name: Pranshu, Department: Development, Salary: 200
EMP: 2, Name: Aditya, Department: IT, Salary: 1500
EMP: 20, Name: ESWAR, Department: IAI, Salary: 2000
EMP: 300, Name: Akshay, Department: HR, Salary: 500
```

Stored Procedure

A **Stored Procedure** is a precompiled SQL block that is stored in a database and can be executed multiple times with different parameters. It improves performance, security, and maintainability.

Basic Sample Example in Oracle SQL

Step 1: Create a Stored Procedure

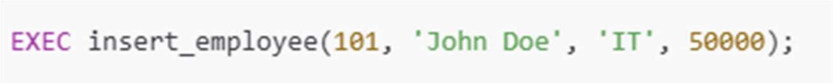


```
CREATE OR REPLACE PROCEDURE insert_employee(
    p_empid IN NUMBER,
    p_name IN VARCHAR2,
    p_dept IN VARCHAR2,
    p_salary IN NUMBER
)
AS
BEGIN
    INSERT INTO abc (EmpId, Name, Dept, Salary) VALUES (p_empid, p_name, p_dept, p_salary);
    COMMIT;
END;
/
```

◆ **Explanation:**

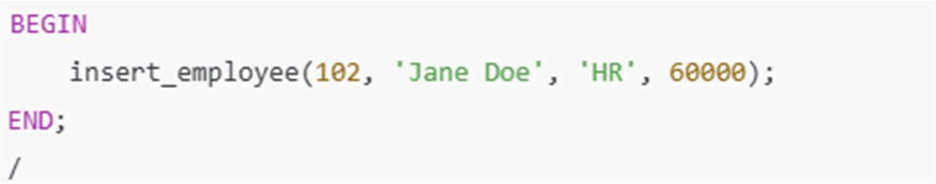
- The procedure **insert_employee** takes **4 input parameters** (EmpId, Name, Dept, Salary).
- It **inserts the values** into the abc table and **commits** the transaction.

Step 2: Call the Stored Procedure in SQL



```
EXEC insert_employee(101, 'John Doe', 'IT', 50000);
```

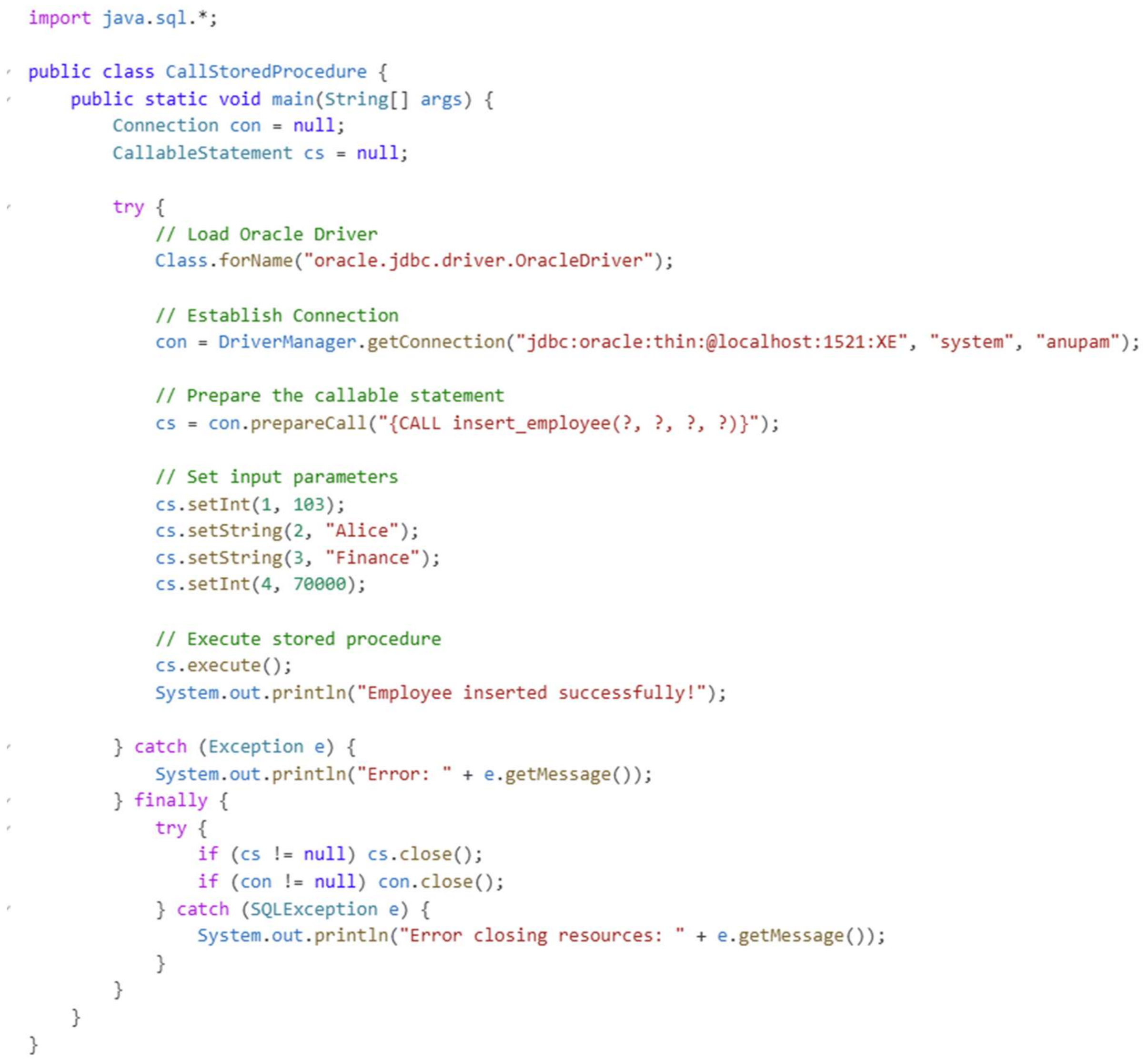
OR



```
BEGIN
    insert_employee(102, 'Jane Doe', 'HR', 60000);
END;
/
```

This will **insert records into the table**.

Step3: Calling a Stored Procedure in Java (JDBC)



```java
import java.sql.*;

public class CallStoredProcedure {
    public static void main(String[] args) {
        Connection con = null;
        CallableStatement cs = null;

        try {
            // Load Oracle Driver
            Class.forName("oracle.jdbc.driver.OracleDriver");

            // Establish Connection
            con = DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:XE", "system", "anupam");

            // Prepare the callable statement
            cs = con.prepareCall("{CALL insert_employee(?, ?, ?, ?)}");

            // Set input parameters
            cs.setInt(1, 103);
            cs.setString(2, "Alice");
            cs.setString(3, "Finance");
            cs.setInt(4, 70000);

            // Execute stored procedure
            cs.execute();
            System.out.println("Employee inserted successfully!");

        } catch (Exception e) {
            System.out.println("Error: " + e.getMessage());
        } finally {
            try {
                if (cs != null) cs.close();
                if (con != null) con.close();
            } catch (SQLException e) {
                System.out.println("Error closing resources: " + e.getMessage());
            }
        }
    }
}
```