Creating Applications Using the

Consider a scenario where New Publishers, a publishing house, maintains details about books and authors in a database. The management of New Publishers wants an application that can help them access the details about authors based on different criteria. For example, the application should be able to retrieve the details of all the authors living in a particular city specified at runtime. In this scenario, you cannot use the `Statement` object to retrieve the details because the value for the city needs to be specified at runtime. You need to use the `PreparedStatement` object as it can accept runtime parameters.

The `PreparedStatement` interface is derived from the `Statement` interface and is available in the `java.sql` package. The `PreparedStatement` object allows you to pass runtime parameters to the SQL statements to query and modify the data in a table.

The `PreparedStatement` objects are compiled and prepared only once by JDBC. The further invocation of the `PreparedStatement` object does not recompile the SQL statements. This helps in reducing the load on the database server and improves the performance of the application.

Methods of the PreparedStatement Interface

The `PreparedStatement` interface inherits the following methods to execute the SQL statements from the `Statement` interface:

- `ResultSet executeQuery()`: Is used to execute the SQL statement and returns the result in a `ResultSet` object.
- `int executeUpdate()`: Executes an SQL statement, such as `INSERT`, `UPDATE`, or `DELETE`, and returns the count of the rows affected.
- `boolean execute()`: Executes an SQL statement and returns the `boolean` value.

Consider an example where you have to retrieve the details of an author by passing the author id at runtime. For this, the following SQL statement with a parameterized query can be used:

```
SELECT * FROM Authors WHERE au_id = ?
```

To submit such a parameterized query to a database from an application, you need to create a `PreparedStatement` object using the `prepareStatement()` method of the `Connection` object. You can use the following code snippet to prepare an SQL statement that accepts values at runtime:

```
stat=con.prepareStatement("SELECT * FROM Authors WHERE au_id = ?");
```

The `prepareStatement()` method of the `Connection` object takes an SQL statement as a parameter. The SQL statement can contain the symbol, `?`, as a placeholder that can be replaced by input parameters at runtime.

Before the SQL statement specified in the `PreparedStatement` object is executed, you must set the value of each `?` parameter. The value can be set by calling an appropriate `setXXX()` method, where `XXX` is the data type of the parameter. For example, consider the following code snippet:

```
stat.setString(1,"A001");
```

```
ResultSet result=stat.executeQuery();
```

In the preceding code snippet, the first parameter of the `setString()` method specifies the index value of the `?` placeholder, and the second parameter is used to set the value of the `?` placeholder. You can use the following code snippet when the value for the `?` placeholder is obtained from the user interface:

```
stat.setString(1,aid.getText());
ResultSet result=stat.executeQuery();
```

In the preceding code snippet, the `setString()` method is used to set the value of the `?` placeholder with the value retrieved at runtime from the `aid` textbox of the user interface.

The `PreparedStatement` interface provides various methods to set the value of placeholders for the specific data types. The following table lists some commonly used methods of the `PreparedStatement` interface.

Method	***Description***
`void setByte(int index, byte val)`	*Sets the Java `byte` type value for the parameter corresponding to the specified index.*
`void setBytes(int index, byte[] val)`	*Sets the Java `byte` type array for the parameter corresponding to the specified index.*
`void setBoolean(int index, boolean val)`	*Sets the Java `boolean` type value for the parameter corresponding to the specified index.*
`void setDouble(int index, double val)`	*Sets the Java `double` type value for the parameter corresponding to the specified index.*
`void setInt(int index, int val)`	*Sets the Java `int` type value for the parameter corresponding to the specified index.*
`void setLong(int index, long val)`	*Sets the Java `long` type value for the parameter corresponding to the specified index.*
`void setFloat(int index, float val)`	*Sets the Java `float` type value for the parameter corresponding to the specified index.*
`void setShort(int index, short val)`	*Sets the Java `short` type value for the parameter corresponding to the specified index.*
`void setString(int index, String val)`	*Sets the Java `String` type value for the parameter corresponding to the specified index.*

The Methods of the PreparedStatement Interface

Retrieving Rows

You can use the following code snippet to retrieve the details of the books written by an author from the Books table by using the PreparedStatement object:

```
String str = "SELECT * FROM Books WHERE au_id = ?";
PreparedStatement ps= con.prepareStatement(str);
ps.setString(1, "A001");
ResultSet rs=ps.executeQuery();
while(rs.next())
{
     System.out.println(rs.getString(1) + " " + rs.getString(2));
}
```

In the preceding code snippet, the str variable stores the SELECT statement that contains one input parameter. The setString() method is used to set the value for the au_id attribute of the Books table. The SELECT statement is executed using the executeQuery() method, which returns a ResultSet object.

Inserting Rows

You can use the following code snippet to create a PreparedStatement object that inserts a row into the Authors table by passing the author's data at runtime:

```
String str = "INSERT INTO Authors (au_id, au_name) VALUES (?,?)";
PreparedStatement ps = con.prepareStatement(str);
ps.setString(1, "1001");
ps.setString(2, "Abraham White");
int rt=ps.executeUpdate();
```

In the preceding code snippet, the str variable stores the INSERT statement that contains two input parameters. The setString() method is used to set the values for the au_id and au_name columns of the Authors table. The INSERT statement is executed using the executeUpdate() method, which returns an integer value that specifies the number of rows inserted into the table.

Updating and Deleting Rows

You can use the following code snippet to modify the state to CA where city is Oakland in the Authors table by using the PreparedStatement object:

```
String str = "UPDATE Authors SET state= ? WHERE city= ? ";
PreparedStatement ps = con.prepareStatement(str);
ps.setString(1, "CA");
ps.setString(2, "Oakland");
int rt=ps.executeUpdate();
```

In the preceding code snippet, two input parameters, state and city, contain the values for the state and city attributes of the Authors table, respectively.

You can use the following code snippet to delete a row from the `Authors` table, where `au_name` is `Abraham White` by using the `PreparedStatement` object:

```
String str = "DELETE FROM Authors WHERE au_name= ? ";
PreparedStatement ps = con.prepareStatement(str);
ps.setString(1, "Abraham White");
int rt=ps.executeUpdate();
```

Just a

Identity the three methods of the `PreparedStatement` interface.

Answer:

The three methods of the `PreparedStatement` interface are:

1. *`ResultSet executeQuery()`*
2. *`int executeUpdate()`*
3. *`boolean execute()`*

Managing Database Transactions

A transaction is a set of one or more SQL statements executed as a single unit. A transaction is complete only when all the SQL statements in a transaction execute successfully. If any one of the SQL statements in the transaction fails, the entire transaction is rolled back, thereby, maintaining the consistency of the data in the database.

JDBC API provides the support for transaction management. For example, a JDBC application is used to transfer money from one bank account to another. This transaction gets completed when the money is deducted from the first account and added to the second. If an error occurs while processing the SQL statements, both the accounts remain unchanged. The set of the SQL statements, which transfers money from one account to another, represents a transaction in the JDBC application.

The database transactions can be committed in the following two ways in the JDBC applications:

- **Implicit**: The `Connection` object uses the *auto-commit* mode to execute the SQL statements implicitly. The auto-commit mode specifies that each SQL statement in a transaction is committed automatically as soon as the execution of the SQL statement completes. By default, all the transaction statements in a JDBC application are auto-committed.
- **Explicit**: For explicitly committing a transaction statement in a JDBC application, you need to use the `setAutoCommit()` method. This method accepts either of the two values, `true` or `false`, to set or reset the auto-commit mode for a database. The auto-commit mode is set to `false` to commit a transaction explicitly. You can set the auto-commit mode to `false` using the following code snippet:

  ```
  con.setAutoCommit(false);
  ```

  In the preceding code snippet, `con` represents a `Connection` object.

Committing a Transaction

When you set the auto-commit mode to `false`, the operations performed by the SQL statements are not reflected permanently in a database. You need to explicitly call the `commit()` method of the `Connection` interface to reflect the changes made by the transactions in a database. All the SQL statements that appear between the `setAutoCommit(false)` method and the `commit()` method are treated as a single transaction and executed as a single unit.

The `rollback()` method is used to undo the changes made in the database after the last commit operation. You need to explicitly invoke the `rollback()` method to revert a database in the last committed state. When the `rollback()` method is invoked, all the pending transactions of a database are cancelled and the database gets reverted to the state in which it was committed previously. You can call the `rollback()` method using the following code snippet:

```
    con.rollback();
```

In the preceding code snippet, `con` represents a `Connection` object.

You can use the following code to create a transaction that includes two `INSERT` statements and the transaction is committed explicitly using the `commit()` method:

```
import java.sql.*;
public class CreateTrans
{
    public static void main(String arg[])
    {
        try
        {
            /*Initialize and load the Type 4 JDBC driver*/
            Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver");

            /*Establish a connection with a data source*/
            try (Connection con =
DriverManager.getConnection("jdbc:sqlserver://sqlserver01;databaseName=Librar
y;user=user1;password=password#1234");)
            {
                /*Set the auto commit mode to false*/
                con.setAutoCommit(false);

                /*Create a transaction*/
                try (PreparedStatement ps = con.prepareStatement("INSERT INTO
Publishers (pub_id, pub_name) VALUES (?, ?)");)
                {
                    /*Specify the value for the placeholders in the
PreparedStatement object*/
                    ps.setString(1, "P006");
                    ps.setString(2, "Darwin Press");
                    int firstctr = ps.executeUpdate();
                    System.out.println("First Row Inserted but not
committed");

                    /*Insert a row in the database table using the
prepareStatement() method*/
                    try (PreparedStatement ps2 = con.prepareStatement("INSERT
INTO Publishers(pub_id, pub_name) VALUES (?, ?)");)
                    {
                        ps2.setString(1, "P007");
                        ps2.setString(2, "MainStream Publishing");
                        int secondctr = ps2.executeUpdate();
                        System.out.println("Second Row Inserted but not
committed");

                        /*Commit a transaction*/
                        con.commit();
                        System.out.println("Transaction Committed, Please
check table for data");

                    }
                }
            }
        }
        catch (Exception e)
        {
            System.out.println("Error : " + e);
        }
    }
}
```

In the preceding code, the auto-commit mode is set to `false` using the `setAutoCommit()` method. All the statements that are executed after setting the auto-commit mode to `false` are treated as a transaction by the database.

Just a

How can you commit a transaction explicitly?

Answer:

You can commit a transaction explicitly by setting the auto-commit mode to `false` and using the `commit()` method.

Implementing Batch Updates in JDBC

A *batch* is a group of update statements sent to a database to be executed as a single unit. You send the batch to a database in a single request using the same `Connection` object. This reduces network calls between the application and the database. Therefore, processing multiple SQL statements in a batch is a more efficient way as compared to processing a single SQL statement.

A JDBC transaction can consist of multiple batches.

The `Statement` interface provides following methods to create and execute a batch of SQL statements:

- `void addBatch(String sql)`: Adds an SQL statement to a batch.
- `int[] executeBatch()`: Sends a batch to a database for processing and returns the total number of rows updated.
- `void clearBatch()`: Removes the SQL statements from the batch.

You can create a `Statement` object to perform batch updates. When the `Statement` object is created, an empty array gets associated with the object. You can add multiple SQL statements to the empty array for executing them as a batch. You also need to disable the auto-commit mode using the `setAutoCommit(false)` method while working with batch updates in JDBC. This enables you to roll back the entire transaction performed using a batch of updates if any SQL statement in the batch fails. You can use the following code snippet to create a batch of SQL statements:

```
con.setAutoCommit(false);
Statement stmt=con.createStatement();
stmt.addBatch("INSERT INTO Publishers (pub_id, pub_name) VALUES (P001, 'Sage
Publications')");
stmt.addBatch("INSERT INTO Product (pub_id, pub_name) VALUES (P002, 'Prisidio
Press')");
```

In the preceding code snippet, `con` is a `Connection` object. The `setAutoCommit()` method is used to set the auto-commit mode to `false`. The batch contains two `INSERT` statements that are added to the batch using the `addBatch()` method.

The SQL statements in a batch are processed in the order in which the statements appear in a batch. You can use the following code snippet to execute a batch of SQL statements:

```
int[] updcount=stmt.executeBatch();
```

In the preceding code snippet, `updcount` is an integer array that stores the values of the update count returned by the `executeBatch()` method. The update count is the total number of rows affected when an SQL statement is processed. The `executeBatch()` method returns the updated count for each SQL statement in a batch, if it is successfully processed.

Exception Handling in Batch Updates

The batch update operations can throw two types of exceptions, `SQLException` and `BatchUpdateException`. The JDBC API methods, `addBatch()` and `executeBatch()`, throw `SQLException` when any problem occurs while accessing a database. The `SQLException` exception is thrown when you try to execute a `SELECT` statement using the `executeBatch()` method. The `BatchUpdateException` class is derived from the `SQLException` class. The `BatchUpdateException` exception is thrown when the SQL statements in the batch cannot be executed due to:

- Presence of illegal arguments in the SQL statement.
- Absence of the database table.

`BatchUpdateException` uses an array of the update count to identify the SQL statement that throws the exception. The update count for all the SQL statements that are executed successfully is stored in the array in the same order in which the SQL statements appear in the batch. You can traverse the array of the update count to determine the SQL statement that is not executed successfully in a batch. The `null` value in an array of the update count indicates that the SQL statement in the batch failed to execute. You can use the following code to use the methods of the `SQLException` class that can be used to print the update counts for the SQL statements in a batch by using the `BatchUpdateException` object:

```
import java.sql.*;

public class BatchUpdate
{
    public static void main(String args[])
    {
        try
        {
            /*Batch Update Code comes here*/
        }
        catch (BatchUpdateException bexp)
        {
            /*Use the getMessage() method to retrieve the message associated
with the exception thrown*/
            System.err.println("SQL Exception:" + bexp.getMessage());
            System.err.println("Update Counts:");
            /*Use the getUpdateCount() method to retrieve the update count
for each SQL statement in a batch*/
            int[] updcount = bexp.getUpdateCounts();
            for (int i = 0; i <= updcount.length; i++)
            {
                /*Print the update count*/
                System.err.println(updcount[i]);
            }
        }
    }
}
```

Creating an Application to Insert Rows in a Table Using Batch Updates

You can execute multiple objects of the `Statement` and `PreparedStatement` interfaces together as batches in a JDBC application using batch updates. You can use the following code to insert data in a table using batch updates:

```
import java.sql.*;

public class BookInfo
{
    public static void main(String args[])
    {
        try
        {
            /*Initialize and load Type 4 JDBC driver*/
            Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver");

            /*Connect to a data source using Library DSN*/
            try (Connection con =
DriverManager.getConnection("jdbc:sqlserver://sqlserver01;databaseName=Librar
y;user=user1;password=password#1234");
                    /*Create a Statement object*/
                    Statement stmt = con.createStatement();)
            {
                con.setAutoCommit(false);

                /*Add the INSERT statements to a batch*/
                stmt.addBatch("INSERT INTO Books (book_id, book_name) VALUES
('B004', 'Kane and Able')");
                stmt.addBatch("INSERT INTO Books (book_id, book_name) VALUES
('B005', 'The Ghost')");
                stmt.addBatch("INSERT INTO Books (book_id, book_name) VALUES
('B006', 'If Tommorrow Comes')");

                /*Execute a batch using executeBatch() method*/
                int[] results = stmt.executeBatch();
                System.out.println("");
                System.out.println("Using Statement object");
                System.out.println("-------------------------");
                for (int i = 0; i < results.length; i++)
                {
                    System.out.println("Rows affected by " + (i + 1) + "
INSERT statement: " + results[i]);
                }

                /*Use the PreparedStatement object to perform batch updates*/
                try (PreparedStatement ps = con.prepareStatement("INSERT INTO
Books (book_id, price) VALUES ( ?, ?)");)
                {
                    System.out.println("");
                    System.out.println("---------------------------------");
                    System.out.println("Using PreparedStatement object");
                    System.out.println("---------------------------------");

                    /*Specify the value for the placeholders*/
```

```
                              ps.setString(1, "B007");
                              ps.setInt(2, 575);
                              ps.addBatch();
                              ps.setString(1, "B008");
                              ps.setInt(2, 350);

                              /*Add the SQL statement to the batch*/
                              ps.addBatch();

                              /*Execute the batch of SQL statements*/
                              int[] numUpdates = ps.executeBatch();
                              for (int i = 0; i < numUpdates.length; i++)
                              {
                                  System.err.println("Rows affected by " + (i + 1) + "
    INSERT statement: " + numUpdates[i]);
                              }

                              /*Commit the INSERT statements in the batch*/
                              con.commit();


                          }
                      }
                  }
                  catch (BatchUpdateException bue)
                  {
                      System.out.println("Error : " + bue);
                  }
                  catch (SQLException sqle)
                  {
                      System.out.println("Error : " + sqle);
                  }
                  catch (Exception e) {
                      System.out.println("Error : " + e);
                  }
              }
          }
```

In the preceding code, two batches are created using the `Statement` and `PreparedStatement` objects. The `INSERT` statements are added to the batch using the `addBatch()` method and are executed using the `executeBatch()` method. The `executeBatch()` method returns an array, which stores the update count for all the SQL statements in the batch. You can display the number of rows affected by each SQL statement in the batch using the `for` loop.

Creating and Calling Stored Procedures in

The `java.sql` package provides the `CallableStatement` interface that contains various methods to enable you to call database stored procedures. The `CallableStatement` interface is derived from the `PreparedStatement` interface.

Creating Stored Procedures

Stored procedures can be created using JDBC applications. You can use the `executeUpdate()` method to execute the `CREATE PROCEDURE` SQL statement. Stored procedures can be of two types, parameterized and non parameterized.

You can use the following code snippet to create a non parameterized stored procedure in a JDBC application:

```
String str = "CREATE PROCEDURE Authors_info "
+"AS "
+ "SELECT au_id,au_name "
+ "FROM Authors "
+ "WHERE city = 'Oakland' "
+ "ORDER BY au_name";
Statement stmt=con.createStatement();
int rt=stmt.executeUpdate(str);
```

In the preceding code snippet, the `SELECT` statement specifies that the data is retrieved from the `Authors` table in the order of the `au_name` column. The `Connection` object `con` is used to send the `CREATE PROCEDURE` SQL statement to a database. When you execute the code, the `Authors_info` stored procedure is created and stored in the database.

You can use the following code snippet to create a parameterized stored procedure:

```
str = " CREATE PROCEDURE Authors_info_prmtz @auth_id varchar(15) ,@auth_name
varchar(20) output, @auth_city varchar(20) output,@auth_state varchar(20)
output "
+ " AS "
+ " SELECT @auth_name=au_name, @auth_city=city, @auth_state=state "
+ " FROM Authors "
+ " WHERE au_id=@auth_id ";
Statement stmt=con.createStatement();
int rt=stmt.executeUpdate(str);
```

In the preceding code snippet, the `Authors_info_prmtz` stored procedure is created that accepts the author id as a parameter and retrieves the corresponding author information from the database. The retrieved information is stored in the `OUT` parameters. The `output` keyword is used to represent `OUT` parameters.

A stored procedure can accept one or multiple parameters. A parameter of a stored procedure can take any of the following forms:

- `IN`: Refers to the argument that you pass to a stored procedure.
- `OUT`: Refers to the return value of a stored procedure.

- `INOUT`: Combines the functionality of the `IN` and `OUT` parameters. The `INOUT` parameter enables you to pass an argument to a stored procedure. The same parameter can also be used to store a return value of a stored procedure.

When you use the stored procedures to perform database operations, it reduces network traffic because instead of sending multiple SQL statements to a database, a single stored procedure is executed.

Calling a Stored Procedure Without Parameters

The `Connection` interface provides the `prepareCall()` method that is used to create the `CallableStatement` object. This object is used to call a stored procedure of a database. The `prepareCall()` method is an overloaded method that has various forms. Some of the commonly used forms are:

- `CallableStatement prepareCall(String str)`: Creates a `CallableStatement` object to call a stored procedure. The `prepareCall()` method accepts a string as a parameter that contains an SQL statement to call a stored procedure. You can also specify the placeholders to accept parameters in the SQL statement.
- `CallableStatement prepareCall(String str, int resSetType, int resSetConcurrency)`: Creates a `CallableStatement` object that returns the `ResultSet` object, which has the specified result set type and concurrency mode. The method accepts the following three parameters:
  - `String` object: Contains an SQL statement to call a stored procedure. The SQL statement can contain one or more parameters.
  - `ResultSet` types: Specifies any of the three `Resultset` types, `TYPE_FORWARD_ONLY`, `TYPE_SCROLL_INSENSITIVE`, or `TYPE_SCROLL_SENSITIVE`.
  - `ResultSet` concurrency modes: Specifies either of these concurrency modes, `CONCUR_READ_ONLY` or `CONCUR_UPDATABLE`, for a result set.
- `CallableStatement prepareCall(String str, int resSetType, int resSetConcurrency, int resSetHoldability)`: Creates a `CallableStatement` object that returns a `ResultSet` object that has the specified result type, concurrency mode, and constant to set the result set state.

The following signature is used to call a stored procedure without parameters:

```
exec <procedure_name>
```

You can use the following code snippet to call a stored procedure that does not accept parameters:

```
String str = "exec Authors_info";
            CallableStatement cstmt = con.prepareCall(str);
            ResultSet rs = cstmt.executeQuery();
            while (rs.next())
              {
               System.out.println(" Author Id : " + rs.getString(1) + "\t");
```

```
                System.out.println(" Author Name : " + rs.getString(2) + "\t");
               }
```

In the preceding code snippet, `con` is the `Connection` object that invokes the `prepareCall()` method. The `str` variable contains the call to the `Authors_info` stored procedure, and this call is passed as a parameter to the `prepareCall()` method.

Calling a Stored Procedure with Parameters

The SQL escape syntax is used to call a stored procedure with parameters. The SQL escape syntax is a standard way to call a stored procedure from RDBMS and is independent of RDBMS. The driver searches the SQL escape syntax in the code and converts the SQL escape syntax into the database compatible form. There are two forms of the SQL escape syntax, one that contains result parameter and the other that does not contain result parameters. Both the forms can take multiple parameters. If the SQL escape syntax contains a result parameter, the result parameter is used to return a value from a stored procedure. The result parameter is an `OUT` parameter. Other parameters of the SQL escape syntax can contain the `IN`, `OUT`, or `INOUT` parameter. The signature of the SQL escape syntax is:

```
{[? =] call <procedure_name> [<parameter1>,<parameter2>, ., <parameterN>]}
```

The placeholders are used to represent the `IN`, `OUT`, and `INOUT` parameters of a stored procedure in the procedure call. The signature to call a stored procedure with parameters is:

```
{ call <procedure_name>(?) };
```

You need to set the value of the `IN` parameters before the `CallableStatement` object is executed. Otherwise, `SQLException` is thrown while processing the stored procedure. The set methods are used to specify the values for the `IN` parameters. The `CallableStatement` interface inherits the set methods from the `PreparedStatement` interface. The signature to set the value of the `IN` parameter is:

```
<CallableStatement_object>.setInt(<value>);
```

In the preceding signature, the `setInt()` method is used to set the value for an integer type, `IN` parameter.

If the stored procedure contains the `OUT` and `INOUT` parameters, these parameters should be registered with the corresponding JDBC types before a call to a stored procedure is processed. The JDBC types determine the Java data types that are used in the get methods while retrieving the values of the `OUT` and `INOUT` parameters. The `registerOut()` method is used to register the parameters. `SQLException` is thrown if the placeholders, representing the `OUT` and `INOUT` parameters, are not registered. The prototypes of the `registerOut()` method are:

- `registerOut(int index, int stype)`: Accepts the position of the placeholder and a constant in the `java.sql.Types` class as parameters. The `java.sql.Types` class contains constants for various JDBC types. For example, if you want to register the `VARCHAR` SQL data type, you should use the `STRING` constant of the `java.sql.Types` class. You can use the following method call to the `registerOut()` method to register a parameter:

```
cstmt.registerOutParameter(1, java.sql.Types.STRING);
```

- `registerOut(int index, int stype, int scale)`: Accepts the position of a placeholder, a constant in the `java.sql.Types` class, and a scale of the value that is returned as parameters. You need to define the scale of a parameter while registering numeric data types, such as `NUMBER`, `DOUBLE`, and `DECIMAL`. For example, if you want to register the `DECIMAL` SQL data type that has three digits after decimal, the value for the scale parameter should be three. You can use the following code snippet to specify the scale parameter while invoking the `registerOut()` method:

  ```
  cstmt.registerOutParameter(1, java.sql.Types.DECIMAL, 3);
  ```

You can use the `prepareCall()` method to call a stored procedure that accepts parameters. The `prepareCall()` method returns a result after processing the SQL and control statements defined in the procedure body. You can use the following code to call a stored procedure with parameters:

```
import java.sql.*;

public class CallProc
{

    public static void main(String args[])
    {
        String id, name, address, city;
        try
        {
            String str = "{call Authors_info_prmtz(?, ?, ?, ?)}";

            /*Initialize and load Type 4 JDBC driver*/
            Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver");
            /*Establish a connection with the database*/
            try (Connection con = 
DriverManager.getConnection("jdbc:sqlserver://sqlserver01;databaseName=Librar
y;user=user1;password=password#1234");
                    /*Call a stored procedure*/
                    CallableStatement cstmt = con.prepareCall(str);)
            {
                /*Pass IN parameter*/
                cstmt.setString(1, "A001");
                /*Register OUT parameters*/
                cstmt.registerOutParameter(2, Types.VARCHAR);
                cstmt.registerOutParameter(3, Types.VARCHAR);
                cstmt.registerOutParameter(4, Types.VARCHAR);
                /*Process the stored procedure*/
                cstmt.execute();
                /*Retrieve Authors information*/
                name = cstmt.getString(2);
                address = cstmt.getString(3);
                city = cstmt.getString(4);
                /*Display author information*/
                System.out.println("");
                System.out.println("Displaying Author Information");
                System.out.println("-----------------------------");
                System.out.println("First name: " + name);
                System.out.println("Address: " + address);
                System.out.println("City: " + city);
```

```
                }
            }
            catch (Exception e)
            {
                System.out.println("Error " + e);
            }
        }
    }
```

In the preceding code, the `Authors_info_prmtz` stored procedure is invoked. This stored procedure accepts one `IN` type parameter and four `OUT` type parameters. The `IN` parameter is used to specify the id of an author whose information you want to retrieve. The `OUT` parameters are used to retrieve the first name, last name, address, and city of the authors. The `setString()` method is used to specify the author id, and the `registerOut()` method is used to register the `OUT` parameters.

Using Metadata in JDBC

Metadata is the information about data, such as structure and properties of table. For example, a database contains the `employee` table that has the `name`, `address`, `salary`, `designation`, and `department` columns. The metadata of the `employee` table includes certain information, such as names of the columns, data type of each column, and constraints to enter data values in the columns. JDBC API provides the following two metadata interfaces to retrieve the information about the database and the result set:

- `DatabaseMetaData`
- `ResultSetMetaData`

Using the DatabaseMetaData Interface

The `DatabaseMetaData` interface provides the methods that enable you to determine the properties of a database. These properties include names of database tables, database version, SQL keywords, and isolation levels of the data stored in the database.

You can create an object of `DatabaseMetaData` using the `getMetaData()` method of the `Connection` interface. You can use the following code snippet to create an object of the `DatabaseMetaData` interface:

```
DatabaseMetaData dm=con.getMetaData();
```

In the preceding code snippet, `con` refers to an object of the `Connection` interface.

The methods declared in the `DatabaseMetaData` interface retrieve the database-specific information. The following table lists some commonly used methods of the `DatabaseMetaData` interface.

Method	***Description***
`ResultSet getColumns(String catalog, String schema, String table_name, String column_name)`	*Retrieves the information about a column of a database table that is available in the specified database catalog.*
`Connection getConnection()`	*Retrieves the database connection that creates the `DatabaseMetaData` object.*
`String getDriverName()`	*Retrieves the name of the JDBC driver for the `DatabaseMetaData` object.*
`String getDriverVersion()`	*Retrieves the version of the JDBC driver.*
`ResultSet getPrimaryKeys(String catalog, String schema, String table)`	*Retrieves the information about the primary keys of the database tables.*

Method	*Description*
`String getURL()`	*Retrieves the URL of the database.*
`boolean isReadOnly()`	*Returns a `boolean` value that indicates whether the database is read only.*
`boolean supportsSavepoints()`	*Returns a `boolean` value that indicates whether the database supports savepoints.*

The Methods of the DatabaseMetaData Interface

The methods in the `DatabaseMetaData` interface retrieve information about the database to which a Java application is connected. You can use the following code to retrieve and display the names of various database tables by using the methods of the `DatabaseMetaData` interface:

```
import java.sql.*;

public class TableNames
{
    public static void main(String args[])
    {
        try {
            /*Initialize and load the Type 4 driver*/
            Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver");
            /*Establish a connection with the database*/
            try (Connection con =
DriverManager.getConnection("jdbc:sqlserver://sqlserver01;databaseName=Librar
y;user=user1;password=password#1234");)
            {
                /*Create a DatabaseMetaData object*/
                DatabaseMetaData dbmd = con.getMetaData();
                String[] tabTypes = {"TABLE"};
                /*Retrieve the names of database tables*/
                System.out.println("");
                System.out.println("Tables Names");
                System.out.println("------------------");
                ResultSet tablesRS = dbmd.getTables(null, null, null,
tabTypes);
                while (tablesRS.next()) /*Display the names of database
tables*/
                {
                    System.out.println(tablesRS.getString("TABLE_NAME"));
                }
                            }
        }
        catch (Exception e)
        {
            System.out.println("Error : " + e);
        }
    }
}
```

In the preceding code, a connection is established with the `Library` data source. The object of the `DatabaseMetaData` interface is declared using the `getMetaData()` method. The `DatabaseMetaData` object is used to retrieve the names of database tables using the `getTables()` method.

Using the ResultSetMetaData Interface

The `ResultSetMetaData` interface contains various methods that enable you to retrieve information about the data in a result set, such as numbers, names, and data types of the columns. The `ResultSet` interface provides the `getMetaData()` method to create an object of the `ResultSetMetaData` interface. You can use the following code snippet to create an object of the `ResultSetMetaData` interface:

```
ResultSetMetaData rm=rs.getMetaData();
```

In the preceding code snippet, `rs` refers to an object of the `ResultSet` interface. `rs` calls the `getMetaData()` method to create an object of the `ResultSetMetaData` interface.

The following table lists some commonly used methods of the `ResultSetMetaData` interface.

Method	***Description***
`int getColumnCount()`	*Returns an integer indicating the total number of columns in a `ResultSet` object.*
`String getColumnLabel(int column_index)`	*Retrieves the title of the table column corresponding to the specified index.*
`String getColumnName(int column_index)`	*Retrieves the name of the table column corresponding to the specified index.*
`int getColumnType(int column_index)`	*Retrieves the SQL data type of the table column corresponding to the specified index.*
`String getTableName(int column_index)`	*Retrieves the name of the database table that contains the column corresponding to the specified index.*
`boolean isAutoIncrement(int column_index)`	*Returns a `boolean` value that indicates whether the table column corresponding to the specified index increments automatically.*
`boolean isCaseSensitive(int column_index)`	*Returns a `boolean` value that indicates whether the table column corresponding to the specified index is case-sensitive.*

Method	*Description*
`boolean isReadOnly(int column_index)`	*Returns a* `boolean` *value that indicates whether the column in a* `ResultSet` *column corresponding to the specified index is read only.*
`boolean isWritable(int column_index)`	*Returns a* `boolean` *value that indicates whether the* `ResultSet` *column corresponding to the specified index is writeable.*

The Methods of the ResultSetMetaData Interface

Just a

What are the metadata interfaces used to retrieve information about the database and the result set?

Answer:

The metadata interfaces used to retrieve information about the database and the result set are:

1. `DatabaseMetaData`
2. `ResultSetMetaData`

Activity 10.2: Creating an Application to Determine the Structure of a Table

Practice Questions

1. The ________ interface provides the methods that enable you to determine the properties of a database or RDBMS.

2. Identify the method of the `PreparedStatement` interface that sets the Java `byte` type value for the parameter corresponding to the specified index.
   a. `setByte(int index, byte val)`
   b. `setBytes(int index, byte[] val)`
   c. `setString(int index, String val)`
   d. `setShort(int index, short val)`

3. Why do you need to disable the auto-commit mode while working with batch updates?

4. Which is a standard way to call a stored procedure from RDBMS?

5. Which method of the `Connection` interface is used to create the `callableStatement` object?
   a. `prepareCall()`
   b. `getMetaData()`
   c. `prepareStatement()`
   d. `createStatement()`

Summary

In this chapter, you learned that:

- The `PreparedStatement` object allows you to pass runtime parameters to the SQL statements using the placeholders.
- There can be multiple placeholders in a single SQL statement. An index value is associated with each placeholder depending upon the position of the placeholder in the SQL statement.
- The placeholder stores the value assigned to it until the value is explicitly changed.
- A transaction is a set of one or more SQL statements executed as a single unit. A transaction is complete only when all the SQL statements in a transaction are successfully executed.
- If the `setAutoCommit()` method is set to `true`, the database operations performed by the SQL statements are automatically committed in the database.
- The `commit()` method reflects the changes made by the SQL statements permanently in the database.
- The `rollback()` method is used to undo the effect of all the SQL operations performed after the last commit operation.
- A batch is a group of update statements sent to a database to be executed as a single unit. You send the batch to a database as a single request using the same `Connection` object.
- The `executeBatch()` method returns an integer array that stores the update count for all the SQL statements that are executed successfully in a batch. The update count is the number of database rows affected by the database operation performed by each SQL statement.
- Batch update operations can throw two types of exceptions, `SQLException` and `BatchUpdateException`.
- `SQLException` is thrown when the database access problem occurs. `SQLException` is also thrown when a `SELECT` statement that returns a `ResultSet` object is executed in a batch.
- `BatchUpdateException` is thrown when the SQL statement in the batch cannot be executed due to the problem in accessing the specified table or presence of illegal arguments in the SQL statement.
- The `CallableStatement` interface contains various methods that enable you to call the stored procedures from a database.
- The parameters of a stored procedure can take any of the following three forms:
  - `IN`
  - `OUT`
  - `INOUT`
- Metadata is the information about data, such as the structure and properties of table.
- JDBC API provides two metadata interfaces to retrieve the information about the database and result set, `DatabaseMetaData` and `ResultSetMetaData`.
- The `DatabaseMetaData` interface declares the methods that enable you to determine the properties of a database.
- The `ResultSetMetaData` interface declares the methods that enable you to determine the information of a result set.

- The `getMetaData()` method of the `Connection` interface enables you to obtain the objects of the `DatabaseMetaData` interface. The methods in the `DatabaseMetaData` interface retrieve the information only about the database to which a Java application is connected.
- The `getMetaData()` method of the `ResultSet` interface enables you to create the instance of the `ResultSetMetaData` interface.