CHAPTER 5

■ ■ ■

JDBC API

QUESTIONS AND EXERCISES

1. What is JDBC API?

   Answer:

   The JDBC API provides a standard database-independent interface to interact with any tabular data source. Most of the time, it is used to interact with a relational database management system (RDBMs). However, using the JDBC API, it is possible to interact with any tabular data source, such as an Excel Spreadsheet, a flat file, etc. Typically, you use the JDBC API to connect to a database, query the data, and update the data. It also lets you execute SQL stored procedures in a database using a database-independent syntax.

2. What is a JDBC driver?

   Answer:

   A JDBC driver is a software component that allows Java programs to interact with a database.

3. What is the role of the `DriverManager` in a JDBC application? How do you load a JDBC driver?

   Answer:

   The `DriverManager` is a service to manage a set of JDBC drivers.

   You call the `getConnection()` method of the `DriverManager` to get a connection to a database. The `DriverManager` loads a JDBC driver depending on the URL passed to the `getConnection()` method.

Before JDK 9, you had to register your JDBC driver with the `DriverManager` using any of the following methods:

- By setting the `jdbc.drivers` system property
- By loading the driver class into the JVM.
- By using the `registerDriver()` method of the `DriverManager` class

Starting from JDK 9 in module mode, you do not need to register a JDBC driver with a `DriverManager`. If you are using JDK 9 in legacy mode, you still need to register your JDBC driver for the `DriverManager` to load it.

4. What does an instance of the `Connection` interface represent? List a few uses of a `Connection` in a Java application.

Answer:

An instance of the `Connection` interface represents a connection to a database.

You can use a `Connection` to do the following:

- Get information about the database
- Execute SQL statements and stored procedure and retrieve results
- Know the attributes of the connection itself

5. Describe the syntax of a connection URL used in establishing a JDBC connection.

Answer:

The syntax to define the connection URL is as follows:

```
<protocol>:<sub-protocol>:<data-source-details>
```

The `<protocol>` part is always set to `jdbc`. The `<sub-protocol>` part is vendor-specific. The `<data-source-details>` part is DBMS specific that is used to locate the database. In some cases, you can also specify some connection properties in this last part of the URL. The following is an example of a connection URL that uses Oracle's thin JDBC driver to connect to an Oracle DBMS:

```
jdbc:oracle:thin:@localhost:1521:mysid
```

6. What is the auto-commit mode of a `Connection`? How do you disable the auto-commit mode of a `Connection`?

Answer:

When you connect to a database, the auto-commit property for the `Connection` object is set to `true` by default. If a connection is in the auto-commit mode, a SQL statement is committed automatically after its successful execution. If a connection is not in the auto-commit mode, you must call the `commit()` or `rollback()` method of the `Connection` object to commit or rollback a transaction. Typically, you disable the auto-commit mode for a connection in a JDBC application, so the logic in your application controls the final outcome of the transaction. To disable the auto-commit mode, you need to call the `setAutoCommit(false)` on the `Connection` after connection has been established. If a connection URL allows you to set the auto-commit mode, you can also specify it as part of the connection URL. The following snippet of code shows how to disable the auto-commit mode:

```
String dbURL = /* Specify your database URL */;
String userId = /* Specify your database user ID */;
String password = /* Specify your database password */;


// Get a connection
Connection conn = DriverManager.getConnection(dbURL, userId, password);

// Disable the auto-commit mode for the connection
conn.setAutoCommit(false);
```

7. When do you use the `DatabaseMetaData` interface?

Answer:

A `DatabaseMetaData` provides detailed information about the features supported by a database and a JDBC driver. The vendor of the JDBC driver provides an implementation of the `DatabaseMetaData` interface. You obtain a `DatabaseMetaData` using the `getMetaData()` method of a `Connection`:

```
// Get a Connection
Connection conn = /* Get a database connection */ ;

// Get a DatabaseMetaData instance
DatabaseMetaData dbmd = conn.getMetaData();
```

The `DatabaseMetaData` interface contains several methods to query the database.

8. Write a snippet of code to check if a database supports stored procedure call using the JDBC syntax. Assume that `conn` is a variable that contains the reference to a `Connection` to the database.

Solution:

To find out if your DBMS supports stored procedure calls using the JDBC syntax, you can call the `supportsStoredProcedures()` method of the `DatabaseMetaData` object. It returns `true` if the DBMS supports stored procedures; otherwise, it returns `false`. The following snippet of code shows you how to check for the stored procedure support for your database:

```
try {
    Connection conn = null;
    DatabaseMetaData dbmd = conn.getMetaData();
    boolean spSupported = dbmd.supportsStoredProcedures();
    if (spSupported) {
        // Database supports stored procedures
    } else {
        // Database does not support stored procedures
    }
} catch (SQLException e) {
    e.printStackTrace();
}
```

9. Suppose you get a SQL statement as a `String` in your Java program. What object, a `Statement`, a `PreparedStatement`, or a `CallableStatement`, would you use to execute the SQL statement?

Answer:

`Statement`

10. List at least two disadvantages of using a `Statement` to execute SQL statements in your Java programs.

Answer:

Using a `Statement` to execute a SQL statement has the following disadvantages:

- The SQL statement in a string form may be subject to hackers attack using a SQL injection technique.
- A SQL in string form is compiled every time it is used. If you need to execute the same SQL statement with different parameters, the performance is worse if you use a `Statement`. You need to use a `PreparedStatement` in such a case, which will compile the statement only once.

11. When do you use a `PreparedStatement`? List a few advantages of using a `PreparedStatement`.

Answer:

You use a `PreparedStatement` if you want to precompile a SQL statement once and execute it multiple times. It lets you specify a SQL statement in the form of a string that uses placeholders. You need to supply the values of the placeholders before you execute the statement. Using a `PreparedStatement` is preferred over using a `Statement` for the following three reasons:

- The SQL statement in a string form may be subject to hackers attack using a SQL injection technique. A `PreparedStatement` constructs a SQL in a string format using placeholders. Using a `PreparedStatement` eliminates the threat of a SQL injection.
- The `PreparedStatement` improves the performance of your JDBC application by compiling a statement once and executing it multiple times.
- A `PreparedStatement` lets you use Java data types to supply values in a SQL statement instead of using strings.

12. When do you use a `CallableStatement`? List a few advantages of using a `CallableStatement`.

Answer:

The `CallableStatement` interface inherits from the `PreparedStatement` interface. A `CallableStatement` is used to call a SQL stored procedure or a function in a database. You can also call a stored procedure or a function using the `Statement` object. However, using a `CallableStatement` is the preferred way because the JDBC API makes it possible to call SQL stored procedures and functions using a standard syntax. The JDBC specification defines an escape sequence for stored procedures/functions to execute them in a database.

13. Write a snippet of code twice–once using a `Statement` and once using a `PreparedStatement`– that will update the `dob` column in a `person` table where the `person_id` column's value is 101. The data type of the `dob` column is date. The dob column should be set to January 12, 1968. Your code should work for all

databases that support date column, irrespective of the format they use to store date values.

Solution:

The following snippets of code assume that you have a `Connection` instance reference stored in a variable named `conn`.

```
/* Using a Statement */
String sql = "update person set dob = {d '1968-01-12'} where person_id = 101";

// Create and execute a statement
Statement stmt = conn.createStatement();
stmt.executeUpdate(sql);

// Commit the transaction
conn.commit();


/* Using a PreparedStatement */
String sql = "update person set dob = ? where person_id = ?";

PreparedStatement pstmt = null;
pstmt = conn.prepareStatement(sql);


// Set the parameters
java.sql.Date dob = java.sql.Date.valueOf("1968-01-12");
pstmt.setDate(1, dob);  // dob
pstmt.setInt(2, 101);   // person_id

// Execute the update statement
pstmt.executeUpdate();

// Commit the transaction
conn.commit();
```

14. Why do you use a `CallableStatement`?

Answer:

A `CallableStatement` is used to call a SQL stored procedure or a function in a database using a database-neutral syntax.

15. What is a `ResultSet`? In brief, describe scrollability, concurrency, and holdability of a `ResultSet`.

Answer:

When you execute a query (a `SELECT` statement) in a database, it returns the matching records in the form of a result set. You can consider a result set as a data arranged in rows and columns. The `SELECT` statement determines the number of rows and columns that is contained in the result set. A `Statement` (or `PreparedStatement` or `CallableStatement`) returns the result of a query as a `ResultSet` object. A `ResultSet` also contains information about the properties of the columns in the result set such as the data types of the columns, names of the columns, etc. A `ResultSet` maintains a cursor, which points to a row in the result set. It works like a cursor in database programs. You can scroll the cursor to a specific row in the result set to access or manipulate the column values for that row. The cursor can point to only one row at a time. The row to which it points at a point in time is called the *current row*. There are different ways to move the cursor of a `ResultSet` to a row in the result set.

A result set has the following three properties: `Scrollability`, `Concurrency`, and `Holdability`.

`Scrollability` determines the ability of the `ResultSet` to scroll through the rows. By default, a `ResultSet` is scrollable only in the forward direction. When you have a forward-only scrollable `ResultSet`, you can move the cursor starting from the first row to the last row. Once you move to the last row, you cannot reuse the `ResultSet` because you cannot scroll back in a forward-only scrollable `ResultSet`. You can also create a `ResultSet` that can scroll in the forward as well as the backward direction.

`Concurrency` refers to the ability of a `ResultSet` to update data. By default, a `ResultSet` is read-only and it does not let you update its data. If you want to update data in a database through a `ResultSet`, you need to request an updatable result set from the JDBC driver.

`Holdability` refers to the state of a `ResultSet` after a transaction that it is associated with has been committed. A `ResultSet` may be closed or kept open when the transaction is committed. The default value of the `holdability` of a `ResultSet` is dependent on the JDBC driver.

16. What does the `getRow()` method of a `ResultSet` return?

Answer:

The `getRow()` method of a `ResultSet` returns the current row number. If there is no current row number, it returns 0. The first row in the result set has a row number of 1.

17. What is the return value of the `next()` method of a `ResultSet`? How do you interpret the return value?

Answer:

The of the `next()` method of a `ResultSet` moves the cursor one row forward from its current row. If this method returns `true`, it means the cursor is positioned at a valid row in the `ResultSet`. If this method returns `false`, it means the cursor is positioned after the last row in the `ResultSet`.

18. How do you determine that the value for a column returned in a `ResultSet` was `null`?

Answer:

You need to read the value of the column from the `ResultSet` and then call the `wasNull()` method on the `ResultSet`. If the `wasNull()` method returns `true`, the last value read from the `ResultSet` was a SQL NULL.

19. Describe the typical sequence of steps you need to perform to insert a record into a table using a `ResultSet`.

Answer:

The following are typical steps you need to use to insert a record into a table using a `ResultSet`:

- Position the cursor to the insert row using the `moveToInsertRow()` method of the `ResultSet`.
- Set the values for all the columns (at least for non-nullable columns) using one of the `updateXxx()` methods of the `ResultSet`, where `Xxx` is the data type of the column.
- Send the newly inserted row to the database by calling the `insertRow()` method of the `ResultSet` interface.
- Commit the changes to make the newly inserted row permanent in the database.

20. How do you delete a record from a table using a `ResultSet`?

Answer:

The following are typical steps you need to use to delete a record into a table using a `ResultSet`:

- Position the cursor at a valid row in the `ResultSet`.
- Call the `deleteRow()` method of the `ResultSet` to delete the current row. The `deleteRow()` method deletes the row from the `ResultSet` and, at the same time, it deletes the row from the database. There is no way to cancel the delete operation except by rolling back the transaction. If the auto-commit mode is enabled on the `Connection`, `deleteRow()` will permanently delete the row from the database.

21. What is the use of a `ResultSetMetaData`?

Answer:

A `ResultSet` contains the rows of data returned by executing a query and detailed information about the columns. The information that it contains about the columns in the result set is called the *result set metadata*. An instance of the `ResultSetMetaData` interface represents the result set metadata. You can get a `ResultSetMetaData` by calling the `getMetaData()` method of the `ResultSet`.

```
ResultSet rs = /* get result set object */;
ResultSetMetaData rsmd = rs.getMetaData();
```

A `ResultSetMetaData` contains a lot of information about all columns in a result set. All of the methods, except `getColumnCount()`, in the `ResultSetMetaData` accept a column index in the result set as an argument. It contains the table name, name, label, database data type, class name in Java, nullability, precision, etc. of a column. It also contains the column count in the result set. Its `getTableName()` method returns the table name of a column; the `getColumnName()` method returns the column's name; the `getColumnLabel()` method returns the column's label; the `getColumnTypeName()` method returns the column type in database; and the `getColumnClassName()` method returns Java class used to represent the data for the column. Its `getColumnCount()` method returns the number of columns in the result set.

The column label is a nice printable text that is used in a query after the column name. The following query uses `"Person ID"` as the column label for the `person_id` column. The `first_name` column does not have a specified label.

```
select person_id as "Person ID", first_name from person
```

The `getColumnLabel(1)` method call will return `"Person ID"`, whereas `getColumnName(1)` will return `person_id` if the above query is used for a result set. If the column label is not specified in a query, the `getColumnLabel()` method returns the column name.

22. What is a rowset. Describe the following rowsets in brief: `JdbcRowSet`, `CachedRowSet`, `WebRowSet`, `FilteredRowSet`, and `JoinRowSet`.

Answer:

An instance of the `RowSet` interface is a wrapper for a result set. The `RowSet` interface inherits from the `ResultSet` interface. In simple terms, a `RowSet` is a Java object that contains a set of rows from a tabular data source. The tabular data source could be a database, a flat file, a spreadsheet, etc. The `RowSet` interface is in the `javax.sql` package.

JdbcRowSet

A `JdbcRowSet` is also called a *connected rowset* because it always maintains a database connection. You can think of a `JdbcRowSet` as a thin wrapper for a `ResultSet`. As a `ResultSet` always maintains a database connection, so does a `JdbcRowSet`. It adds some methods that let you configure the connection behaviors. You can use its `setAutoCommit()` method to enable or disable the auto-commit mode for the connection. You can use its `commit()` and `rollback()` methods to commit or rollback changes made to its data.

CachedRowSet

A `CachedRowSet` is also called a *disconnected rowset* because it is disconnected from a database when it does not need a database connection. It keeps the database connection open only for the duration it needs to interact with the database. Once it is done with the connection, it disconnects. For example, it connects to a database when it needs to retrieve or update data. It retrieves all data generated by the command and caches the data in memory. It provides a new feature called *paging*, which lets you deal with large volume of data in chunks. A `CachedRowSet` is always serializable, scrollable, and updatable.

WebRowSet

The `WebRowSet` is a `CachedRowSet` with XML export/import support.

FilteredRowSet

The `FilteredRowSet` is a `WebRowSet` with the filtering capability at the client side. You can apply a filter to the rowset by using a where clause in its SQL command, which is executed in a database. A `FilteredRowSet` lets you filter the rows of a rowset after it has retrieved the data from a database. You can think of a `FilteredRowSet` as a rowset that lets you view its rows based on a set of criteria, which is called a *filter*.

JoinRowSet

The `JoinRowSet` is a `WebRowSet` with the ability to combine (or join) two or more disconnected rowsets into one rowset. Rows from two or more tables are joined in a query using a SQL `JOIN`. A `JoinRowSet` lets you have a SQL `JOIN` between two or more rowsets without using a SQL `JOIN` in the query. Using a `JoinRowSet` is easy. You retrieve data in multiple rowsets: `CachedRowSet`, `WebRowSet`, or `FilteredRowSet`. Create an empty `JoinRowSet` and add all rowsets to it by calling its `addRowSet()` method. The first rowset that is added to the `JoinRowSet` becomes the reference rowset for establishing the joins when more rowsets are added. You can specify the `JOIN` columns in a rowset individually or when you add a rowset to a `JoinRowSet`.

23. Suppose you want to read a picture stored in a database in a Java program. What Java type will you use to read the picture data?

Answer:

`java.swl.Blob`

24. What are savepoint, rollback, and commit in database parlance? How do you perform these activities in a Java program using JDBC?

Answer:

A database transaction consists of one or more changes as a unit of work. A savepoint in a transaction is like a marker that marks a point in a transaction so that, if needed, the transaction can be rolled back (or undone) up to that point. Calling the `setSavepoint()` method of the `Connection` interface creates a savepoint and returns its reference. You need to use the reference returned from the `setSavepoint()` method to rollback a transaction after a specific savepoint. You can also release a savepoint explicitly by calling `releaseSavepoint(Savepoint sp)` method of a `Connection`. Releasing a savepoint also releases all subsequent savepoints that were created after this savepoint. For example, calling `conn.releaseSavepoint(sp2)` will release savepoints `sp2`, `sp3`, and `sp4`. All savepoints in a transaction are released when the transaction is committed or rolled back entirely.

Rolling back is a transaction undoes all the changes make in that transaction. Using the `rollback()` method of a Connection rolls back all changes of a transaction. Using the `rollback(Savepoint savepoint)` method rolls back all changes made after the specified savepoint.

Committing a transaction makes all changes in that transaction permanent. The `commit()` method of a `Connection` makes all changes in a transaction permanent.

25. How do you enable SQL tracing in a JDBC application?

Answer:

You can enable JDBC tracing that will log JDBC activities to a `PrintWriter`. You can use the `setLogWriter(PrintWriter out)` static method of the `DriverManager` to set a log writer if you are using the `DriverManager` to connect to a database. If you are using a `DataSource`, you can use its `setLogWriter(PrintWriter out)` method to set a log writer. Setting `null` as a log writer disables the JDBC tracing. The following snippet of code sets a log writer to a `C:\jdbc.log` file on Windows:

```
// Sets the log writer to a file c:\jdbc.log
PrintWriter pw = new PrintWriter("C:\\jdbc.log");
DriverManager.setLogWriter(pw);
```

When you call the `setLogWriter()` method of the `DriverManager` class with the Java security enabled, Java checks for a `java.sql.SQLPermission`. You can grant this permission to an executing code in a security policy file. The following is an example of an entry in a security policy file that grants a permission to execute the `setLogWriter()` method on the `DriverManager`:

```
grant {
    permission java.sql.SQLPermission "setLog";
};
```