

UNIT -3 JDBC

SONAWANE KHUSHBU K.

Introduction to JDBC

- **JDBC (Java Database Connectivity)** is a **Java API** that allows Java applications to **connect to databases, execute SQL queries, and retrieve results**. It provides a way for Java programs to interact with relational databases like MySQL, Oracle, PostgreSQL, and SQL Server.

Why Use JDBC?

- **Database Connectivity:** Enables Java applications to connect to databases and perform operations like **insert**, **update**, **delete**, and **retrieve data**.
- **Cross-Platform Compatibility:** Works with any relational database that has a JDBC driver.
- **Standardized API:** Provides a standard interface for interacting with databases, regardless of the database vendor.

JDBC Architecture

The JDBC architecture consists of the following components:

Component	Description
Java Application	The client program that wants to interact with the database.
JDBC API	Provides classes and interfaces to interact with databases.
JDBC Driver Manager	Manages database drivers and establishes connections.
JDBC Driver	A vendor-specific implementation that communicates with the database.
Database	The relational database to which the application connects.

Key JDBC Classes and Interfaces


Class/Interface	Description
<code>DriverManager</code>	Manages database drivers and establishes connections.
<code>Connection</code>	Represents the connection to the database.
<code>Statement</code>	Used to execute SQL queries.
<code>PreparedStatement</code>	A precompiled SQL statement that can accept parameters.
<code>CallableStatement</code>	Used to call stored procedures in a database.
<code>ResultSet</code>	Represents the result set returned by a SQL query.

Steps to Use JDBC in Java

Here are the basic steps to use JDBC in a Java program:

Step 1: Load the JDBC Driver

java

 Copy code

```
Class.forName("com.mysql.cj.jdbc.Driver");
```

In modern Java versions, the driver is **automatically loaded** when you include the JAR file.

Step 2: Establish a Connection

Use the **DriverManager** class to establish a connection to the database.

```
Connection conn = DriverManager.getConnection("jdbc:mysql://localhost:3306/mydb", "root",  
"password");
```

✓ Step 3: Create a Statement

Create a `Statement` or `PreparedStatement` object to execute SQL queries.


java

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

✓ Step 4: Execute SQL Queries

Use the `executeQuery()` method to retrieve data and `executeUpdate()` to update the database.

java

 Copy code

```
ResultSet rs = stmt.executeQuery("SELECT * FROM users");
```

✅ Step 5: Process the ResultSet

Retrieve the data from the `ResultSet` object.

java

```
while (rs.next()) {  
    System.out.println("Name: " + rs.getString("name"));  
}
```

✅ Step 6: Close the Connection

Always close the connection, statement, and result set to free resources.

java

```
rs.close();  
stmt.close();  
conn.close();
```


Types of JDBC Statements

Type	Description
Statement	Used to execute simple SQL queries without parameters.
PreparedStatement	Used to execute parameterized SQL queries.
CallableStatement	Used to execute stored procedures in the database.

Types of JDBC Drivers

There are **four types of JDBC drivers**, categorized based on how they interact with the database:

Type	Description	Example
Type 1: JDBC-ODBC Bridge Driver	Uses ODBC driver to connect to the database.	Deprecated
Type 2: Native-API Driver	Uses database-specific native APIs.	Oracle OCI driver
Type 3: Network Protocol Driver	Uses a middleware server to connect to the database.	IBM DB2 driver
Type 4: Thin Driver	Directly communicates with the database via the network.	MySQL, PostgreSQL

- ☐ **Advantages of JDBC:**
- **Platform-Independent** – Works across different operating systems.
- **Database-Independent** – Supports multiple databases through various drivers.
- **Standardized API** – Provides a consistent way to interact with databases.
- **Supports Dynamic SQL Queries** – Allows both static and dynamic query execution.

Common Exceptions in JDBC:

Exception	Description
<code>SQLException</code>	General exception for SQL-related issues.
<code>SQLSyntaxErrorException</code>	Thrown when there is a syntax error in the SQL query.
<code>SQLTimeoutException</code>	Thrown when a query takes too long to execute.
<code>SQLIntegrityConstraintViolationException</code>	Thrown when a primary/foreign key constraint is violated.

□ Java Database Connectivity (JDBC) in Java

- **What is Java Database Connectivity (JDBC)?**
- **Java Database Connectivity (JDBC)** is a **Java API** that enables Java programs to **connect to relational databases**, execute **SQL queries**, and **retrieve results**. It provides a way for Java applications to interact with databases such as **MySQL, Oracle, PostgreSQL, SQL Server**, etc.
- JDBC is a part of the **Java Standard Edition (Java SE)** and is included in the **java.sql** package.

□ Features of JDBC:

- **Platform-Independent:** JDBC allows Java applications to work with any database that has a JDBC driver.
- **Database-Independent:** Supports various relational databases like MySQL, Oracle, PostgreSQL, etc.
- **Secure and Reliable:** Ensures secure and reliable connections with databases.
- **Dynamic SQL Execution:** Supports both static and dynamic SQL queries.
- **Supports Transactions:** Enables handling of database transactions.

JDBC Components:

JDBC consists of several key components to facilitate communication between a Java application and a database.

Component	Description
DriverManager	Manages database drivers and establishes connections.
Connection	Represents the connection between a Java application and the database.
Statement	Used to execute SQL queries (SELECT, INSERT, UPDATE, DELETE).
PreparedStatement	Used to execute parameterized SQL queries for better security and performance.
CallableStatement	Used to call stored procedures in a database.
ResultSet	Represents the result set returned by a SELECT query.


Steps to Establish JDBC Connection:

Here are the basic steps to establish a JDBC connection and perform database operations:

Step 1: Load the JDBC Driver

The first step is to load the JDBC driver class.

java

 Copy code

```
Class.forName("com.mysql.cj.jdbc.Driver");
```

Note: In newer versions of Java, the driver is automatically loaded when the application starts, so this step can be skipped.

✓ Step 2: Establish a Connection

The next step is to establish a connection to the database using `DriverManager`.

java

Copy code

```
Connection conn = DriverManager.getConnection("jdbc:mysql://localhost:3306/mydb", "root",
```

✓ Step 3: Create a Statement

After establishing a connection, you can create a `Statement` or `PreparedStatement` to execute SQL queries.

java

Copy code


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



✓ Step 4: Execute SQL Queries

You can now execute **SELECT**, **INSERT**, **UPDATE**, or **DELETE** queries using the `executeQuery()` or `executeUpdate()` methods.

java


 Copy code

```
ResultSet rs = stmt.executeQuery("SELECT * FROM users");
```


✓ Step 5: Process the Result Set

If you executed a **SELECT** query, process the `ResultSet` to retrieve the data.

java

 Copy code

```
while (rs.next()) {  
    int id = rs.getInt("id");  
    String name = rs.getString("name");  
    System.out.println("ID: " + id + ", Name: " + name);  
}
```



✓ Step 6: Close the Connection

Finally, close the **ResultSet**, **Statement**, and **Connection** objects to free up resources.

```
java
```

```
rs.close();  
stmt.close();  
conn.close();
```



Types of JDBC Statements

Statement Type	Description
<code>Statement</code>	Used to execute simple SQL queries without parameters.
<code>PreparedStatement</code>	Used to execute parameterized SQL queries .
<code>CallableStatement</code>	Used to call stored procedures .

Types of JDBC Drivers

There are **four types of JDBC drivers**, classified based on how they communicate with the database.

Type	Description	Example
Type 1	JDBC-ODBC Bridge Driver (Deprecated).	Not used anymore.
Type 2	Native-API Driver (platform-dependent).	Oracle OCI Driver.
Type 3	Network Protocol Driver (uses middleware).	IBM DB2 Driver.
Type 4	Thin Driver (pure Java driver that directly communicates with the database).	MySQL, PostgreSQL.

- ☐ **Advantages of JDBC**

- **Platform-Independent:** Works on any operating system.
- **Database-Independent:** Supports multiple relational databases.
- **Secure and Reliable:** Ensures secure connections with databases.
- **Supports Dynamic Queries:** Allows execution of dynamic and static SQL queries.
- **Handles Transactions:** Supports database transactions for data integrity.



Common JDBC Exceptions

Exception	Description
<code>SQLException</code>	General exception for SQL-related issues.
<code>SQLSyntaxErrorException</code>	Thrown when there is a syntax error in the SQL query.
<code>SQLTimeoutException</code>	Thrown when a query takes too long to execute.
<code>SQLIntegrityConstraintViolationException</code>	Thrown when a primary/foreign key constraint is violated.

- **Key Points**

- **Performance:** PreparedStatement is generally more efficient for repeated execution because the SQL statement is compiled once.
- **Security:** PreparedStatement is safer against SQL injection due to parameterized queries.
- **Flexibility:** If your queries require dynamic values (like user input), PreparedStatement is the preferred choice.
- **Complexity:** For very simple queries that won't be repeated or parameterized, Statement might be simpler and sufficient.

what is Statement Object, PreparedStatement object and callable statement object

- In Java, **Statement**, **PreparedStatement**, and **CallableStatement** are three different interfaces in the java.sql package used to execute SQL queries with a relational database via JDBC (Java Database Connectivity). Each has unique features suited to specific use cases.

1. Statement Object


- The **Statement** object is used to execute **static SQL queries** at runtime. It is suitable for simple SQL queries where you don't need to pass any parameters dynamically.
- **Key Features:**
 - Executes static SQL queries.
 - Doesn't accept input parameters.
 - Susceptible to **SQL injection attacks**.
 - Slower compared to PreparedStatement for repeated queries because the SQL query is compiled each time.

Common Methods:

Method	Description
<code>executeQuery()</code>	Executes SELECT queries and returns a <code>ResultSet</code> .
<code>executeUpdate()</code>	Executes INSERT, UPDATE, or DELETE queries and returns the number of affected rows.
<code>execute()</code>	Executes any SQL statement (returns boolean).

Example:

java

 Copy code

```
Statement stmt = conn.createStatement();  
String sql = "INSERT INTO students (id, name, age) VALUES (1, 'John Doe', 22)";  
stmt.executeUpdate(sql);
```

CRUD operations with Statement Object

- In Java, **CRUD operations (Create, Read, Update, Delete)** are fundamental when interacting with relational databases through JDBC (Java Database Connectivity). The Statement object in JDBC is a simple way to execute SQL queries directly on the database.
- Let's break down how to perform CRUD operations using the Statement object in Java:

1. Prerequisites: JDBC Setup

Before performing CRUD operations, ensure you have:

- Database connection URL, username, and password
- JDBC Driver dependency (e.g., MySQL, PostgreSQL)

java


```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.Statement;
import java.sql.ResultSet;

public class JDBCExample {
    // Database URL, Username, and Password
    static final String DB_URL = "jdbc:mysql://localhost:3306/mydatabase";
    static final String USER = "root";
    static final String PASS = "password";
}
```

2. Create Operation (INSERT)

To insert data into a table:

java


 Copy code

```
public static void insertRecord() {  
    try (Connection conn = DriverManager.getConnection(DB_URL, USER, PASS);  
        Statement stmt = conn.createStatement()) {  
  
        String sql = "INSERT INTO students (id, name, age) VALUES (1, 'John Doe', 22)";  
        stmt.executeUpdate(sql);  
        System.out.println("Record inserted successfully.");  
  
    } catch (Exception e) {  
        e.printStackTrace();  
    }  
}
```


3. Read Operation (SELECT)

To fetch data from the table:

java

 Copy code


```
public static void readRecords() {  
    try (Connection conn = DriverManager.getConnection(DB_URL, USER, PASS);  
        Statement stmt = conn.createStatement()) {  
  
        String sql = "SELECT id, name, age FROM students";  
        ResultSet rs = stmt.executeQuery(sql);  
  
        while (rs.next()) {  
            int id = rs.getInt("id");  
            String name = rs.getString("name");  
            int age = rs.getInt("age");  
            System.out.println("ID: " + id + ", Name: " + name + ", Age: " + age);  
        }  
  
    } catch (Exception e) {  
        e.printStackTrace();  
    }  
}
```



4. Update Operation (UPDATE)

To update existing records:

java


 Copy code

```
public static void updateRecord() {  
    try (Connection conn = DriverManager.getConnection(DB_URL, USER, PASS);  
        Statement stmt = conn.createStatement()) {  
  
        String sql = "UPDATE students SET age = 23 WHERE id = 1";  
        stmt.executeUpdate(sql);  
        System.out.println("Record updated successfully.");  
  
    } catch (Exception e) {  
        e.printStackTrace();  
    }  
}
```

5. Delete Operation (DELETE)

To delete a record from the table:

java

 Copy code

```
public static void deleteRecord() {  
    try (Connection conn = DriverManager.getConnection(DB_URL, USER, PASS);  
        Statement stmt = conn.createStatement()) {  
  
        String sql = "DELETE FROM students WHERE id = 1";  
        stmt.executeUpdate(sql);  
        System.out.println("Record deleted successfully.");  
  
    } catch (Exception e) {  
        e.printStackTrace();  
    }  
}
```


- **Key Points to Note:**
- **Database Connection:** Use `DriverManager.getConnection()` to establish a connection.
- **Statement Object:** Use `Statement` to execute SQL queries.
- **Error Handling:** Always wrap your code in a try-catch block to handle SQL exceptions.
- **Resource Management:** Use try-with-resources to automatically close connections, statements, and result sets.

2. PreparedStatement object


- The **PreparedStatement** object is used to execute **parameterized SQL queries**. It is more efficient and secure compared to Statement.
- **Key Features:**
- **Precompiled SQL queries**, which improves performance for repeated queries.
- Allows **parameter binding**, preventing SQL injection attacks.
- Can execute both **static** and **dynamic** queries.
- Supports **input parameters**.

Common Methods:

Method	Description
<code>setInt(index, value)</code>	Sets an integer parameter at the specified index.
<code>setString(index, value)</code>	Sets a string parameter at the specified index.
<code>executeQuery()</code>	Executes SELECT queries and returns a <code>ResultSet</code> .
<code>executeUpdate()</code>	Executes INSERT, UPDATE, or DELETE queries and returns the number of affected rows.

Example:

java

 Copy code

```
String sql = "INSERT INTO students (id, name, age) VALUES (?, ?, ?)";
PreparedStatement pstmt = conn.prepareStatement(sql);
pstmt.setInt(1, 1);
pstmt.setString(2, "John Doe");
pstmt.setInt(3, 22);
pstmt.executeUpdate();
```



- **Why Use PreparedStatement?**
- Prevents **SQL injection attacks**.
- Improves **performance** by pre-compiling SQL queries.
- Ideal for queries executed multiple times with different parameters.


CRUD operations with PreparedStatement object

- Here's a detailed guide to performing **CRUD operations (Create, Read, Update, Delete)** using the **PreparedStatement** object in Java. The PreparedStatement object is part of the JDBC API and is used to execute **parameterized SQL queries**, which makes it more secure and efficient than the Statement object.

Step 1: JDBC Setup (Database Connection)

Before performing CRUD operations, set up the **database connection** using `DriverManager`.

java

 Copy code

```
import java.sql.*;


public class JDBCPreparedStatementExample {
    static final String DB_URL = "jdbc:mysql://localhost:3306/mydatabase";
    static final String USER = "root";
    static final String PASS = "password";
}
```

Step 2: Create Operation (INSERT)

Here's how to use `PreparedStatement` to insert data into the database.

Code Example:

java

 Copy code

```
public static void insertRecord(int id, String name, int age) {  
    String sql = "INSERT INTO students (id, name, age) VALUES (?, ?, ?)";  
  
    try (Connection conn = DriverManager.getConnection(DB_URL, USER, PASS);  
        PreparedStatement pstmt = conn.prepareStatement(sql)) {  
  
        pstmt.setInt(1, id);  
        pstmt.setString(2, name);  
        pstmt.setInt(3, age);  
  
        int rowsInserted = pstmt.executeUpdate();  
        if (rowsInserted > 0) {  
            System.out.println("A new record was inserted successfully!");  
        }  
    } catch (SQLException e) {  
        e.printStackTrace();  
    }  
}
```




Step 3: Read Operation (SELECT)


To retrieve data from the database, use `PreparedStatement` with the `executeQuery()` method.

Code Example:

java

 Copy code

```
public static void readRecords() {  
    String sql = "SELECT id, name, age FROM students";  
  
    try (Connection conn = DriverManager.getConnection(DB_URL, USER, PASS);  
        PreparedStatement pstmt = conn.prepareStatement(sql)) {  
  
        ResultSet rs = pstmt.executeQuery();  
  
        while (rs.next()) {  
            int id = rs.getInt("id");  
            String name = rs.getString("name");  
            int age = rs.getInt("age");  
  
            System.out.println("ID: " + id + ", Name: " + name + ", Age: " + age);  
        }  
  
    } catch (SQLException e) {  
        e.printStackTrace();  
    }  
}
```




Step 4: Update Operation (UPDATE)

To update an existing record, use the `executeUpdate()` method with a parameterized SQL query.

Code Example:

java

 Copy code

```
public static void updateRecord(int id, int newAge) {  
    String sql = "UPDATE students SET age = ? WHERE id = ?";  
  
    try (Connection conn = DriverManager.getConnection(DB_URL, USER, PASS);  
        PreparedStatement pstmt = conn.prepareStatement(sql)) {  
  
        pstmt.setInt(1, newAge);  
        pstmt.setInt(2, id);  
  
        int rowsUpdated = pstmt.executeUpdate();  
        if (rowsUpdated > 0) {  
            System.out.println("An existing record was updated successfully!");  
        }  
    } catch (SQLException e) {  
        e.printStackTrace();  
    }  
}
```




Step 5: Delete Operation (DELETE)

To delete a record, use the `executeUpdate()` method with a `DELETE` query.

Code Example:

java

 Copy code

```
public static void deleteRecord(int id) {  
    String sql = "DELETE FROM students WHERE id = ?";  
  
    try (Connection conn = DriverManager.getConnection(DB_URL, USER, PASS);  
        PreparedStatement pstmt = conn.prepareStatement(sql)) {  
  
        pstmt.setInt(1, id);  
  
        int rowsDeleted = pstmt.executeUpdate();  
        if (rowsDeleted > 0) {  
            System.out.println("A record was deleted successfully!");  
        }  
    } catch (SQLException e) {  
        e.printStackTrace();  
    }  
}
```



3. CallableStatement Object

- The **CallableStatement** object is used to **execute stored procedures** in the database. Stored procedures are precompiled SQL code that can be executed on the database server.
- **Key Features:**
 - Executes **stored procedures**.
 - Can handle **input** and **output parameters**.
 - Supports **complex queries and business logic** directly in the database.


Common Methods:

Method	Description
<code>registerOutParameter(index, type)</code>	Registers an output parameter of the given type.
<code>setInt(index, value)</code>	Sets an integer parameter at the specified index.
<code>setString(index, value)</code>	Sets a string parameter at the specified index.
<code>execute()</code>	Executes the stored procedure.
<code>getInt(index)</code>	Retrieves an integer output parameter.
<code>getString(index)</code>	Retrieves a string output parameter.

Example:

Assume a stored procedure named `getStudentName` exists in the database:


sql

 Copy code

```
CREATE PROCEDURE getStudentName(IN studentId INT, OUT studentName VARCHAR(50))
BEGIN
    SELECT name INTO studentName FROM students WHERE id = studentId;
END;
```

Java Code:

java

 Copy code

```
CallableStatement cstmt = conn.prepareCall("{CALL getStudentName(?, ?)}");
cstmt.setInt(1, 1); // Input parameter
cstmt.registerOutParameter(2, java.sql.Types.VARCHAR); // Output parameter
cstmt.execute();

String studentName = cstmt.getString(2);
System.out.println("Student Name: " + studentName);
```



- **Advantages of Using PreparedStatement:**

- ☐ **Prevents SQL Injection:** Since parameter values are set separately from the SQL query, it mitigates SQL injection risks.
- ☐ **Improves Performance:** SQL queries are precompiled, making it faster for repeated execution.
- ☐ **Simplifies Code:** Allows for dynamic parameter setting, making queries cleaner and easier to manage.


CRUD operations with callable statement object

- Here's a detailed guide on how to perform **CRUD operations (Create, Read, Update, Delete)** using the **CallableStatement** object in Java. The **CallableStatement** object is used to execute **stored procedures** in the database, which are precompiled SQL statements stored in the database server.

What is CallableStatement?

- `CallableStatement` is an interface in the **JDBC API** used to execute **stored procedures** in relational databases.
- Stored procedures can accept **input parameters**, return **output parameters**, and handle **complex business logic**.
- The syntax for calling a stored procedure is:

sql

 Copy code


```
{CALL procedure_name(?, ?, ?)}
```


- **CRUD Operations with CallableStatement**
- We'll perform the following CRUD operations using stored procedures:
- **Create (INSERT)**
- **Read (SELECT)**
- **Update (UPDATE)**
- **Delete (DELETE)**

Step 1: Create Stored Procedures in the Database

Stored Procedure for Insert (CREATE)

sql

 Copy code

```
DELIMITER //  
CREATE PROCEDURE insertStudent(IN id INT, IN name VARCHAR(50), IN age INT)  
BEGIN  
    INSERT INTO students (id, name, age) VALUES (id, name, age);  
END //  
DELIMITER ;
```

Stored Procedure for Read (SELECT)

sql

```
DELIMITER //  
CREATE PROCEDURE getStudents()  
BEGIN  
    SELECT * FROM students;  
END //  
DELIMITER ;
```

Stored Procedure for Update (UPDATE)

sql

```
DELIMITER //  
CREATE PROCEDURE updateStudentAge(IN id INT, IN newAge INT)  
BEGIN  
    UPDATE students SET age = newAge WHERE id = id;  
END //  
DELIMITER ;
```



Stored Procedure for Delete (DELETE)


sql

```
DELIMITER //  
CREATE PROCEDURE deleteStudent(IN id INT)  
BEGIN  
    DELETE FROM students WHERE id = id;  
END //  
DELIMITER ;
```

Step 2: Java Code for CRUD Operations with CallableStatement

JDBC Setup (Database Connection)

java

 Copy code

```
import java.sql.*;


public class CallableStatementExample {
    static final String DB_URL = "jdbc:mysql://localhost:3306/mydatabase";
    static final String USER = "root";
    static final String PASS = "password";

    public static void main(String[] args) {
        insertStudent(1, "John Doe", 22);
        getStudents();
        updateStudentAge(1, 23);
        deleteStudent(1);
    }
}
```



1. Create Operation (INSERT)

java


 Copy code

```
public static void insertStudent(int id, String name, int age) {  
    String sql = "{CALL insertStudent(?, ?, ?)}";  
  
    try (Connection conn = DriverManager.getConnection(DB_URL, USER, PASS);  
        CallableStatement cstmt = conn.prepareCall(sql)) {  
  
        cstmt.setInt(1, id);  
        cstmt.setString(2, name);  
        cstmt.setInt(3, age);  
  
        cstmt.execute();  
        System.out.println("Student inserted successfully.");  
  
    } catch (SQLException e) {  
        e.printStackTrace();  
    }  
}
```




2. Read Operation (SELECT)

java


 Copy

```
public static void getStudents() {  
    String sql = "{CALL getStudents()}";  
  
    try (Connection conn = DriverManager.getConnection(DB_URL, USER, PASS);  
        CallableStatement cstmt = conn.prepareCall(sql)) {  
  
        ResultSet rs = cstmt.executeQuery();  
  
        while (rs.next()) {  
            int id = rs.getInt("id");  
            String name = rs.getString("name");  
            int age = rs.getInt("age");  
            System.out.println("ID: " + id + ", Name: " + name + ", Age: " + age);  
        }  
  
    } catch (SQLException e) {  
        e.printStackTrace();  
    }  
}
```



3. Update Operation (UPDATE)

java


 Copy code

```
public static void updateStudentAge(int id, int newAge) {  
    String sql = "{CALL updateStudentAge(?, ?)}";  
  
    try (Connection conn = DriverManager.getConnection(DB_URL, USER, PASS);  
        CallableStatement cstmt = conn.prepareCall(sql)) {  
  
        cstmt.setInt(1, id);  
        cstmt.setInt(2, newAge);  
  
        cstmt.execute();  
        System.out.println("Student's age updated successfully.");  
  
    } catch (SQLException e) {  
        e.printStackTrace();  
    }  
}
```



4. Delete Operation (DELETE)

java

 Copy code

```
public static void deleteStudent(int id) {  
    String sql = "{CALL deleteStudent(?)}";  
  
    try (Connection conn = DriverManager.getConnection(DB_URL, USER, PASS);  
        CallableStatement cstmt = conn.prepareCall(sql)) {  
  
        cstmt.setInt(1, id);  
  
        cstmt.execute();  
        System.out.println("Student deleted successfully.");  
  
    } catch (SQLException e) {  
        e.printStackTrace();  
    }  
}
```

- **Advantages of Using CallableStatement:**

- ☐ **Reusable and Precompiled Queries:** Stored procedures are precompiled, improving performance.
- ☐ **Security:** Prevents SQL injection attacks.
- ☐ **Complex Logic:** Supports business logic that runs directly on the database server.
- ☐ **Input and Output Parameters:** Can handle input and output parameters in a single call.


The ResultSet Object

- **What is the ResultSet Object in Java?**
- In Java's **JDBC API**, the **ResultSet** object represents the **result set** of a **SQL query** executed using a **Statement**, **PreparedStatement**, or **CallableStatement**. It provides a way to **read data** from a relational database **row by row**.
- The **ResultSet** acts like a **cursor** that moves through the data returned by a query.

You can get a `ResultSet` object by executing **SELECT** queries using the following JDBC objects:

1. `Statement` :


java

 Copy code

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

2. `PreparedStatement` :


java

 Copy code

```
PreparedStatement pstmt = conn.prepareStatement("SELECT * FROM students WHERE id = ?")  
pstmt.setInt(1, 1);  
ResultSet rs = pstmt.executeQuery();
```

3. `CallableStatement` :

java

 Copy code

```
CallableStatement cstmt = conn.prepareCall("{CALL getStudents()}");  
ResultSet rs = cstmt.executeQuery();
```



Navigating Through the ResultSet

The `ResultSet` object provides several methods to navigate through the result set.

Method	Description
<code>next()</code>	Moves the cursor to the next row.
<code>previous()</code>	Moves the cursor to the previous row (if supported).
<code>first()</code>	Moves the cursor to the first row (if supported).
<code>last()</code>	Moves the cursor to the last row (if supported).
<code>absolute(int row)</code>	Moves the cursor to the specified row number.
<code>relative(int rows)</code>	Moves the cursor forward or backward by a specified number of rows.

- **EXAMPLE:**

- `import java.sql.*;`

- `public class ResultSetExample {`

- `static final String DB_URL = "jdbc:mysql://localhost:3306/mydatabase";`

- `static final String USER = "root";`

- `static final String PASS = "password";`

- `public static void main(String[] args) {`
- `try {`
- `Connection conn = DriverManager.getConnection(DB_URL, USER,`
`PASS);`
- `Statement stmt = conn.createStatement();`
- `ResultSet rs = stmt.executeQuery("SELECT * FROM students");`
- `while (rs.next()) {`
- `int id = rs.getInt("id");`
- `String name = rs.getString("name");`
- `int age = rs.getInt("age");`
- `System.out.println("ID: " + id + ", Name: " + name + ", Age: " + age);`
- `}`

- `rs.close();`
- `stmt.close();`
- `conn.close();`
- `} catch (SQLException e) {`
- `e.printStackTrace();`
- `}`
- `}`
- `}`


Types of ResultSet

There are three types of `ResultSet` based on **navigational capability** and **updatability**.

Type	Description
<code>TYPE_FORWARD_ONLY</code>	The cursor can only move forward. (Default)
<code>TYPE_SCROLL_INSENSITIVE</code>	The cursor can scroll backward and forward. The result set is not sensitive to database changes.
<code>TYPE_SCROLL_SENSITIVE</code>	The cursor can scroll backward and forward. The result set is sensitive to database changes.

Example of Scrollable ResultSet:

java

 Copy code

```
Statement stmt = conn.createStatement(
    ResultSet.TYPE_SCROLL_INSENSITIVE,
    ResultSet.CONCUR_READ_ONLY
);
ResultSet rs = stmt.executeQuery("SELECT * FROM students");

// Move to the last row
rs.last();
System.out.println("Total rows: " + rs.getRow());

// Move to the first row
rs.first();
System.out.println("First row: " + rs.getString("name"));
```


Updatable ResultSet

By default, `ResultSet` is read-only. To make it **updatable**, use the following constants:

Concurrency Mode	Description
<code>CONCUR_READ_ONLY</code>	Default mode, read-only.
<code>CONCUR_UPDATABLE</code>	Allows updates to the <code>ResultSet</code> .

Example of Updatable ResultSet:

java

 Cop

```
Statement stmt = conn.createStatement(  
    ResultSet.TYPE_SCROLL_INSENSITIVE,  
    ResultSet.CONCUR_UPDATABLE  
);  
ResultSet rs = stmt.executeQuery("SELECT * FROM students");  
  
// Update the age of the first student  
rs.first();  
rs.updateInt("age", 25);  
rs.updateRow();  
System.out.println("Age updated successfully.");
```

Handling Null Values in ResultSet

To check for null values in a column, use the `wasNull()` method.

java

```
int age = rs.getInt("age");  
if (rs.wasNull()) {  
    System.out.println("Age is null.");  
}
```

Closing the ResultSet

Always close the `ResultSet` object after use to free up resources.

```
java
```

```
rs.close();
```


Advantages of Using ResultSet:

- ✓ **Efficient Data Retrieval:** Allows row-by-row processing of query results.
- ✓ **Dynamic Navigation:** Can move the cursor in different directions based on the type.
- ✓ **Updatable ResultSet:** Supports modifying data directly within the `ResultSet`.

Summary of CRUD Operations Using ResultSet:

Operation	Method Used	Description
Create	<code>rs.moveToInsertRow()</code>	Insert a new row into the ResultSet.
Read	<code>rs.next()</code> , <code>rs.getXXX()</code>	Retrieve and navigate through rows.
Update	<code>rs.updateXXX()</code> , <code>rs.updateRow()</code>	Modify the current row.
Delete	<code>rs.deleteRow()</code>	Delete the current row.

what is different between Statement Object, PreparedStatement object in java

Aspect	Statement	PreparedStatement
Definition	Used to execute static SQL queries.	Used to execute precompiled and parameterized SQL queries.
Query Structure	The SQL query is provided directly as a string.	The SQL query is precompiled and can include placeholders (?) for parameters.
Parameter Handling	Cannot handle parameters; values must be directly included in the query string.	Allows parameters to be set using methods like <code>setInt()</code> , <code>setString()</code> , etc.
Security	More vulnerable to SQL injection attacks as user input is directly concatenated into the query string.	More secure against SQL injection because the query and parameters are handled separately.
Performance	Queries are parsed and compiled every time they are executed.	Queries are precompiled once and can be reused, making them faster for repeated executions.
Use Case	Suitable for simple or one-time queries. 	Ideal for queries executed multiple times or with varying parameters.

Performance	Queries are parsed and compiled every time they are executed.	Queries are precompiled once and can be reused, making them faster for repeated executions.
Use Case	Suitable for simple or one-time queries.	Ideal for queries executed multiple times or with varying parameters.
Batch Execution	Supports batch execution using <code>addBatch()</code> .	Supports batch execution with parameters, making it more efficient for batch processing.
Example Code	<pre>java Statement stmt = conn.createStatement(); ResultSet rs = stmt.executeQuery("SELECT * FROM users");</pre>	<pre>java PreparedStatement pstmt = conn.prepareStatement("SELECT * FROM users WHERE id = ?"); pstmt.setInt(1, 10); ResultSet rs = pstmt.executeQuery();</pre>

Summary Comparison:

Feature	Statement	PreparedStatement	CallableStatement
Use Case	Static SQL queries	Parameterized SQL queries	Stored procedures and functions
Security	Vulnerable to SQL Injection	Safe from SQL Injection	Safe from SQL Injection
Efficiency	Low for repeated queries	High, reuses compiled query	High for complex operations
Example CRUD	Simple <code>SELECT</code> , <code>UPDATE</code>	<code>INSERT</code> , <code>UPDATE</code> , <code>DELETE</code> with params	Executes stored procedures

THE END

Sonawane khushbu k.