东方瑞通 终身学习 Founded in 1998

JDBC

IT管理方向

办公应用 与协作

软件开发

数据库

运维方向

通用技能

云计算 大数据

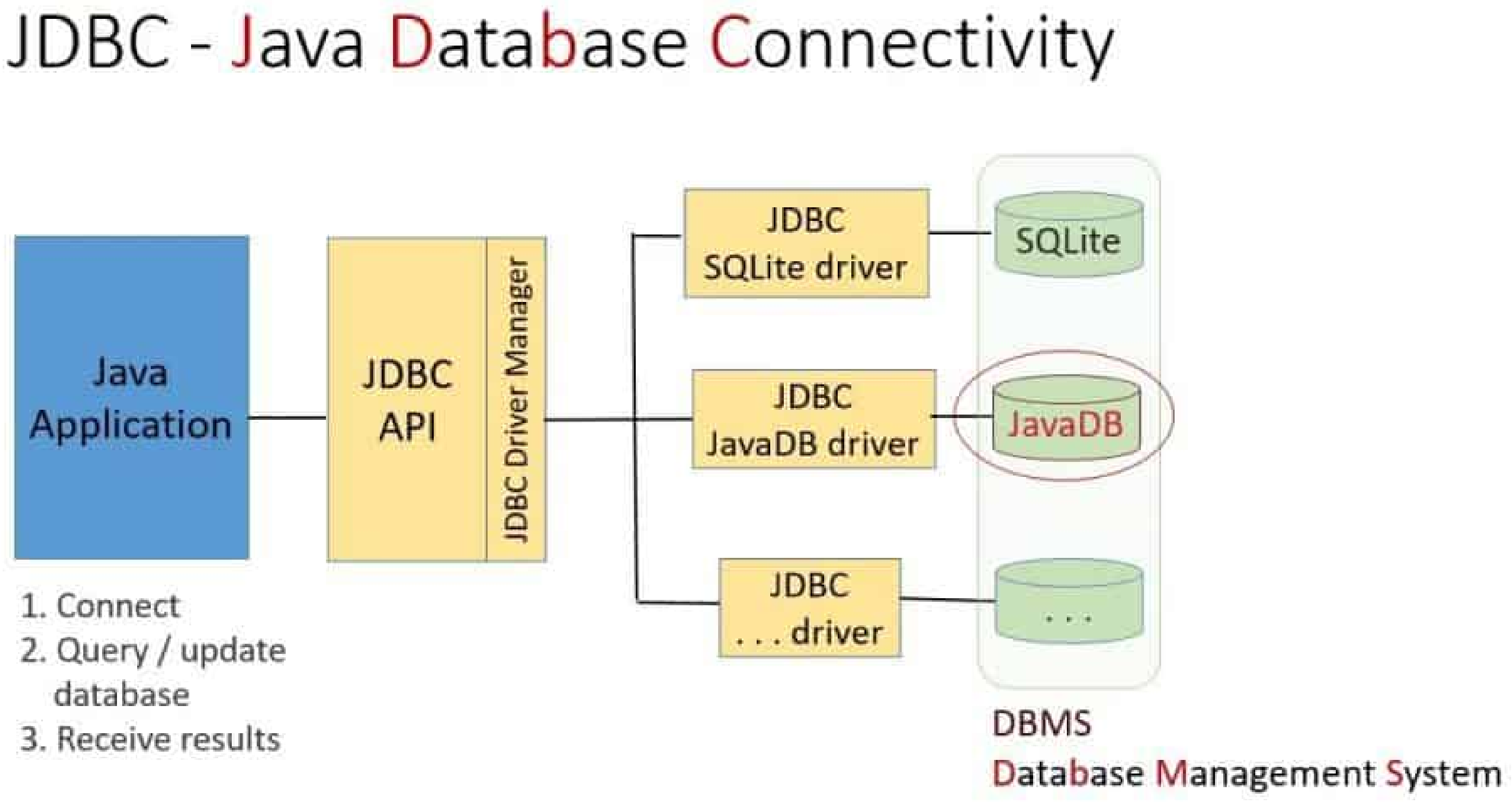
ERP工程师

云大物移智

关注微信公众号
享终身免费学习

1. Introduction

JDBC API is a Java API that can access any kind of tabular data, especially data stored in a Relational Database. JDBC works with Java on a variety of platforms, such as Windows, Mac OS, and the various versions of UNIX.



2. Why to Learn JDBC?

JDBC stands for **J**ava **D**ata**b**ase **C**onnectivity, which is a standard Java API for database-independent connectivity between the Java programming language and a wide range of databases.

The JDBC library includes APIs for each of the tasks mentioned below that are commonly associated with database usage.

- Making a connection to a database.
- Creating SQL or MySQL statements.
- Executing SQL or MySQL queries in the database.
- Viewing & Modifying the resulting records.

3. Applications of JDBC

Fundamentally, JDBC is a specification that provides a complete set of interfaces that allows for portable access to an underlying database. Java can be used to write different types of executables, such as –

- Java Applications
- Java Applets
- Java Servlets
- Java ServerPages (JSPs)
- Enterprise JavaBeans (EJBs).

All of these different executables are able to use a JDBC driver to access a database, and take advantage of the stored data.

JDBC provides the same capabilities as ODBC, allowing Java programs to contain database-independent code.

4. The JDBC 4.0 Packages

The java.sql and javax.sql are the primary packages for JDBC 4.0. This is the latest JDBC version at the time of writing this tutorial. It offers the main classes for interacting with your data sources.

The new features in these packages include changes in the following areas –

- Automatic database driver loading.
- Exception handling improvements.
- Enhanced BLOB/CLOB functionality.
- Connection and statement interface enhancements.
- National character set support.
- SQL ROWID access.
- SQL 2003 XML data type support.
- Annotations.

5. JDBC Architecture

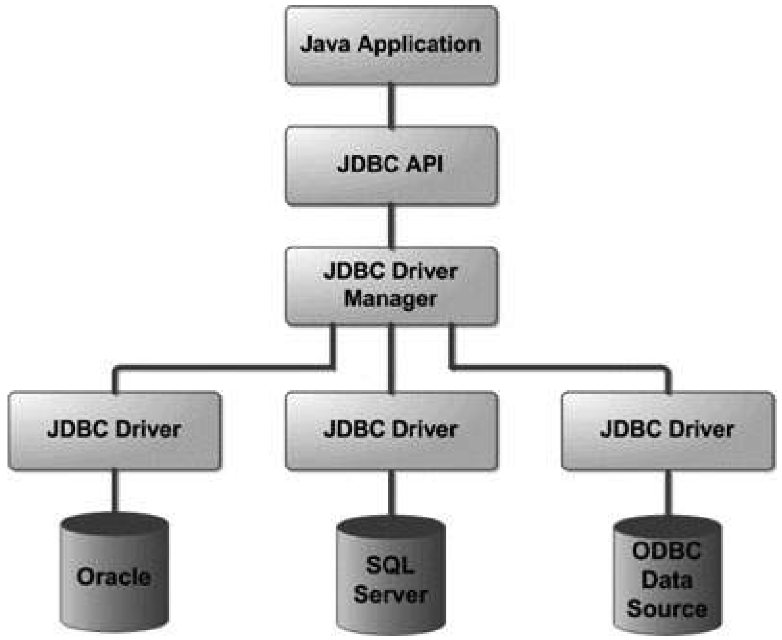
The JDBC API supports both two-tier and three-tier processing models for database access but in general, JDBC Architecture consists of two layers −

- **JDBC API:** This provides the application-to-JDBC Manager connection.
- **JDBC Driver API:** This supports the JDBC Manager-to-Driver Connection.

The JDBC API uses a driver manager and database-specific drivers to provide transparent connectivity to heterogeneous databases.

The JDBC driver manager ensures that the correct driver is used to access each data source. The driver manager is capable of supporting multiple concurrent drivers connected to multiple heterogeneous databases.

Following is the architectural diagram, which shows the location of the driver manager with respect to the JDBC drivers and the Java application −



图一、 JDBC Architecture

6. Common JDBC Components

The JDBC API provides the following interfaces and classes −

- **DriverManager:** This class manages a list of database drivers. Matches connection requests from the java application with the proper database driver using communication sub protocol. The first driver that recognizes a certain subprotocol under JDBC will be used to establish a database Connection.
- **Driver:** This interface handles the communications with the database server. You will interact directly with Driver objects very rarely. Instead, you use DriverManager objects, which manages objects of this type. It also abstracts the details associated with working with Driver objects.
- **Connection:** This interface with all methods for contacting a database. The connection object represents communication context, i.e., all communication with database is through connection object only.
- **Statement:** You use objects created from this interface to submit the SQL statements to the database. Some derived interfaces accept parameters in addition to executing stored procedures.
- **ResultSet:** These objects hold data retrieved from a database after you execute an SQL query using Statement objects. It acts as an iterator to allow you to move through its data.
- **SQLException:** This class handles any errors that occur in a database application.

7. SQL Syntax

Structured Query Language (SQL) is a standardized language that allows you to perform operations on a database, such as creating entries, reading content, updating content, and deleting entries.

SQL is supported by almost any database you will likely use, and it allows you to write database code independently of the underlying database.

7.1. Create Database

The CREATE DATABASE statement is used for creating a new database. The syntax is −

```
CREATE DATABASE DATABASE_NAME;
-- The following SQL statement creates a Database named EMP
CREATE DATABASE EMP;
```

7.2. Drop Database

The DROP DATABASE statement is used for deleting an existing database. The syntax is −

```
DROP DATABASE DATABASE_NAME;
```

Note: To create or drop a database you should have administrator privilege on your database server. Be careful, deleting a database would loss all the data stored in the database.

7.3. Create Table

The CREATE TABLE statement is used for creating a new table. The syntax is −

```
CREATE TABLE table_name
(
  column_name column_data_type,
  column_name column_data_type,
  column_name column_data_type
  ...
);
-- The following SQL statement creates a table named Employees with four columns −
CREATE TABLE Employees
(
  id INT NOT NULL,
  age INT NOT NULL,
  first VARCHAR(255),
  last VARCHAR(255),
  PRIMARY KEY ( id )
);
```

7.4. Drop Table

The DROP TABLE statement is used for deleting an existing table. The syntax is −

```
DROP TABLE table_name;
-- The following SQL statement deletes a table named Employees −
DROP TABLE Employees;
```

7.5. INSERT Data

The syntax for INSERT, looks similar to the following, where column1, column2, and so on represents the new data to appear in the respective columns −

```
INSERT INTO table_name VALUES (column1, column2, ...);
-- The following SQL INSERT statement inserts a new row in the Employees database created earlier −
INSERT INTO Employees VALUES (100, 18, 'Zara', 'Ali');
```

7.6. SELECT Data

The SELECT statement is used to retrieve data from a database. The syntax for SELECT is −

```
SELECT column_name, column_name, ...
   FROM table_name
   WHERE conditions;
-- The following SQL statement selects the age, first and last columns from the Employees table, where id column is 100 −
SELECT first, last, age
    FROM Employees
    WHERE id = 100;
-- The following SQL statement selects the age, first and last columns from the Employees table where *first* column contains *Zara* −
SELECT first, last, age
    FROM Employees
    WHERE first LIKE '%Zara%';
```

The WHERE clause can use the comparison operators such as =, !=, <, >, <=,and >=, as well as the BETWEEN and LIKE operators.

7.7. UPDATE Data

The UPDATE statement is used to update data. The syntax for UPDATE is −

```
UPDATE table_name
    SET column_name = value, column_name = value, ...
    WHERE conditions;
-- The following SQL UPDATE statement changes the age column of the employee whose id is 100 −
UPDATE Employees SET age=20 WHERE id=100;
```

The WHERE clause can use the comparison operators such as =, !=, <, >, <=,and >=, as well as the BETWEEN and LIKE operators.

7.8. DELETE Data

The DELETE statement is used to delete data from tables. The syntax for DELETE is −

```
DELETE FROM table_name WHERE conditions;
-- The following SQL DELETE statement deletes the record of the employee whose id is 100 −
DELETE FROM Employees WHERE id=100;
```

The WHERE clause can use the comparison operators such as =, !=, <, >, <=,and >=, as well as the BETWEEN and LIKE operators.

8. Creating JDBC Application

There are following six steps involved in building a JDBC application −

- **Import the packages:** Requires that you include the packages containing the JDBC classes needed for database programming. Most often, using *import java.sql.** will suffice.
- **Register the JDBC driver:** Requires that you initialize a driver so you can open a communication channel with the database.
- **Open a connection:** Requires using the *DriverManager.getConnection()* method to create a Connection object, which represents a physical connection with the database.
- **Execute a query:** Requires using an object of type Statement for building and submitting an SQL statement to the database.
- **Extract data from result set:** Requires that you use the appropriate *ResultSet.getXXX()* method to retrieve the data from the result set.
- **Clean up the environment:** Requires explicitly closing all database resources versus relying on the JVM's garbage collection.

9. Sample Code

This sample example can serve as a **template** when you need to create your own JDBC application in the future.

This sample code has been written based on the environment and database setup done in the previous chapter.

Copy and paste the following example in FirstExample.java, compile and run as follows −

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

public class FirstExample {
  // 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 = "username";
  static final String PASS = "password";

  public static void main(String[] args) {
  Connection conn = null;
  Statement stmt = null;
  try{
    //STEP 2: Register JDBC driver
    Class.forName("com.mysql.jdbc.Driver");
```

Sample Code

```
//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...");
stmt = conn.createStatement();
String sql;
sql = "SELECT id, first, last, age FROM Employees";
ResultSet rs = stmt.executeQuery(sql);

//STEP 5: Extract data from result set
while(rs.next()){
  //Retrieve by column name
  int id  = rs.getInt("id");
  int age = rs.getInt("age");
  String first = rs.getString("first");
  String last = rs.getString("last");

  //Display values
  System.out.print("ID: " + id);
  System.out.print(", Age: " + age);
  System.out.print(", First: " + first);
  System.out.println(", Last: " + last);
}
//STEP 6: Clean-up environment
rs.close();
```

Sample Code

```
      stmt.close();
      conn.close();
   }catch(SQLException se){
      //Handle errors for JDBC
      se.printStackTrace();
   }catch(Exception e){
      //Handle errors for Class.forName
      e.printStackTrace();
   }finally{
      //finally block used to close resources
      try{
         if(stmt!=null)
            stmt.close();
      }catch(SQLException se2){
      }// nothing we can do
      try{
         if(conn!=null)
            conn.close();
      }catch(SQLException se){
         se.printStackTrace();
      }//end finally try
   }//end try
   System.out.println("Goodbye!");
}//end main
}//end FirstExample
```

Now let us compile the above example as follows −

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

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

```
C:\>java FirstExample
Connecting to database...
Creating statement...
ID: 100, Age: 18, First: Zara, Last: Ali
ID: 101, Age: 25, First: Mahnaz, Last: Fatma
ID: 102, Age: 30, First: Zaid, Last: Khan
ID: 103, Age: 28, First: Sumit, Last: Mittal
C:\>
```

10. Statements

Once a connection is obtained we can interact with the database. The JDBC *Statement, CallableStatement,* and *PreparedStatement* interfaces define the methods and properties that enable you to send SQL or PL/SQL commands and receive data from your database.

They also define methods that help bridge data type differences between Java and SQL data types used in a database.

The following table provides a summary of each interface's purpose to decide on the interface to use.

Interfaces	**Recommended Use**
Statement	Use this for general-purpose access to your database. Useful when you are using static SQL statements at runtime. The Statement interface cannot accept parameters.
PreparedStatement	Use this when you plan to use the SQL statements many times. The PreparedStatement interface accepts input parameters at runtime.
CallableStatement	Use this when you want to access the database stored procedures. The CallableStatement interface can also accept runtime input parameters.

10.1. The PreparedStatement Objects

The *PreparedStatement* interface extends the Statement interface, which gives you added functionality with a couple of advantages over a generic Statement object.

This statement gives you the flexibility of supplying arguments dynamically.

All parameters in JDBC are represented by the **?** symbol, which is known as the parameter marker. You must supply values for every parameter before executing the SQL statement.

```
PreparedStatement pstmt = null;
try {
  String SQL = "Update Employees SET age = ? WHERE id = ?";
  pstmt = conn.prepareStatement(SQL);
  . . .
}
catch (SQLException e) {
  . . .
}
finally {
  pstmt.close();
}
```

The **setXXX()** methods bind values to the parameters, where **XXX** represents the Java data type of the value you wish to bind to the input parameter. If you forget to supply the values, you will receive an SQLException.

Each parameter marker is referred by its ordinal position. The first marker represents position 1, the next position 2, and so forth. This method differs from that of Java array indices, which starts at 0.

Statements

All of the **Statement object's** methods for interacting with the database (a) execute(), (b) executeQuery(), and (c) executeUpdate() also work with the PreparedStatement object. However, the methods are modified to use SQL statements that can input the parameters.

10.2. Sample Code

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

public class JDBCPrepStmt {
    public static void main(String[] args) {
        Connection connection = null;
        try {
            // Loading driver
            Class.forName("com.mysql.jdbc.Driver");

            // Creating connection
            connection = DriverManager.getConnection("jdbc:mysql://localhost:3306/netjs", "root", "admin");
            JDBCPrepStmt prep = new JDBCPrepStmt();
            prep.insertEmployee(connection, "Kate", 24);
            prep.updateEmployee(connection, 22, 30);
            prep.displayEmployee(connection, 22);

            //prep.deleteEmployee(connection, 24);
        } catch (ClassNotFoundException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
```

Statements

```
    } catch (SQLException e) {
      // TODO Auto-generated catch block
      e.printStackTrace();
    } finally {
      if (connection != null) {
        // closing connection
        try {
          connection.close();
        } catch (SQLException e) {
          // TODO Auto-generated catch block
          e.printStackTrace();
        }
      } // if condition
    } // finally
  }

  // Method to insert
  private void insertEmployee(Connection connection, String name, int age) throws SQLException {
    String insertSQL = "Insert into employee (name, age) values (?, ?)";
    PreparedStatement prepStmt = null;
    try {
      prepStmt = connection.prepareStatement(insertSQL);
      prepStmt.setString(1, name);
      prepStmt.setInt(2, age);
      int count = prepStmt.executeUpdate();
      System.out.println("Count of rows inserted " + count);
    } finally {
        if (prepStmt != null) {
            prepStmt.close();
        }
    }
}

// Method to update
private void updateEmployee(Connection connection, int id, int age) throws SQLException {
    String updateSQL = "Update employee set age = ? where id = ?";
    PreparedStatement prepStmt = null;
    try {
        prepStmt = connection.prepareStatement(updateSQL);
        prepStmt.setInt(1, age);
        prepStmt.setInt(2, id);
        int count = prepStmt.executeUpdate();
        System.out.println("Count of rows updated " + count);
    } finally {
        if (prepStmt != null) {
            prepStmt.close();
        }
    }
}

// Method to delete
private void deleteEmployee(Connection connection, int id) throws SQLException {
    String deleteSQL = "Delete from employee where id = ?";
    PreparedStatement prepStmt = null;
```

Statements

```
    try {
        prepStmt = connection.prepareStatement(deleteSQL);
        prepStmt.setInt(1, id);
        int count = prepStmt.executeUpdate();
        System.out.println("Count of rows deleted " + count);
    } finally {
        if (prepStmt != null) {
            prepStmt.close();
        }
    }
}

// Method to retrieve
private void displayEmployee(Connection connection, int id) throws SQLException {
    String selectSQL = "Select * from employee where id = ?";
    PreparedStatement prepStmt = null;
    try {
        prepStmt = connection.prepareStatement(selectSQL);
        prepStmt.setInt(1, id);
        ResultSet rs = prepStmt.executeQuery();
        while (rs.next()) {
            System.out.println(
                    "id : " + rs.getInt("id") + " Name : " + rs.getString("name") + " Age : " + rs.getInt("age"));
        }
    } finally {
        if (prepStmt != null) {
            prepStmt.close();
            }
        }
    }


}
```

11. Transactions

If your JDBC Connection is in *auto-commit* mode, which it is by default, then every SQL statement is committed to the database upon its completion.

That may be fine for simple applications, but there are three reasons why you may want to turn off the auto-commit and manage your own transactions –

- To increase performance.
- To maintain the integrity of business processes.
- To use distributed transactions.

Transactions enable you to control if, and when, changes are applied to the database. It treats a single SQL statement or a group of SQL statements as one logical unit, and if any statement fails, the whole transaction fails.

To enable manual- transaction support instead of the *auto-commit* mode that the JDBC driver uses by default, use the Connection object's **setAutoCommit()** method. If you pass a boolean false to setAutoCommit( ), you turn off auto-commit. You can pass a boolean true to turn it back on again.

For example, if you have a Connection object named conn, code the following to turn off auto-commit –

```
conn.setAutoCommit(false);
```

11.1. Commit & Rollback

Once you are done with your changes and you want to commit the changes then call **commit()** method on connection object as follows –

```
conn.commit( );
```

Otherwise, to roll back updates to the database made using the Connection named conn, use the following code –

```
conn.rollback( );
```

The following example illustrates the use of a commit and rollback object –

Transactions

```
try{
  //Assume a valid connection object conn
  conn.setAutoCommit(false);
  Statement stmt = conn.createStatement();

  String SQL = "INSERT INTO Employees  " +
               "VALUES (106, 20, 'Rita', 'Tez')";
  stmt.executeUpdate(SQL);
  //Submit a malformed SQL statement that breaks
  String SQL = "INSERTED IN Employees  " +
               "VALUES (107, 22, 'Sita', 'Singh')";
  stmt.executeUpdate(SQL);
  // If there is no error.
  conn.commit();
}catch(SQLException se){
  // If there is any error.
  conn.rollback();
}
```

In this case, none of the above INSERT statement would success and everything would be rolled back.

11.2. Using Savepoints

The new JDBC 3.0 Savepoint interface gives you the additional transactional control. Most modern DBMS, support savepoints within their environments such as Oracle's PL/SQL.

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

The Connection object has two new methods that help you manage savepoints –

- **setSavepoint(String savepointName):** Defines a new savepoint. It also returns a Savepoint object.
- **releaseSavepoint(Savepoint savepointName):** Deletes a savepoint. Notice that it requires a Savepoint object as a parameter. This object is usually a savepoint generated by the setSavepoint() method.

There is one **rollback (String savepointName)** method, which rolls back work to the specified savepoint.

The following example illustrates the use of a Savepoint object –

Transactions

```
try{
  //Assume a valid connection object conn
  conn.setAutoCommit(false);
  Statement stmt = conn.createStatement();

  //set a Savepoint
  Savepoint savepoint1 = conn.setSavepoint("Savepoint1");
  String SQL = "INSERT INTO Employees " +
               "VALUES (106, 20, 'Rita', 'Tez')";
  stmt.executeUpdate(SQL);
  //Submit a malformed SQL statement that breaks
  String SQL = "INSERTED IN Employees " +
               "VALUES (107, 22, 'Sita', 'Tez')";
  stmt.executeUpdate(SQL);
  // If there is no error, commit the changes.
  conn.commit();

}catch(SQLException se){
  // If there is any error.
  conn.rollback(savepoint1);
}
```

In this case, none of the above INSERT statement would success and everything would be rolled back.

Thank you

全国统一咨询热线：400-690-6115

北京|上海|广州|深圳|天津|成都|重庆|武汉|济南|青岛|杭州|西安