Java & MySQL - Statement

JDBC Statement interface defines the methods and properties to enable send SQL commands to MySQL database and retrieve data from the database. Statement is used for general-purpose access to your database. It is useful when you are using static SQL statements at runtime. The Statement interface cannot accept parameters.

Before you can use a Statement object to execute a SQL statement, you need to create one using the Connection object's createStatement( ) method, as in the following example −

Statement stmt = null;

try {

stmt = conn.createStatement( );

. . .

}

catch (SQLException e) {

. . .

}

finally {

. . .

}

Once you've created a Statement object, you can then use it to execute an SQL statement with one of its three execute methods.

**boolean execute (String SQL)** − Returns a boolean value of true if a ResultSet object can be retrieved; otherwise, it returns false. Use this method to execute SQL DDL statements or when you need to use truly dynamic SQL.

**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, as you would with a SELECT statement.

Closing Statement Object

Just as you close a Connection object to save database resources, for the same reason you should also close the Statement object.

A simple call to the close() method will do the job. If you close the Connection object first, it will close the Statement object as well. However, you should always explicitly close the Statement object to ensure proper cleanup.

Statement stmt = null;

try {

stmt = conn.createStatement( );

. . .

}

catch (SQLException e) {

. . .

}

finally {

stmt.close();

}

We're using try with resources which handles the resource closure automatically. Following example demonstrates all of the above said concepts.

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

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

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.ResultSet;

import java.sql.SQLException;

import java.sql.Statement;

public class TestApplication {

static final String DB\_URL = "jdbc:mysql://localhost/TUTORIALSPOINT";

static final String USER = "guest";

static final String PASS = "guest123";

static final String QUERY = "SELECT id, first, last, age FROM Employees";

static final String UPDATE\_QUERY = "UPDATE Employees set age=30 WHERE id=103";

public static void main(String[] args) {

// Open a connection

try(Connection conn = DriverManager.getConnection(DB\_URL, USER, PASS);

Statement stmt = conn.createStatement();

) {

// Let us check if it returns a true Result Set or not.

Boolean ret = stmt.execute(UPDATE\_QUERY);

System.out.println("Return value is : " + ret.toString() );

// Let us update age of the record with ID = 103;

int rows = stmt.executeUpdate(UPDATE\_QUERY);

System.out.println("Rows impacted : " + rows );

// Let us select all the records and display them.

ResultSet rs = stmt.executeQuery(QUERY);

// Extract data from result set

while (rs.next()) {

// Retrieve by column name

System.out.print("ID: " + rs.getInt("id"));

System.out.print(", Age: " + rs.getInt("age"));

System.out.print(", First: " + rs.getString("first"));

System.out.println(", Last: " + rs.getString("last"));

}

rs.close();

} catch (SQLException e) {

e.printStackTrace();

}

}

}

Now let us compile the above example as follows −

C:\>javac TestApplication.java

C:\>

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

C:\>java TestApplication

Return value is : false

Rows impacted : 1

ID: 100, Age: 18, First: Zara, Last: Ali

ID: 101, Age: 25, First: Mehnaz, Last: Fatma

ID: 102, Age: 30, First: Zaid, Last: Khan

ID: 103, Age: 30, First: Sumit, Last: Mittal

C:\>

Java & MySQL - PreparedStatement

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.

Creating PreparedStatement Object

PreparedStatement pstmt = null;

try {

String SQL = "Update Employees SET age = ? WHERE id = ?";

pstmt = conn.prepareStatement(SQL);

. . .

}

catch (SQLException e) {

. . .

}

finally {

. . .

}

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.

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.

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.

Closing PreparedStatement Object

Just as you close a Statement object, for the same reason you should also close the PreparedStatement object.

A simple call to the close() method will do the job. If you close the Connection object first, it will close the PreparedStatement object as well. However, you should always explicitly close the PreparedStatement object to ensure proper cleanup.

PreparedStatement pstmt = null;

try {

String SQL = "Update Employees SET age = ? WHERE id = ?";

pstmt = conn.prepareStatement(SQL);

. . .

}

catch (SQLException e) {

. . .

}

finally {

pstmt.close();

}

We're using try with resources which handles the resource closure automatically. Following example demonstrates all of the above said concepts.

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

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

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.PreparedStatement;

import java.sql.ResultSet;

import java.sql.SQLException;

public class TestApplication {

static final String DB\_URL = "jdbc:mysql://localhost/TUTORIALSPOINT";

static final String USER = "guest";

static final String PASS = "guest123";

static final String QUERY = "SELECT id, first, last, age FROM Employees";

static final String UPDATE\_QUERY = "UPDATE Employees set age=? WHERE id=?";

public static void main(String[] args) {

// Open a connection

try(Connection conn = DriverManager.getConnection(DB\_URL, USER, PASS);

PreparedStatement stmt = conn.prepareStatement(UPDATE\_QUERY);

) {

// Bind values into the parameters.

stmt.setInt(1, 35); // This would set age

stmt.setInt(2, 102); // This would set ID

// Let us update age of the record with ID = 102;

int rows = stmt.executeUpdate();

System.out.println("Rows impacted : " + rows );

// Let us select all the records and display them.

ResultSet rs = stmt.executeQuery(QUERY);

// Extract data from result set

while (rs.next()) {

// Retrieve by column name

System.out.print("ID: " + rs.getInt("id"));

System.out.print(", Age: " + rs.getInt("age"));

System.out.print(", First: " + rs.getString("first"));

System.out.println(", Last: " + rs.getString("last"));

}

rs.close();

} catch (SQLException e) {

e.printStackTrace();

}

}

}

Now let us compile the above example as follows −

C:\>javac TestApplication.java

C:\>

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

C:\>java TestApplication

Return value is : false

Rows impacted : 1

ID: 100, Age: 18, First: Zara, Last: Ali

ID: 101, Age: 25, First: Mehnaz, Last: Fatma

ID: 102, Age: 35, First: Zaid, Last: Khan

ID: 103, Age: 30, First: Sumit, Last: Mittal

C:\>

Java & MySQL - CallableStatement

The *CallableStatement* interface is used to execute a call to a database stored procedure.

Suppose, you need to execute the following stored procedure in TUTORIALSPOINT database −

DELIMITER $$

DROP PROCEDURE IF EXISTS `TUTORIALSPOINT`.`getEmpName` $$

CREATE PROCEDURE `TUTORIALSPOINT`.`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.

Closing CallableStatement Object

Just as you close other Statement object, for the same reason you should also close the CallableStatement object.

A simple call to the close() method will do the job. If you close the Connection object first, it will close the CallableStatement object as well. However, you should always explicitly close the CallableStatement object to ensure proper cleanup.

CallableStatement cstmt = null;

try {

String SQL = "{call getEmpName (?, ?)}";

cstmt = conn.prepareCall (SQL);

. . .

}

catch (SQLException e) {

. . .

}

finally {

cstmt.close();

}

We're using try with resources which handles the resource closure automatically. Following example demonstrates all of the above said concepts.

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

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

import java.sql.CallableStatement;

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.SQLException;

public class TestApplication {

static final String DB\_URL = "jdbc:mysql://localhost/TUTORIALSPOINT";

static final String USER = "guest";

static final String PASS = "guest123";

static final String QUERY = "{call getEmpName (?, ?)}";

public static void main(String[] args) {

// Open a connection

try(Connection conn = DriverManager.getConnection(DB\_URL, USER, PASS);

CallableStatement stmt = conn.prepareCall(QUERY);

) {

// Bind values into the parameters.

stmt.setInt(1, 102); // This would set ID

// 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: 102 is " + empName);

} catch (SQLException e) {

e.printStackTrace();

}

}

}

Now let us compile the above example as follows −

C:\>javac TestApplication.java

C:\>

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

C:\>java TestApplication

Executing stored procedure...

Emp Name with ID: 102 is Zaid

C:\>

JDBC SQL Escape Syntax

The escape syntax gives you the flexibility to use database specific features unavailable to you by using standard JDBC methods and properties.

The general SQL escape syntax format is as follows −

{keyword 'parameters'}

Here are the following escape sequences, which you would find very useful while performing the JDBC programming −

d, t, ts Keywords

They help identify date, time, and timestamp literals. As you know, no two DBMSs represent time and date the same way. This escape syntax tells the driver to render the date or time in the target database's format. For Example −

{d 'yyyy-mm-dd'}

Where yyyy = year, mm = month; dd = date. Using this syntax {d '2009-09-03'} is March 9, 2009.

Here is a simple example showing how to INSERT date in a table −

//Create a Statement object

stmt = conn.createStatement();

//Insert data ==> ID, First Name, Last Name, DOB

String sql="INSERT INTO STUDENTS VALUES" +

"(100,'Zara','Ali', {d '2001-12-16'})";

stmt.executeUpdate(sql);

Similarly, you can use one of the following two syntaxes, either **t** or **ts** −

{t 'hh:mm:ss'}

Where hh = hour; mm = minute; ss = second. Using this syntax {t '13:30:29'} is 1:30:29 PM.

{ts 'yyyy-mm-dd hh:mm:ss'}

This is combined syntax of the above two syntax for 'd' and 't' to represent timestamp.

escape Keyword

This keyword identifies the escape character used in LIKE clauses. Useful when using the SQL wildcard %, which matches zero or more characters. For example −

String sql = "SELECT symbol FROM MathSymbols WHERE symbol LIKE '\%' {escape '\'}";

stmt.execute(sql);

If you use the backslash character (\) as the escape character, you also have to use two backslash characters in your Java String literal, because the backslash is also a Java escape character.

fn Keyword

This keyword represents scalar functions used in a DBMS. For example, you can use SQL function *length* to get the length of a string −

{fn length('Hello World')}

This returns 11, the length of the character string 'Hello World'.

call Keyword

This keyword is used to call the stored procedures. For example, for a stored procedure requiring an IN parameter, use the following syntax −

{call my\_procedure(?)};

For a stored procedure requiring an IN parameter and returning an OUT parameter, use the following syntax −

{? = call my\_procedure(?)};

oj Keyword

This keyword is used to signify outer joins. The syntax is as follows −

{oj outer-join}

Where outer-join = table {LEFT|RIGHT|FULL} OUTERJOIN {table | outer-join} on search-condition. For example −

String sql = "SELECT Employees FROM {oj ThisTable RIGHT OUTER JOIN ThatTable on id = '100'}";

stmt.execute(sql);

Java & MySQL - ResultSet

The SQL statements that read data from a database query, return the data in a result set. The SELECT statement is the standard way to select rows from a database and view them in a result set. The *java.sql.ResultSet* interface represents the result set of a database query.

A ResultSet object maintains a cursor that points to the current row in the result set. The term "result set" refers to the row and column data contained in a ResultSet object.

The methods of the ResultSet interface can be broken down into three categories −

**Navigational methods** − Used to move the cursor around.

**Get methods** − Used to view the data in the columns of the current row being pointed by the cursor.

**Update methods** − Used to update the data in the columns of the current row. The updates can then be updated in the underlying database as well.

The cursor is movable based on the properties of the ResultSet. These properties are designated when the corresponding Statement that generates the ResultSet is created.

JDBC provides the following connection methods to create statements with desired ResultSet −

**createStatement(int RSType, int RSConcurrency);**

**prepareStatement(String SQL, int RSType, int RSConcurrency);**

**prepareCall(String sql, int RSType, int RSConcurrency);**

The first argument indicates the type of a ResultSet object and the second argument is one of two ResultSet constants for specifying whether a result set is read-only or updatable.

Type of ResultSet

The possible RSType are given below. If you do not specify any ResultSet type, you will automatically get one that is TYPE\_FORWARD\_ONLY.

|  |  |
| --- | --- |
| **Type** | **Description** |
| ResultSet.TYPE\_FORWARD\_ONLY | The cursor can only move forward in the result set. |
| ResultSet.TYPE\_SCROLL\_INSENSITIVE | The cursor can scroll forward and backward, and the result set is not sensitive to changes made by others to the database that occur after the result set was created. |
| ResultSet.TYPE\_SCROLL\_SENSITIVE. | The cursor can scroll forward and backward, and the result set is sensitive to changes made by others to the database that occur after the result set was created. |

Concurrency of ResultSet

The possible RSConcurrency are given below. If you do not specify any Concurrency type, you will automatically get one that is CONCUR\_READ\_ONLY.

|  |  |
| --- | --- |
| **Concurrency** | **Description** |
| ResultSet.CONCUR\_READ\_ONLY | Creates a read-only result set. This is the default |
| ResultSet.CONCUR\_UPDATABLE | Creates an updateable result set. |

All our examples written so far can be written as follows, which initializes a Statement object to create a forward-only, read only ResultSet object −

try(

Statement stmt = conn.createStatement(

ResultSet.TYPE\_FORWARD\_ONLY,

ResultSet.CONCUR\_READ\_ONLY);)

}

catch(Exception ex) {

....

}

finally {

....

}

Java & MySQL - Navigating a ResultSet

There are several methods in the ResultSet interface that involve moving the cursor, including −

|  |  |
| --- | --- |
| **S.N.** | **Methods & Description** |
| 1 | **public void beforeFirst() throws SQLException**  Moves the cursor just before the first row. |
| 2 | **public void afterLast() throws SQLException**  Moves the cursor just after the last row. |
| 3 | **public boolean first() throws SQLException**  Moves the cursor to the first row. |
| 4 | **public void last() throws SQLException**  Moves the cursor to the last row. |
| 5 | **public boolean absolute(int row) throws SQLException**  Moves the cursor to the specified row. |
| 6 | **public boolean relative(int row) throws SQLException**  Moves the cursor the given number of rows forward or backward, from where it is currently pointing. |
| 7 | **public boolean previous() throws SQLException**  Moves the cursor to the previous row. This method returns false if the previous row is off the result set. |
| 8 | **public boolean next() throws SQLException**  Moves the cursor to the next row. This method returns false if there are no more rows in the result set. |
| 9 | **public int getRow() throws SQLException**  Returns the row number that the cursor is pointing to. |
| 10 | **public void moveToInsertRow() throws SQLException**  Moves the cursor to a special row in the result set that can be used to insert a new row into the database. The current cursor location is remembered. |
| 11 | **public void moveToCurrentRow() throws SQLException**  Moves the cursor back to the current row if the cursor is currently at the insert row; otherwise, this method does nothing |

Following is the example which makes use of few navigation methods described.

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

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

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.ResultSet;

import java.sql.SQLException;

import java.sql.Statement;

public class TestApplication {

static final String DB\_URL = "jdbc:mysql://localhost/TUTORIALSPOINT";

static final String USER = "guest";

static final String PASS = "guest123";

static final String QUERY = "SELECT id, first, last, age FROM Employees";

public static void main(String[] args) {

// Open a connection

try(Connection conn = DriverManager.getConnection(DB\_URL, USER, PASS);

Statement stmt = conn.createStatement(

ResultSet.TYPE\_SCROLL\_INSENSITIVE,

ResultSet.CONCUR\_READ\_ONLY);

ResultSet rs = stmt.executeQuery(QUERY);

) {

// Move cursor to the last row.

System.out.println("Moving cursor to the last...");

rs.last();

// Extract data from result set

System.out.println("Displaying record...");

//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);

// Move cursor to the first row.

System.out.println("Moving cursor to the first row...");

rs.first();

// Extract data from result set

System.out.println("Displaying record...");

// Retrieve by column name

id = rs.getInt("id");

age = rs.getInt("age");

first = rs.getString("first");

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);

// Move cursor to the first row.

System.out.println("Moving cursor to the next row...");

rs.next();

// Extract data from result set

System.out.println("Displaying record...");

id = rs.getInt("id");

age = rs.getInt("age");

first = rs.getString("first");

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);

} catch (SQLException e) {

e.printStackTrace();

}

}

}

Now let us compile the above example as follows −

C:\>javac TestApplication.java

C:\>

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

C:\>java TestApplication

Moving cursor to the last...

Displaying record...

ID: 103, Age: 30, First: Sumit, Last: Mittal

Moving cursor to the first row...

Displaying record...

ID: 100, Age: 18, First: Zara, Last: Ali

Moving cursor to the next row...

Displaying record...

ID: 101, Age: 25, First: Mehnaz, Last: Fatma

C:\>

Java & MySQL - Viewing a ResultSet

The ResultSet interface contains dozens of methods for getting the data of the current row.

There is a get method for each of the possible data types, and each get method has two versions −

One that takes in a column name.

One that takes in a column index.

For example, if the column you are interested in viewing contains an int, you need to use one of the getInt() methods of ResultSet −

|  |  |
| --- | --- |
| **S.N.** | **Methods & Description** |
| 1 | **public int getInt(String columnName) throws SQLException**  Returns the int in the current row in the column named columnName. |
| 2 | **public int getInt(int columnIndex) throws SQLException**  Returns the int in the current row in the specified column index. The column index starts at 1, meaning the first column of a row is 1, the second column of a row is 2, and so on. |

Similarly, there are get methods in the ResultSet interface for each of the eight Java primitive types, as well as common types such as java.lang.String, java.lang.Object, and java.net.URL.

There are also methods for getting SQL data types java.sql.Date, java.sql.Time, java.sql.TimeStamp, java.sql.Clob, and java.sql.Blob. Check the documentation for more information about using these SQL data types.

Following is the example which makes use of few viewing methods described.

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

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

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.ResultSet;

import java.sql.SQLException;

import java.sql.Statement;

public class TestApplication {

static final String DB\_URL = "jdbc:mysql://localhost/TUTORIALSPOINT";

static final String USER = "guest";

static final String PASS = "guest123";

static final String QUERY = "SELECT id, first, last, age FROM Employees";

public static void main(String[] args) {

// Open a connection

try(Connection conn = DriverManager.getConnection(DB\_URL, USER, PASS);

Statement stmt = conn.createStatement(

ResultSet.TYPE\_SCROLL\_INSENSITIVE,

ResultSet.CONCUR\_READ\_ONLY);

ResultSet rs = stmt.executeQuery(QUERY);

) {

// Move cursor to the last row.

System.out.println("Moving cursor to the last...");

rs.last();

// Extract data from result set

System.out.println("Displaying record...");

//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);

// Move cursor to the first row.

System.out.println("Moving cursor to the first row...");

rs.first();

// Extract data from result set

System.out.println("Displaying record...");

// Retrieve by column name

id = rs.getInt("id");

age = rs.getInt("age");

first = rs.getString("first");

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);

// Move cursor to the first row.

System.out.println("Moving cursor to the next row...");

rs.next();

// Extract data from result set

System.out.println("Displaying record...");

id = rs.getInt("id");

age = rs.getInt("age");

first = rs.getString("first");

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);

} catch (SQLException e) {

e.printStackTrace();

}

}

}

Now let us compile the above example as follows −

C:\>javac TestApplication.java

C:\>

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

C:\>java TestApplication

Moving cursor to the last...

Displaying record...

ID: 103, Age: 30, First: Sumit, Last: Mittal

Moving cursor to the first row...

Displaying record...

ID: 100, Age: 18, First: Zara, Last: Ali

Moving cursor to the next row...

Displaying record...

ID: 101, Age: 25, First: Mehnaz, Last: Fatma

C:\>

Java & MySQL - Updating a ResultSet

The ResultSet interface contains a collection of update methods for updating the data of a result set.

As with the get methods, there are two update methods for each data type −

One that takes in a column name.

One that takes in a column index.

For example, to update a String column of the current row of a result set, you would use one of the following updateString() methods −

|  |  |
| --- | --- |
| **S.N.** | **Methods & Description** |
| 1 | **public void updateString(int columnIndex, String s) throws SQLException**  Changes the String in the specified column to the value of s. |
| 2 | **public void updateString(String columnName, String s) throws SQLException**  Similar to the previous method, except that the column is specified by its name instead of its index. |

There are update methods for the eight primitive data types, as well as String, Object, URL, and the SQL data types in the java.sql package.

Updating a row in the result set changes the columns of the current row in the ResultSet object, but not in the underlying database. To update your changes to the row in the database, you need to invoke one of the following methods.

|  |  |
| --- | --- |
| **S.N.** | **Methods & Description** |
| 1 | **public void updateRow()**  Updates the current row by updating the corresponding row in the database. |
| 2 | **public void deleteRow()**  Deletes the current row from the database |
| 3 | **public void refreshRow()**  Refreshes the data in the result set to reflect any recent changes in the database. |
| 4 | **public void cancelRowUpdates()**  Cancels any updates made on the current row. |
| 5 | **public void insertRow()**  Inserts a row into the database. This method can only be invoked when the cursor is pointing to the insert row. |

Following is the example, which makes use of the **ResultSet.CONCUR\_UPDATABLE** and **ResultSet.TYPE\_SCROLL\_INSENSITIVE** described in the Result Set tutorial. This example would explain INSERT, UPDATE and DELETE operation on a table.

It should be noted that tables you are working on should have Primary Key set properly. Let's update our referece table first.

Drop the table and Create the table **Employees** again as follows −

mysql> use TUTORIALSPOINT;

mysql> drop table Employees;

Query OK, 0 rows affected (0.08 sec)

mysql> create table Employees

-> (

-> id int primary key auto\_increment,

-> age int not null,

-> first varchar (255),

-> last varchar (255)

-> );

Query OK, 0 rows affected (0.08 sec)

mysql>

Create Data Records

Finally you create few records in Employee table as follows −

mysql> INSERT INTO Employees(AGE, FIRST, LAST) VALUES (18, 'Zara', 'Ali');

Query OK, 1 row affected (0.05 sec)

mysql> INSERT INTO Employees(AGE, FIRST, LAST) VALUES (25, 'Mahnaz', 'Fatma');

Query OK, 1 row affected (0.00 sec)

mysql> INSERT INTO Employees(AGE, FIRST, LAST) VALUES (30, 'Zaid', 'Khan');

Query OK, 1 row affected (0.00 sec)

mysql> INSERT INTO Employees(AGE, FIRST, LAST) VALUES (28, 'Sumit', 'Mittal');

Query OK, 1 row affected (0.00 sec)

mysql>

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

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.ResultSet;

import java.sql.SQLException;

import java.sql.Statement;

public class TestApplication {

static final String DB\_URL = "jdbc:mysql://localhost/TUTORIALSPOINT";

static final String USER = "guest";

static final String PASS = "guest123";

static final String QUERY = "SELECT id, first, last, age FROM Employees";

public static void printResultSet(ResultSet rs) throws SQLException{

// Ensure we start with first row

rs.beforeFirst();

while(rs.next()){

// Display values

System.out.print("ID: " + rs.getInt("id"));

System.out.print(", Age: " + rs.getInt("age"));

System.out.print(", First: " + rs.getString("first"));

System.out.println(", Last: " + rs.getString("last"));

}

System.out.println();

}

public static void main(String[] args) {

// Open a connection

try(Connection conn = DriverManager.getConnection(DB\_URL, USER, PASS);

Statement stmt = conn.createStatement(

ResultSet.TYPE\_SCROLL\_INSENSITIVE,

ResultSet.CONCUR\_UPDATABLE);

ResultSet rs = stmt.executeQuery(QUERY);

) {

System.out.println("List result set for reference....");

printResultSet(rs);

// Loop through result set and add 5 in age

// Move to Before first position so while-loop works properly

rs.beforeFirst();

//STEP 7: Extract data from result set

while(rs.next()){

// Retrieve by column name

int newAge = rs.getInt("age") + 5;

rs.updateDouble( "age", newAge );

rs.updateRow();

}

System.out.println("List result set showing new ages...");

printResultSet(rs);

// Insert a record into the table.

// Move to insert row and add column data with updateXXX()

System.out.println("Inserting a new record...");

rs.moveToInsertRow();

rs.updateString("first","John");

rs.updateString("last","Paul");

rs.updateInt("age",40);

// Commit row

rs.insertRow();

System.out.println("List result set showing new set...");

printResultSet(rs);

} catch (SQLException e) {

e.printStackTrace();

}

}

}

Now let us compile the above example as follows −

C:\>javac TestApplication.java

C:\>

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

C:\>java TestApplication

List result set for reference....

ID: 1, Age: 18, First: Zara, Last: Ali

ID: 2, Age: 25, First: Mahnaz, Last: Fatma

ID: 3, Age: 30, First: Zaid, Last: Khan

ID: 4, Age: 28, First: Sumit, Last: Mittal

List result set showing new ages...

ID: 1, Age: 23, First: Zara, Last: Ali

ID: 2, Age: 30, First: Mahnaz, Last: Fatma

ID: 3, Age: 35, First: Zaid, Last: Khan

ID: 4, Age: 33, First: Sumit, Last: Mittal

Inserting a new record...

List result set showing new set...

ID: 1, Age: 23, First: Zara, Last: Ali

ID: 2, Age: 30, First: Mahnaz, Last: Fatma

ID: 3, Age: 35, First: Zaid, Last: Khan

ID: 4, Age: 33, First: Sumit, Last: Mittal

ID: 5, Age: 40, First: John, Last: Paul

List the record before deleting...

ID: 1, Age: 23, First: Zara, Last: Ali

ID: 2, Age: 30, First: Mahnaz, Last: Fatma

ID: 3, Age: 35, First: Zaid, Last: Khan

ID: 4, Age: 33, First: Sumit, Last: Mittal

ID: 5, Age: 40, First: John, Last: Paul

C:\>

Java & MySQL - 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);

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( );

Using Savepoints

The new JDBC 3.0 Savepoint interface gives you the additional transactional control.

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.

Java & MySQL - 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 −

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.

Following is the example, which makes use of **commit** and **rollback** described.

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

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

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.ResultSet;

import java.sql.SQLException;

import java.sql.Statement;

public class TestApplication {

static final String DB\_URL = "jdbc:mysql://localhost/TUTORIALSPOINT";

static final String USER = "guest";

static final String PASS = "guest123";

static final String QUERY = "SELECT id, first, last, age FROM Employees";

static final String INSERT\_QUERY = "INSERT INTO Employees (first, last, age) values('Rita', 'Tez', 20)";

static final String INSERT\_QUERY\_2 = "INSERT INTO Employees (first, last, age) values('Sita', 'Singh', 20)";

public static void printResultSet(ResultSet rs) throws SQLException{

// Ensure we start with first row

rs.beforeFirst();

while(rs.next()){

// Display values

System.out.print("ID: " + rs.getInt("id"));

System.out.print(", Age: " + rs.getInt("age"));

System.out.print(", First: " + rs.getString("first"));

System.out.println(", Last: " + rs.getString("last"));

}

System.out.println();

}

public static void main(String[] args) {

Connection conn = null;

Statement stmt = null;

try{

// Open a connection

System.out.println("Connecting to database...");

conn = DriverManager.getConnection(DB\_URL,USER,PASS);

// Set auto commit as false.

conn.setAutoCommit(false);

// Execute a query to create statment with

// required arguments for RS example.

System.out.println("Creating statement...");

stmt = conn.createStatement(

ResultSet.TYPE\_SCROLL\_INSENSITIVE,

ResultSet.CONCUR\_UPDATABLE);

// INSERT a row into Employees table

System.out.println("Inserting one row....");

stmt.executeUpdate(INSERT\_QUERY);

// INSERT one more row into Employees table

stmt.executeUpdate(INSERT\_QUERY\_2);

// Commit data here.

System.out.println("Commiting data here....");

conn.commit();

// Now list all the available records.

String sql = "SELECT id, first, last, age FROM Employees";

ResultSet rs = stmt.executeQuery(sql);

System.out.println("List result set for reference....");

printResultSet(rs);

// Clean-up environment

rs.close();

stmt.close();

conn.close();

}catch(SQLException se){

se.printStackTrace();

// If there is an error then rollback the changes.

System.out.println("Rolling back data here....");

try{

if(conn!=null)

conn.rollback();

}catch(SQLException se2){

se2.printStackTrace();

}

}catch(Exception e){

e.printStackTrace();

}finally{

// finally block used to close resources

try{

if(stmt!=null)

stmt.close();

}catch(SQLException se2){

se2.printStackTrace();

}

try{

if(conn!=null)

conn.close();

}catch(SQLException se){

se.printStackTrace();

}

}

}

}

Now let us compile the above example as follows −

C:\>javac TestApplication.java

C:\>

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

C:\>java TestApplication

Connecting to database...

Creating statement...

Inserting one row....

Commiting data here....

List result set for reference....

ID: 1, Age: 23, First: Zara, Last: Ali

ID: 2, Age: 30, First: Mahnaz, Last: Fatma

ID: 3, Age: 35, First: Zaid, Last: Khan

ID: 4, Age: 33, First: Sumit, Last: Mittal

ID: 5, Age: 40, First: John, Last: Paul

ID: 6, Age: 20, First: Rita, Last: Tez

ID: 7, Age: 20, First: Sita, Last: Singh

C:\>

Java & MySQL - SavePoint

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 −

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.

Following is the example, which makes use of **setSavepoint** and **rollback** described.

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

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

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.ResultSet;

import java.sql.SQLException;

import java.sql.Savepoint;

import java.sql.Statement;

public class TestApplication {

static final String DB\_URL = "jdbc:mysql://localhost/TUTORIALSPOINT";

static final String USER = "guest";

static final String PASS = "guest123";

static final String QUERY = "SELECT id, first, last, age FROM Employees";

static final String DELETE\_QUERY = "DELETE FROM Employees WHERE ID = 8";

static final String DELETE\_QUERY\_1 = "DELETE FROM Employees WHERE ID = 9";

public static void printResultSet(ResultSet rs) throws SQLException{

// Ensure we start with first row

rs.beforeFirst();

while(rs.next()){

// Display values

System.out.print("ID: " + rs.getInt("id"));

System.out.print(", Age: " + rs.getInt("age"));

System.out.print(", First: " + rs.getString("first"));

System.out.println(", Last: " + rs.getString("last"));

}

System.out.println();

}

public static void main(String[] args) {

// Open a connection

try(Connection conn = DriverManager.getConnection(DB\_URL, USER, PASS);

Statement stmt = conn.createStatement(

ResultSet.TYPE\_SCROLL\_INSENSITIVE,

ResultSet.CONCUR\_UPDATABLE);

) {

conn.setAutoCommit(false);

ResultSet rs = stmt.executeQuery(QUERY);

System.out.println("List result set for reference....");

printResultSet(rs);

// delete row having ID = 8

// But save point before doing so.

Savepoint savepoint1 = conn.setSavepoint("ROWS\_DELETED\_1");

System.out.println("Deleting row....");

stmt.executeUpdate(DELETE\_QUERY);

// Rollback the changes after save point 1.

conn.rollback(savepoint1);

// delete rows having ID = 9

// But save point before doing so.

conn.setSavepoint("ROWS\_DELETED\_2");

System.out.println("Deleting row....");

stmt.executeUpdate(DELETE\_QUERY\_1);

rs = stmt.executeQuery(QUERY);

System.out.println("List result set for reference....");

printResultSet(rs);

// Clean-up environment

rs.close();

} catch (SQLException e) {

e.printStackTrace();

}

}

}

Now, let us compile the above example as follows −

C:\>javac TestApplication.java

C:\>

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

C:\>java TestApplication

List result set for reference....

ID: 1, Age: 23, First: Zara, Last: Ali

ID: 2, Age: 30, First: Mahnaz, Last: Fatma

ID: 3, Age: 35, First: Zaid, Last: Khan

ID: 4, Age: 33, First: Sumit, Last: Mittal

ID: 5, Age: 40, First: John, Last: Paul

ID: 7, Age: 20, First: Sita, Last: Singh

ID: 8, Age: 20, First: Rita, Last: Tez

ID: 9, Age: 20, First: Sita, Last: Singh

Deleting row....

Deleting row....

List result set for reference....

ID: 1, Age: 23, First: Zara, Last: Ali

ID: 2, Age: 30, First: Mahnaz, Last: Fatma

ID: 3, Age: 35, First: Zaid, Last: Khan

ID: 4, Age: 33, First: Sumit, Last: Mittal

ID: 5, Age: 40, First: John, Last: Paul

ID: 7, Age: 20, First: Sita, Last: Singh

ID: 8, Age: 20, First: Rita, Last: Tez

C:\>

Java & MySQL - Batch Processing

Batch Processing allows you to group related SQL statements into a batch and submit them with one call to the database.

When you send several SQL statements to the database at once, you reduce the amount of communication overhead, thereby improving performance.

JDBC drivers are not required to support this feature. You should use the *DatabaseMetaData.supportsBatchUpdates()* method to determine if the target database supports batch update processing. The method returns true if your JDBC driver supports this feature.

The **addBatch()** method of *Statement, PreparedStatement,* and *CallableStatement* is used to add individual statements to the batch. The **executeBatch()** is used to start the execution of all the statements grouped together.

The **executeBatch()** returns an array of integers, and each element of the array represents the update count for the respective update statement.

Just as you can add statements to a batch for processing, you can remove them with the **clearBatch()** method. This method removes all the statements you added with the addBatch() method. However, you cannot selectively choose which statement to remove.

Batching with Statement Object

Here is a typical sequence of steps to use Batch Processing with Statement Object −

Create a Statement object using either *createStatement()* methods.

Set auto-commit to false using *setAutoCommit()*.

Add as many as SQL statements you like into batch using *addBatch()* method on created statement object.

Execute all the SQL statements using *executeBatch()* method on created statement object.

Finally, commit all the changes using *commit()* method.

Example

The following code snippet provides an example of a batch update using Statement object −

// Create statement object

Statement stmt = conn.createStatement();

// Set auto-commit to false

conn.setAutoCommit(false);

// Create SQL statement

String SQL = "INSERT INTO Employees (id, first, last, age) " +

"VALUES(200,'Zia', 'Ali', 30)";

// Add above SQL statement in the batch.

stmt.addBatch(SQL);

// Create one more SQL statement

String SQL = "INSERT INTO Employees (id, first, last, age) " +

"VALUES(201,'Raj', 'Kumar', 35)";

// Add above SQL statement in the batch.

stmt.addBatch(SQL);

// Create one more SQL statement

String SQL = "UPDATE Employees SET age = 35 " +

"WHERE id = 100";

// Add above SQL statement in the batch.

stmt.addBatch(SQL);

// Create an int[] to hold returned values

int[] count = stmt.executeBatch();

//Explicitly commit statements to apply changes

conn.commit();

Batching with PrepareStatement Object

Here is a typical sequence of steps to use Batch Processing with PrepareStatement Object −

Create SQL statements with placeholders.

Create PrepareStatement object using either *prepareStatement()* methods.

Set auto-commit to false using *setAutoCommit()*.

Add as many as SQL statements you like into batch using *addBatch()* method on created statement object.

Execute all the SQL statements using *executeBatch()* method on created statement object.

Finally, commit all the changes using *commit()* method.

The following code snippet provides an example of a batch update using PrepareStatement object −

// Create SQL statement

String SQL = "INSERT INTO Employees (id, first, last, age) " +

"VALUES(?, ?, ?, ?)";

// Create PrepareStatement object

PreparedStatemen pstmt = conn.prepareStatement(SQL);

//Set auto-commit to false

conn.setAutoCommit(false);

// Set the variables

pstmt.setInt( 1, 400 );

pstmt.setString( 2, "Pappu" );

pstmt.setString( 3, "Singh" );

pstmt.setInt( 4, 33 );

// Add it to the batch

pstmt.addBatch();

// Set the variables

pstmt.setInt( 1, 401 );

pstmt.setString( 2, "Pawan" );

pstmt.setString( 3, "Singh" );

pstmt.setInt( 4, 31 );

// Add it to the batch

pstmt.addBatch();

//add more batches

.

.

.

.

//Create an int[] to hold returned values

int[] count = stmt.executeBatch();

//Explicitly commit statements to apply changes

conn.commit();

Java & MySQL - Batching with Statement Object

Here is a typical sequence of steps to use Batch Processing with Statement Object −

Create a Statement object using either *createStatement()* methods.

Set auto-commit to false using *setAutoCommit()*.

Add as many as SQL statements you like into batch using *addBatch()* method on created statement object.

Execute all the SQL statements using *executeBatch()* method on created statement object.

Finally, commit all the changes using *commit()* method.

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

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

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.ResultSet;

import java.sql.SQLException;

import java.sql.Statement;

public class TestApplication {

static final String DB\_URL = "jdbc:mysql://localhost/TUTORIALSPOINT";

static final String USER = "guest";

static final String PASS = "guest123";

public static void printResultSet(ResultSet rs) throws SQLException{

// Ensure we start with first row

rs.beforeFirst();

while(rs.next()){

// Display values

System.out.print("ID: " + rs.getInt("id"));

System.out.print(", Age: " + rs.getInt("age"));

System.out.print(", First: " + rs.getString("first"));

System.out.println(", Last: " + rs.getString("last"));

}

System.out.println();

}

public static void main(String[] args) {

// Open a connection

try(Connection conn = DriverManager.getConnection(DB\_URL, USER, PASS);

Statement stmt = conn.createStatement(

ResultSet.TYPE\_SCROLL\_INSENSITIVE,

ResultSet.CONCUR\_UPDATABLE)

) {

conn.setAutoCommit(false);

ResultSet rs = stmt.executeQuery("Select \* from Employees");

printResultSet(rs);

// Create SQL statement

String SQL = "INSERT INTO Employees (first, last, age) " +

"VALUES('Zia', 'Ali', 30)";

// Add above SQL statement in the batch.

stmt.addBatch(SQL);

// Create one more SQL statement

SQL = "INSERT INTO Employees (first, last, age) " +

"VALUES('Raj', 'Kumar', 35)";

// Add above SQL statement in the batch.

stmt.addBatch(SQL);

// Create one more SQL statement

SQL = "UPDATE Employees SET age = 35 " +

"WHERE id = 7";

// Add above SQL statement in the batch.

stmt.addBatch(SQL);

// Create an int[] to hold returned values

int[] count = stmt.executeBatch();

//Explicitly commit statements to apply changes

conn.commit();

rs = stmt.executeQuery("Select \* from Employees");

printResultSet(rs);

stmt.close();

rs.close();

} catch (SQLException e) {

e.printStackTrace();

}

}

}

Now let us compile the above example as follows −

C:\>javac TestApplication.java

C:\>

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

C:\>java TestApplication

ID: 1, Age: 23, First: Zara, Last: Ali

ID: 2, Age: 30, First: Mahnaz, Last: Fatma

ID: 3, Age: 35, First: Zaid, Last: Khan

ID: 4, Age: 33, First: Sumit, Last: Mittal

ID: 5, Age: 40, First: John, Last: Paul

ID: 7, Age: 20, First: Sita, Last: Singh

ID: 8, Age: 20, First: Rita, Last: Tez

ID: 9, Age: 20, First: Sita, Last: Singh

ID: 1, Age: 23, First: Zara, Last: Ali

ID: 2, Age: 30, First: Mahnaz, Last: Fatma

ID: 3, Age: 35, First: Zaid, Last: Khan

ID: 4, Age: 33, First: Sumit, Last: Mittal

ID: 5, Age: 40, First: John, Last: Paul

ID: 7, Age: 35, First: Sita, Last: Singh

ID: 8, Age: 20, First: Rita, Last: Tez

ID: 9, Age: 20, First: Sita, Last: Singh

ID: 10, Age: 30, First: Zia, Last: Ali

ID: 11, Age: 35, First: Raj, Last: Kumar

C:\>

Java & MySQL - Batching with PrepareStatement Object

Here is a typical sequence of steps to use Batch Processing with PrepareStatement Object −

Create SQL statements with placeholders.

Create PrepareStatement object using either *prepareStatement()* methods.

Set auto-commit to false using *setAutoCommit()*.

Add as many as SQL statements you like into batch using *addBatch()* method on created statement object.

Execute all the SQL statements using *executeBatch()* method on created statement object.

Finally, commit all the changes using *commit()* method.

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

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

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.PreparedStatement;

import java.sql.ResultSet;

import java.sql.SQLException;

import java.sql.Statement;

public class TestApplication {

static final String DB\_URL = "jdbc:mysql://localhost/TUTORIALSPOINT";

static final String USER = "guest";

static final String PASS = "guest123";

static final String INSERT\_QUERY = "INSERT INTO Employees(first,last,age) VALUES(?, ?, ?)";

public static void printResultSet(ResultSet rs) throws SQLException{

// Ensure we start with first row

rs.beforeFirst();

while(rs.next()){

// Display values

System.out.print("ID: " + rs.getInt("id"));

System.out.print(", Age: " + rs.getInt("age"));

System.out.print(", First: " + rs.getString("first"));

System.out.println(", Last: " + rs.getString("last"));

}

System.out.println();

}

public static void main(String[] args) {

// Open a connection

try(Connection conn = DriverManager.getConnection(DB\_URL, USER, PASS);

PreparedStatement stmt = conn.prepareStatement(INSERT\_QUERY,

ResultSet.TYPE\_SCROLL\_INSENSITIVE,

ResultSet.CONCUR\_UPDATABLE)

) {

conn.setAutoCommit(false);

ResultSet rs = stmt.executeQuery("Select \* from Employees");

printResultSet(rs);

// Set the variables

stmt.setString( 1, "Pappu" );

stmt.setString( 2, "Singh" );

stmt.setInt( 3, 33 );

// Add it to the batch

stmt.addBatch();

// Set the variables

stmt.setString( 1, "Pawan" );

stmt.setString( 2, "Singh" );

stmt.setInt( 3, 31 );

// Add it to the batch

stmt.addBatch();

// Create an int[] to hold returned values

int[] count = stmt.executeBatch();

//Explicitly commit statements to apply changes

conn.commit();

rs = stmt.executeQuery("Select \* from Employees");

printResultSet(rs);

stmt.close();

rs.close();

} catch (SQLException e) {

e.printStackTrace();

}

}

}

Now let us compile the above example as follows −

C:\>javac TestApplication.java

C:\>

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

C:\>java TestApplication

ID: 1, Age: 23, First: Zara, Last: Ali

ID: 2, Age: 30, First: Mahnaz, Last: Fatma

ID: 3, Age: 35, First: Zaid, Last: Khan

ID: 4, Age: 33, First: Sumit, Last: Mittal

ID: 5, Age: 40, First: John, Last: Paul

ID: 7, Age: 35, First: Sita, Last: Singh

ID: 8, Age: 20, First: Rita, Last: Tez

ID: 9, Age: 20, First: Sita, Last: Singh

ID: 10, Age: 30, First: Zia, Last: Ali

ID: 11, Age: 35, First: Raj, Last: Kumar

ID: 1, Age: 23, First: Zara, Last: Ali

ID: 2, Age: 30, First: Mahnaz, Last: Fatma

ID: 3, Age: 35, First: Zaid, Last: Khan

ID: 4, Age: 33, First: Sumit, Last: Mittal

ID: 5, Age: 40, First: John, Last: Paul

ID: 7, Age: 35, First: Sita, Last: Singh

ID: 8, Age: 20, First: Rita, Last: Tez

ID: 9, Age: 20, First: Sita, Last: Singh

ID: 10, Age: 30, First: Zia, Last: Ali

ID: 11, Age: 35, First: Raj, Last: Kumar

ID: 12, Age: 33, First: Pappu, Last: Singh

ID: 13, Age: 31, First: Pawan, Last: Singh

C:\>

Java & MySQL - Streaming Data

A PreparedStatement object has the ability to use input and output streams to supply parameter data. This enables you to place entire files into database columns that can hold large values, such as CLOB and BLOB data types.

There are following methods, which can be used to stream data −

**setAsciiStream()** − This method is used to supply large ASCII values.

**setCharacterStream()** − This method is used to supply large UNICODE values.

**setBinaryStream()** − This method is used to supply large binary values.

The setXXXStream() method requires an extra parameter, the file size, besides the parameter placeholder. This parameter informs the driver how much data should be sent to the database using the stream.

This example would create a database table XML\_Data and then XML content would be written into this table.

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

import java.io.ByteArrayInputStream;

import java.io.ByteArrayOutputStream;

import java.io.File;

import java.io.FileInputStream;

import java.io.FileNotFoundException;

import java.io.IOException;

import java.io.InputStream;

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.PreparedStatement;

import java.sql.ResultSet;

import java.sql.SQLException;

import java.sql.Statement;

public class TestApplication {

static final String DB\_URL = "jdbc:mysql://localhost/TUTORIALSPOINT";

static final String USER = "guest";

static final String PASS = "guest123";

static final String QUERY = "SELECT Data FROM XML\_Data WHERE id=100";

static final String INSERT\_QUERY="INSERT INTO XML\_Data VALUES (?,?)";

static final String CREATE\_TABLE\_QUERY = "CREATE TABLE XML\_Data (id INTEGER, Data LONG)";

static final String DROP\_TABLE\_QUERY = "DROP TABLE XML\_Data";

static final String XML\_DATA = "<Employee><id>100</id><first>Zara</first><last>Ali</last><Salary>10000</Salary><Dob>18-08-1978</Dob></Employee>";

public static void createXMLTable(Statement stmt)

throws SQLException{

System.out.println("Creating XML\_Data table..." );

//Drop table first if it exists.

try{

stmt.executeUpdate(DROP\_TABLE\_QUERY);

}catch(SQLException se){

}

stmt.executeUpdate(CREATE\_TABLE\_QUERY);

}

public static void main(String[] args) {

// Open a connection

try(Connection conn = DriverManager.getConnection(DB\_URL, USER, PASS);

Statement stmt = conn.createStatement();

PreparedStatement pstmt = conn.prepareStatement(INSERT\_QUERY);

) {

createXMLTable(stmt);

ByteArrayInputStream bis = new ByteArrayInputStream(XML\_DATA.getBytes());

pstmt.setInt(1,100);

pstmt.setAsciiStream(2,bis,XML\_DATA.getBytes().length);

pstmt.execute();

//Close input stream

bis.close();

ResultSet rs = stmt.executeQuery(QUERY);

// Get the first row

if (rs.next ()){

//Retrieve data from input stream

InputStream xmlInputStream = rs.getAsciiStream (1);

int c;

ByteArrayOutputStream bos = new ByteArrayOutputStream();

while (( c = xmlInputStream.read ()) != -1)

bos.write(c);

//Print results

System.out.println(bos.toString());

}

// Clean-up environment

rs.close();

} catch (SQLException | IOException e) {

e.printStackTrace();

}

}

}

Now let us compile the above example as follows −

C:\>javac TestApplication.java

C:\>

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

C:\>java TestApplication

Creating XML\_Data table...

<Employee><id>100</id><first>Zara</first><last>Ali</last><Salary>10000</Salary><Dob>18-08-1978</Dob></Employee>

C:\>

Java & MySQL - Create Database Example

This tutorial provides an example on how to create a Database using JDBC application. Before executing the following example, make sure you have the following in place −

You should have admin privilege to create a database in the given schema. To execute the following example, you need to replace the *username* and *password* with your actual user name and password.

Your MySQL is up and running.

Required Steps

The following steps are required to create a new Database using 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.

**Open a connection** − Requires using the *DriverManager.getConnection()* method to create a Connection object, which represents a physical connection with the database server.

To create a new database, you need not give any database name while preparing database URL as mentioned in the below example.

**Execute a query** − Requires using an object of type Statement for building and submitting an SQL statement to the database.

**Clean up the environment** − try with resources automatically closes the resources.

Sample Code

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

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.SQLException;

import java.sql.Statement;

public class TestApplication {

static final String DB\_URL = "jdbc:mysql://localhost/";

static final String USER = "guest";

static final String PASS = "guest123";

public static void main(String[] args) {

// Open a connection

try(Connection conn = DriverManager.getConnection(DB\_URL, USER, PASS);

Statement stmt = conn.createStatement();

) {

String sql = "CREATE DATABASE STUDENTS";

stmt.executeUpdate(sql);

System.out.println("Database created successfully...");

} catch (SQLException e) {

e.printStackTrace();

}

}

}

Now let us compile the above example as follows −

C:\>javac TestApplication.java

C:\>

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

C:\>java TestApplication

Database created successfully...

C:\>

Java & MySQL - Select Database Example

This chapter provides an example on how to select a Database using JDBC application. Before executing the following example, make sure you have the following in place −

To execute the following example you need to replace the *username* and *password* with your actual user name and password.

Your MySQL database you are using, is up and running.

Required Steps

The following steps are required to create a new Database using JDBC application −

**Import the packages** − Requires that you include the packages containing the JDBC classes needed for the database programming. Most often, using *import java.sql.\** will suffice.

**Open a connection** − Requires using the *DriverManager.getConnection()* method to create a Connection object, which represents a physical connection with a **selected** database.

Selection of database is made while you prepare database URL. Following example would make connection with **STUDENTS** database.

**Clean up the environment** − try with resources automatically closes the resources.

Sample Code

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

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.SQLException;

import java.sql.Statement;

public class TestApplication {

static final String DB\_URL = "jdbc:mysql://localhost/TUTORIALSPOINT";

static final String USER = "guest";

static final String PASS = "guest123";

public static void main(String[] args) {

System.out.println("Connecting to a selected database...");

// Open a connection

try(Connection conn = DriverManager.getConnection(DB\_URL, USER, PASS);) {

System.out.println("Connected database successfully...");

} catch (SQLException e) {

e.printStackTrace();

}

}

}

Now let us compile the above example as follows −

C:\>javac TestApplication.java

C:\>

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

C:\>java TestApplication

Connecting to a selected database...

Connected database successfully...

C:\>

Java & MySQL - Drop Database Example

This chapter provides an example on how to drop an existing Database using JDBC application. Before executing the following example, make sure you have the following in place −

To execute the following example you need to replace the *username* and *password* with your actual user name and password.

Your MySQL is up and running.

**NOTE** − This is a serious operation and you have to make a firm decision before proceeding to delete a database because everything you have in your database would be lost.

Required Steps

The following steps are required to create a new Database using 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.

**Open a connection** − Requires using the *DriverManager.getConnection()* method to create a Connection object, which represents a physical connection with a database server.

Deleting a database does not require database name to be in your database URL. Following example would delete **STUDENTS** database.

**Execute a query** − Requires using an object of type Statement for building and submitting an SQL statement to delete the database.

**Clean up the environment** − try with resources automatically closes the resources.

Sample Code

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

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.SQLException;

import java.sql.Statement;

public class TestApplication {

static final String DB\_URL = "jdbc:mysql://localhost/";

static final String USER = "guest";

static final String PASS = "guest123";

public static void main(String[] args) {

// Open a connection

try(Connection conn = DriverManager.getConnection(DB\_URL, USER, PASS);

Statement stmt = conn.createStatement();

) {

String sql = "DROP DATABASE STUDENTS";

stmt.executeUpdate(sql);

System.out.println("Database dropped successfully...");

} catch (SQLException e) {

e.printStackTrace();

}

}

}

Now let us compile the above example as follows −

C:\>javac TestApplication.java

C:\>

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

C:\>java TestApplication

Database dropped successfully...

C:\>

Java & MySQL - Create Table Example

This chapter provides an example on how to create a table using JDBC application. Before executing the following example, make sure you have the following in place −

To execute the following example you can replace the *username* and *password* with your actual user name and password.

Your MySQL is up and running.

Required Steps

The following steps are required to create a new Database using 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.

**Open a connection** − Requires using the *DriverManager.getConnection()* method to create a Connection object, which represents a physical connection with a database server.

**Execute a query** − Requires using an object of type Statement for building and submitting an SQL statement to create a table in a seleted database.

**Clean up the environment** − try with resources automatically closes the resources.

Sample Code

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

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.SQLException;

import java.sql.Statement;

public class TestApplication {

static final String DB\_URL = "jdbc:mysql://localhost/TUTORIALSPOINT";

static final String USER = "guest";

static final String PASS = "guest123";

public static void main(String[] args) {

// Open a connection

try(Connection conn = DriverManager.getConnection(DB\_URL, USER, PASS);

Statement stmt = conn.createStatement();

) {

String sql = "CREATE TABLE REGISTRATION " +

"(id INTEGER not NULL, " +

" first VARCHAR(255), " +

" last VARCHAR(255), " +

" age INTEGER, " +

" PRIMARY KEY ( id ))";

stmt.executeUpdate(sql);

System.out.println("Created table in given database...");

} catch (SQLException e) {

e.printStackTrace();

}

}

}

Now let us compile the above example as follows −

C:\>javac TestApplication.java

C:\>

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

C:\>java TestApplication

Created table in given database...

C:\>

Java & MySQL - Drop Table Example

This chapter provides an example on how to delete a table using JDBC application. Before executing the following example, make sure you have the following in place −

To execute the following example you can replace the *username* and *password* with your actual user name and password.

Your MySQL database you are using, is up and running.

**NOTE** − This is a serious operation and you have to make a firm decision before proceeding to delete a table, because everything you have in your table would be lost.

Required Steps

The following steps are required to create a new Database using 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.

**Open a connection** − Requires using the *DriverManager.getConnection()* method to create a Connection object, which represents a physical connection with a database server.

**Execute a query** − Requires using an object of type Statement for building and submitting an SQL statement to drop a table in a seleted database.

**Clean up the environment** − try with resources automatically closes the resources.

Sample Code

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

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.SQLException;

import java.sql.Statement;

public class TestApplication {

static final String DB\_URL = "jdbc:mysql://localhost/TUTORIALSPOINT";

static final String USER = "guest";

static final String PASS = "guest123";

public static void main(String[] args) {

// Open a connection

try(Connection conn = DriverManager.getConnection(DB\_URL, USER, PASS);

Statement stmt = conn.createStatement();

) {

String sql = "DROP TABLE REGISTRATION";

stmt.executeUpdate(sql);

System.out.println("Table deleted in given database...");

} catch (SQLException e) {

e.printStackTrace();

}

}

}

Now let us compile the above example as follows −

C:\>javac TestApplication.java

C:\>

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

C:\>java TestApplication

Table deleted in given database...

C:\>

Java & MySQL - Insert Records Example

This chapter provides an example on how to insert records in a table using JDBC application. Before executing following example, make sure you have the following in place −

To execute the following example you can replace the *username* and *password* with your actual user name and password.

Your MySQL database you are using is up and running.

Required Steps

The following steps are required to create a new Database using 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 communications channel with the database.

**Open a connection** − Requires using the *DriverManager.getConnection()* method to create a Connection object, which represents a physical connection with a database server.

**Execute a query** − Requires using an object of type Statement for building and submitting an SQL statement to insert records into a table.

**Clean up the environment** − try with resources automatically closes the resources.

Sample Code

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

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.SQLException;

import java.sql.Statement;

public class TestApplication {

static final String DB\_URL = "jdbc:mysql://localhost/TUTORIALSPOINT";

static final String USER = "guest";

static final String PASS = "guest123";

public static void main(String[] args) {

// Open a connection

try(Connection conn = DriverManager.getConnection(DB\_URL, USER, PASS);

Statement stmt = conn.createStatement();

) {

// Execute a query

System.out.println("Inserting records into the table...");

String sql = "INSERT INTO Registration VALUES (100, 'Zara', 'Ali', 18)";

stmt.executeUpdate(sql);

sql = "INSERT INTO Registration VALUES (101, 'Mahnaz', 'Fatma', 25)";

stmt.executeUpdate(sql);

sql = "INSERT INTO Registration VALUES (102, 'Zaid', 'Khan', 30)";

stmt.executeUpdate(sql);

sql = "INSERT INTO Registration VALUES(103, 'Sumit', 'Mittal', 28)";

stmt.executeUpdate(sql);

System.out.println("Inserted records into the table...");

} catch (SQLException e) {

e.printStackTrace();

}

}

}

Now let us compile the above example as follows −

C:\>javac TestApplication.java

C:\>

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

C:\>java TestApplication

Inserting records into the table...

Inserted records into the table...

C:\>

Java & MySQL - Select Records Example

This chapter provides an example on how to select/ fetch records from a table using JDBC application. Before executing the following example, make sure you have the following in place −

To execute the following example you can replace the *username* and *password* with your actual user name and password.

Your MySQL database you are using is up and running.

Required Steps

The following steps are required to create a new Database using 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.

**Open a connection** − Requires using the *DriverManager.getConnection()* method to create a Connection object, which represents a physical connection with a database server.

**Execute a query** − Requires using an object of type Statement for building and submitting an SQL statement to select (i.e. fetch ) records from a table.

**Extract Data** − Once SQL query is executed, you can fetch records from the table.

**Clean up the environment** − try with resources automatically closes the resources.

Sample Code

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

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.ResultSet;

import java.sql.SQLException;

import java.sql.Statement;

public class TestApplication {

static final String DB\_URL = "jdbc:mysql://localhost/TUTORIALSPOINT";

static final String USER = "guest";

static final String PASS = "guest123";

static final String QUERY = "SELECT id, first, last, age FROM Registration";

public static void main(String[] args) {

// Open a connection

try(Connection conn = DriverManager.getConnection(DB\_URL, USER, PASS);

Statement stmt = conn.createStatement();

ResultSet rs = stmt.executeQuery(QUERY);

) {

while(rs.next()){

//Display values

System.out.print("ID: " + rs.getInt("id"));

System.out.print(", Age: " + rs.getInt("age"));

System.out.print(", First: " + rs.getString("first"));

System.out.println(", Last: " + rs.getString("last"));

}

} catch (SQLException e) {

e.printStackTrace();

}

}

}

Now let us compile the above example as follows −

C:\>javac TestApplication.java

C:\>

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

C:\>java TestApplication

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:\>

Java & MySQL - Update Records Example

This chapter provides an example on how to update records in a table using JDBC application. Before executing the following example, make sure you have the following in place −

To execute the following example you can replace the *username* and *password* with your actual user name and password.

Your MySQL database you are using is up and running.

Required Steps

The following steps are required to create a new Database using 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.

**Open a connection** − Requires using the *DriverManager.getConnection()* method to create a Connection object, which represents a physical connection with a database server.

**Execute a query** − Requires using an object of type Statement for building and submitting an SQL statement to update records in a table. This Query makes use of **IN** and **WHERE** clause to update conditional records.

**Clean up the environment** − try with resources automatically closes the resources.

Sample Code

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

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.ResultSet;

import java.sql.SQLException;

import java.sql.Statement;

public class TestApplication {

static final String DB\_URL = "jdbc:mysql://localhost/TUTORIALSPOINT";

static final String USER = "guest";

static final String PASS = "guest123";

static final String QUERY = "SELECT id, first, last, age FROM Registration";

public static void main(String[] args) {

// Open a connection

try(Connection conn = DriverManager.getConnection(DB\_URL, USER, PASS);

Statement stmt = conn.createStatement();

) {

String sql = "UPDATE Registration " +

"SET age = 30 WHERE id in (100, 101)";

stmt.executeUpdate(sql);

ResultSet rs = stmt.executeQuery(QUERY);

while(rs.next()){

//Display values

System.out.print("ID: " + rs.getInt("id"));

System.out.print(", Age: " + rs.getInt("age"));

System.out.print(", First: " + rs.getString("first"));

System.out.println(", Last: " + rs.getString("last"));

}

rs.close();

} catch (SQLException e) {

e.printStackTrace();

}

}

}

Now let us compile the above example as follows −

C:\>javac TestApplication.java

C:\>

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

C:\>java TestApplication

ID: 100, Age: 30, First: Zara, Last: Ali

ID: 101, Age: 30, First: Mahnaz, Last: Fatma

ID: 102, Age: 30, First: Zaid, Last: Khan

ID: 103, Age: 28, First: Sumit, Last: Mittal

C:\>

Java & MySQL - Delete Records Example

This chapter provides an example on how to delete records from a table using JDBC application. Before executing following example, make sure you have the following in place −

To execute the following example you can replace the *username* and *password* with your actual user name and password.

Your MySQL database you are using is up and running.

Required Steps

The following steps are required to create a new Database using 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 communications channel with the database.

**Open a connection** − Requires using the *DriverManager.getConnection()* method to create a Connection object, which represents a physical connection with a database server.

**Execute a query** − Requires using an object of type Statement for building and submitting an SQL statement to delete records from a table. This Query makes use of the **WHERE** clause to delete conditional records.

**Clean up the environment** − try with resources automatically closes the resources.

Sample Code

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

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.ResultSet;

import java.sql.SQLException;

import java.sql.Statement;

public class TestApplication {

static final String DB\_URL = "jdbc:mysql://localhost/TUTORIALSPOINT";

static final String USER = "guest";

static final String PASS = "guest123";

static final String QUERY = "SELECT id, first, last, age FROM Registration";

public static void main(String[] args) {

// Open a connection

try(Connection conn = DriverManager.getConnection(DB\_URL, USER, PASS);

Statement stmt = conn.createStatement();

) {

String sql = "DELETE FROM Registration " +

"WHERE id = 101";

stmt.executeUpdate(sql);

ResultSet rs = stmt.executeQuery(QUERY);

while(rs.next()){

//Display values

System.out.print("ID: " + rs.getInt("id"));

System.out.print(", Age: " + rs.getInt("age"));

System.out.print(", First: " + rs.getString("first"));

System.out.println(", Last: " + rs.getString("last"));

}

rs.close();

} catch (SQLException e) {

e.printStackTrace();

}

}

}

Now let us compile the above example as follows −

C:\>javac TestApplication.java

C:\>

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

C:\>java TestApplication

ID: 100, Age: 30, First: Zara, Last: Ali

ID: 102, Age: 30, First: Zaid, Last: Khan

ID: 103, Age: 28, First: Sumit, Last: Mittal

C:\>

Java & MySQL - WHERE Clause Example

This chapter provides an example on how to select records from a table using JDBC application. This would add additional conditions using WHERE clause while selecting records from the table. Before executing the following example, make sure you have the following in place −

To execute the following example you can replace the *username* and *password* with your actual user name and password.

Your MySQL database you are using, is up and running.

Required Steps

The following steps are required to create a new Database using JDBC application −

**Import the packages** − Requires that you include the packages containing the JDBC classes needed for the 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 communications channel with the database.

**Open a connection** − Requires using the *DriverManager.getConnection()* method to create a Connection object, which represents a physical connection with a database server.

**Execute a query** − Requires using an object of type Statement for building and submitting an SQL statement to fetch records from a table, which meet the given condition. This Query makes use of the **WHERE** clause to select records.

**Clean up the environment** − try with resources automatically closes the resources.

Sample Code

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

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.ResultSet;

import java.sql.SQLException;

import java.sql.Statement;

public class TestApplication {

static final String DB\_URL = "jdbc:mysql://localhost/TUTORIALSPOINT";

static final String USER = "guest";

static final String PASS = "guest123";

static final String QUERY = "SELECT id, first, last, age FROM Registration";

public static void main(String[] args) {

// Open a connection

try(Connection conn = DriverManager.getConnection(DB\_URL, USER, PASS);

Statement stmt = conn.createStatement();) {

System.out.println("Fetching records without condition...");

ResultSet rs = stmt.executeQuery(QUERY);

while(rs.next()){

//Display values

System.out.print("ID: " + rs.getInt("id"));

System.out.print(", Age: " + rs.getInt("age"));

System.out.print(", First: " + rs.getString("first"));

System.out.println(", Last: " + rs.getString("last"));

}

// Select all records having ID equal or greater than 101

System.out.println("Fetching records with condition...");

String sql = "SELECT id, first, last, age FROM Registration" +

" WHERE id >= 101 ";

rs = stmt.executeQuery(sql);

while(rs.next()){

//Display values

System.out.print("ID: " + rs.getInt("id"));

System.out.print(", Age: " + rs.getInt("age"));

System.out.print(", First: " + rs.getString("first"));

System.out.println(", Last: " + rs.getString("last"));

}

rs.close();

} catch (SQLException e) {

e.printStackTrace();

}

}

}

Now let us compile the above example as follows −

C:\>javac TestApplication.java

C:\>

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

C:\>java TestApplication

Fetching records without condition...

ID: 100, Age: 30, First: Zara, Last: Ali

ID: 102, Age: 30, First: Zaid, Last: Khan

ID: 103, Age: 28, First: Sumit, Last: Mittal

Fetching records with condition...

ID: 102, Age: 30, First: Zaid, Last: Khan

ID: 103, Age: 28, First: Sumit, Last: Mittal

C:\>

Java & MySQL - LIKE Clause Example

This chapter provides an example on how to select records from a table using JDBC application. This would add additional conditions using LIKE clause while selecting records from the table. Before executing the following example, make sure you have the following in place −

To execute the following example you can replace the *username* and *password* with your actual user name and password.

Your MySQL database you are using, is up and running.

Required Steps

The following steps are required to create a new Database using 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.

**Open a connection** − Requires using the *DriverManager.getConnection()* method to create a Connection object, which represents a physical connection with a database server.

**Execute a query** − Requires using an object of type Statement for building and submitting an SQL statement to fetch records from a table which meet given condition. This Query makes use of **LIKE** clause to select records to select all the students whose first name starts with "za".

**Clean up the environment** − try with resources automatically closes the resources.

Sample Code

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

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.ResultSet;

import java.sql.SQLException;

import java.sql.Statement;

public class TestApplication {

static final String DB\_URL = "jdbc:mysql://localhost/TUTORIALSPOINT";

static final String USER = "guest";

static final String PASS = "guest123";

static final String QUERY = "SELECT id, first, last, age FROM Registration";

public static void main(String[] args) {

// Open a connection

try(Connection conn = DriverManager.getConnection(DB\_URL, USER, PASS);

Statement stmt = conn.createStatement();) {

System.out.println("Fetching records without condition...");

ResultSet rs = stmt.executeQuery(QUERY);

while(rs.next()){

//Display values

System.out.print("ID: " + rs.getInt("id"));

System.out.print(", Age: " + rs.getInt("age"));

System.out.print(", First: " + rs.getString("first"));

System.out.println(", Last: " + rs.getString("last"));

}

// Select all records having ID equal or greater than 101

System.out.println("Fetching records with condition...");

String sql = "SELECT id, first, last, age FROM Registration" +

" WHERE first LIKE '%za%'";

rs = stmt.executeQuery(sql);

while(rs.next()){

//Display values

System.out.print("ID: " + rs.getInt("id"));

System.out.print(", Age: " + rs.getInt("age"));

System.out.print(", First: " + rs.getString("first"));

System.out.println(", Last: " + rs.getString("last"));

}

rs.close();

} catch (SQLException e) {

e.printStackTrace();

}

}

}

Now let us compile the above example as follows −

C:\>javac TestApplication.java

C:\>

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

C:\>java TestApplication

Fetching records without condition...

ID: 100, Age: 30, First: Zara, Last: Ali

ID: 102, Age: 30, First: Zaid, Last: Khan

ID: 103, Age: 28, First: Sumit, Last: Mittal

Fetching records with condition...

ID: 100, Age: 30, First: Zara, Last: Ali

ID: 102, Age: 30, First: Zaid, Last: Khan

C:\>

Java & MySQL - Sorting Data Example

This chapter provides an example on how to sort records from a table using JDBC application. This would use **asc** and **desc** keywords to sort records in ascending or descending order. Before executing the following example, make sure you have the following in place −

To execute the following example you can replace the *username* and *password* with your actual user name and password.

Your MySQL database you are using, is up and running.

Required Steps

The following steps are required to create a new Database using 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.

**Open a connection** − Requires using the *DriverManager.getConnection()* method to create a Connection object, which represents a physical connection with a database server.

**Execute a query** − Requires using an object of type Statement for building and submitting an SQL statement to sort records from a table. These Queries make use of **asc** and **desc** clauses to sort data in ascending and descening orders.

**Clean up the environment** − try with resources automatically closes the resources.

Sample Code

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

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.ResultSet;

import java.sql.SQLException;

import java.sql.Statement;

public class TestApplication {

static final String DB\_URL = "jdbc:mysql://localhost/TUTORIALSPOINT";

static final String USER = "guest";

static final String PASS = "guest123";

static final String QUERY = "SELECT id, first, last, age FROM Registration";

public static void main(String[] args) {

// Open a connection

try(Connection conn = DriverManager.getConnection(DB\_URL, USER, PASS);

Statement stmt = conn.createStatement();) {

System.out.println("Fetching records in ascending order...");

ResultSet rs = stmt.executeQuery(QUERY + " ORDER BY first ASC");

while(rs.next()){

//Display values

System.out.print("ID: " + rs.getInt("id"));

System.out.print(", Age: " + rs.getInt("age"));

System.out.print(", First: " + rs.getString("first"));

System.out.println(", Last: " + rs.getString("last"));

}

System.out.println("Fetching records in descending order...");

rs = stmt.executeQuery(QUERY + " ORDER BY first DESC");

while(rs.next()){

//Display values

System.out.print("ID: " + rs.getInt("id"));

System.out.print(", Age: " + rs.getInt("age"));

System.out.print(", First: " + rs.getString("first"));

System.out.println(", Last: " + rs.getString("last"));

}

rs.close();

} catch (SQLException e) {

e.printStackTrace();

}

}

}

Now let us compile the above example as follows −

C:\>javac TestApplication.java

C:\>

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

C:\>java TestApplication

Fetching records in ascending order...

ID: 103, Age: 28, First: Sumit, Last: Mittal

ID: 102, Age: 30, First: Zaid, Last: Khan

ID: 100, Age: 30, First: Zara, Last: Ali

Fetching records in descending order...

ID: 100, Age: 30, First: Zara, Last: Ali

ID: 102, Age: 30, First: Zaid, Last: Khan

ID: 103, Age: 28, First: Sumit, Last: Mittal