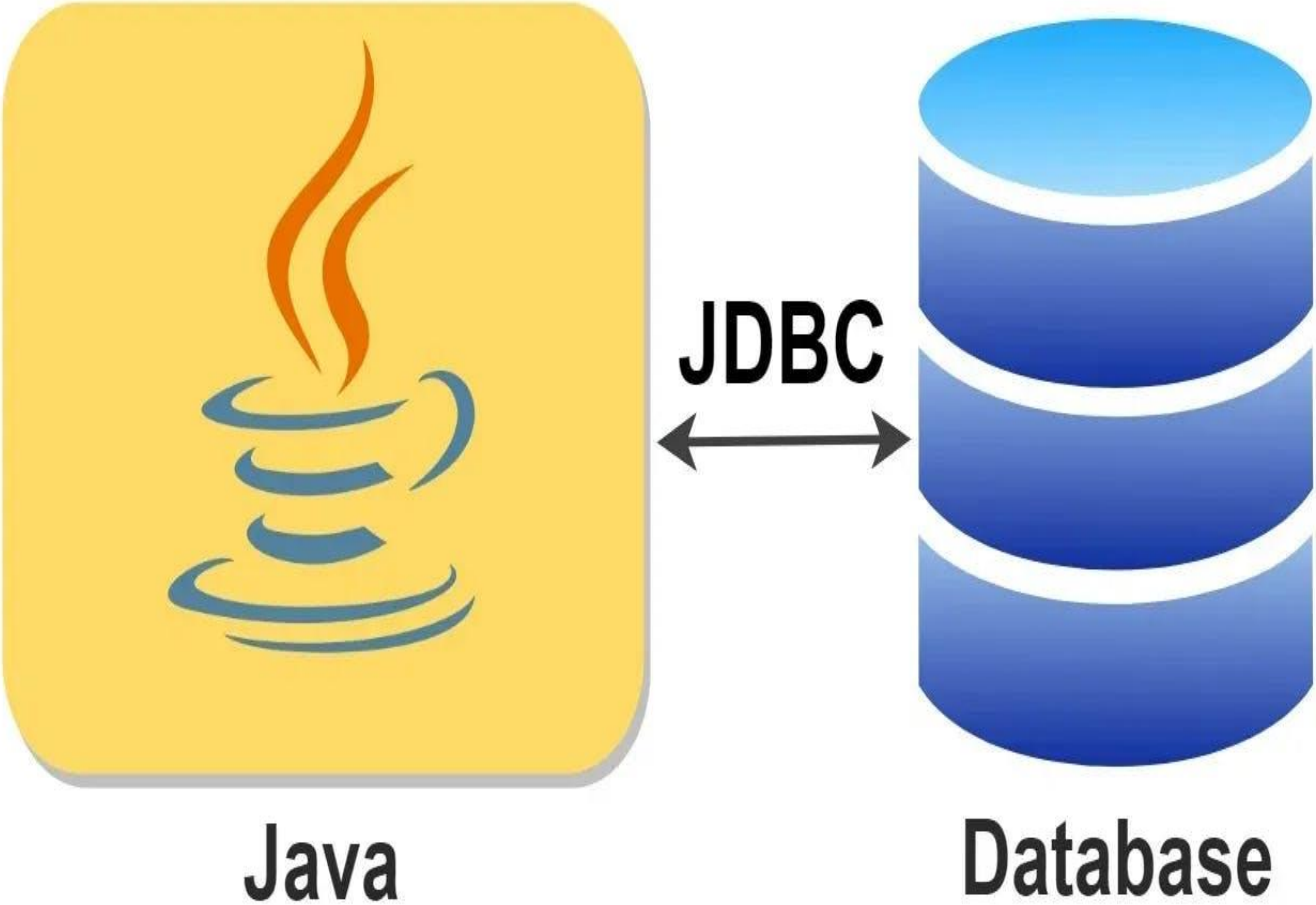
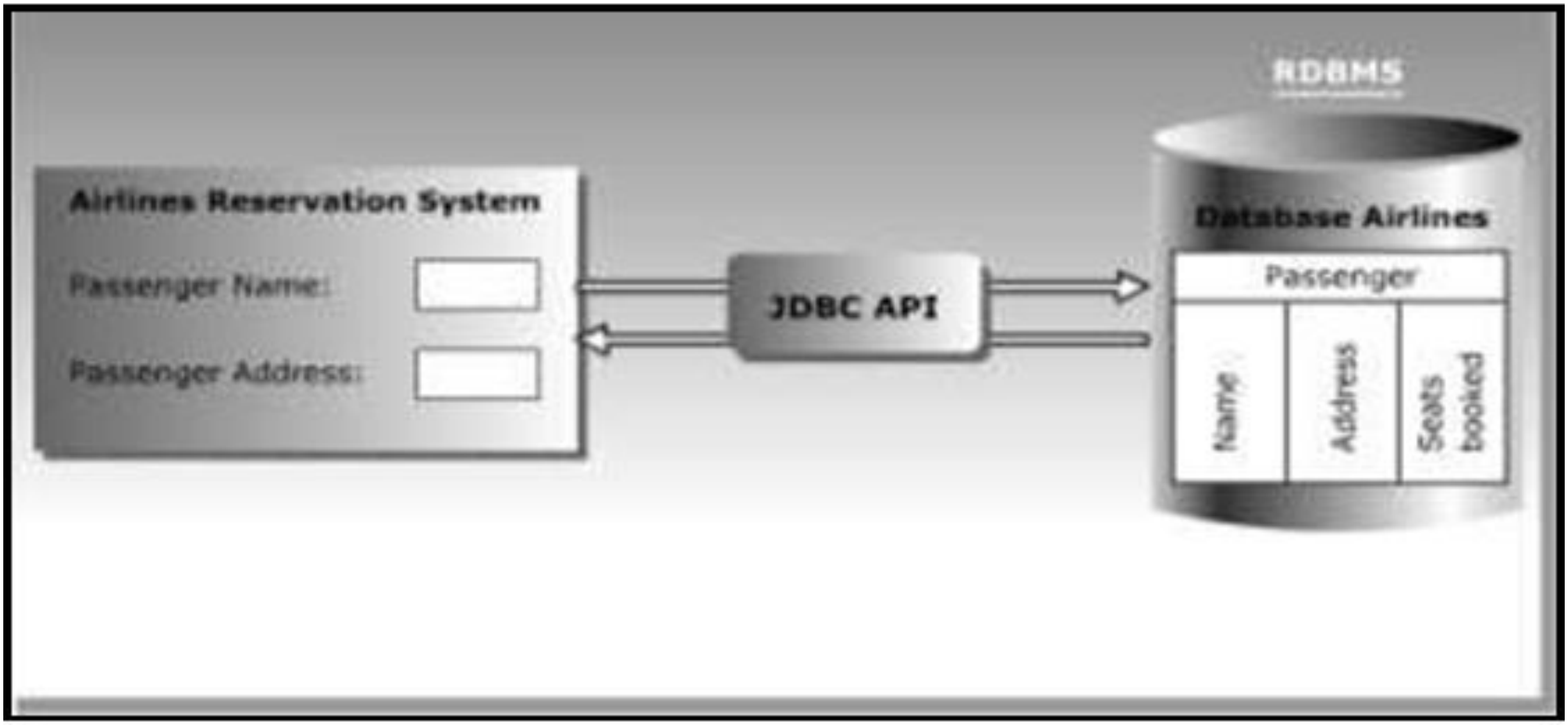
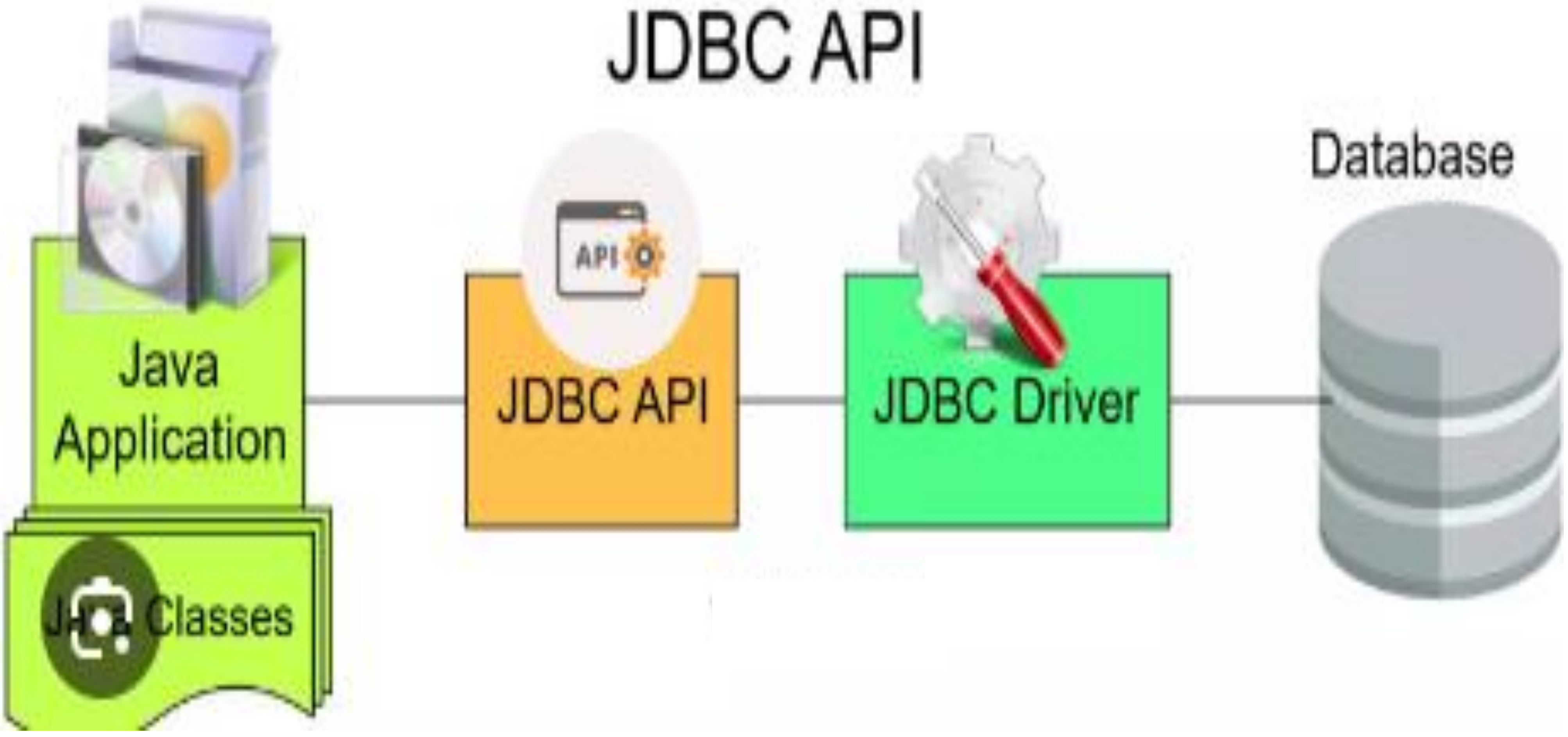


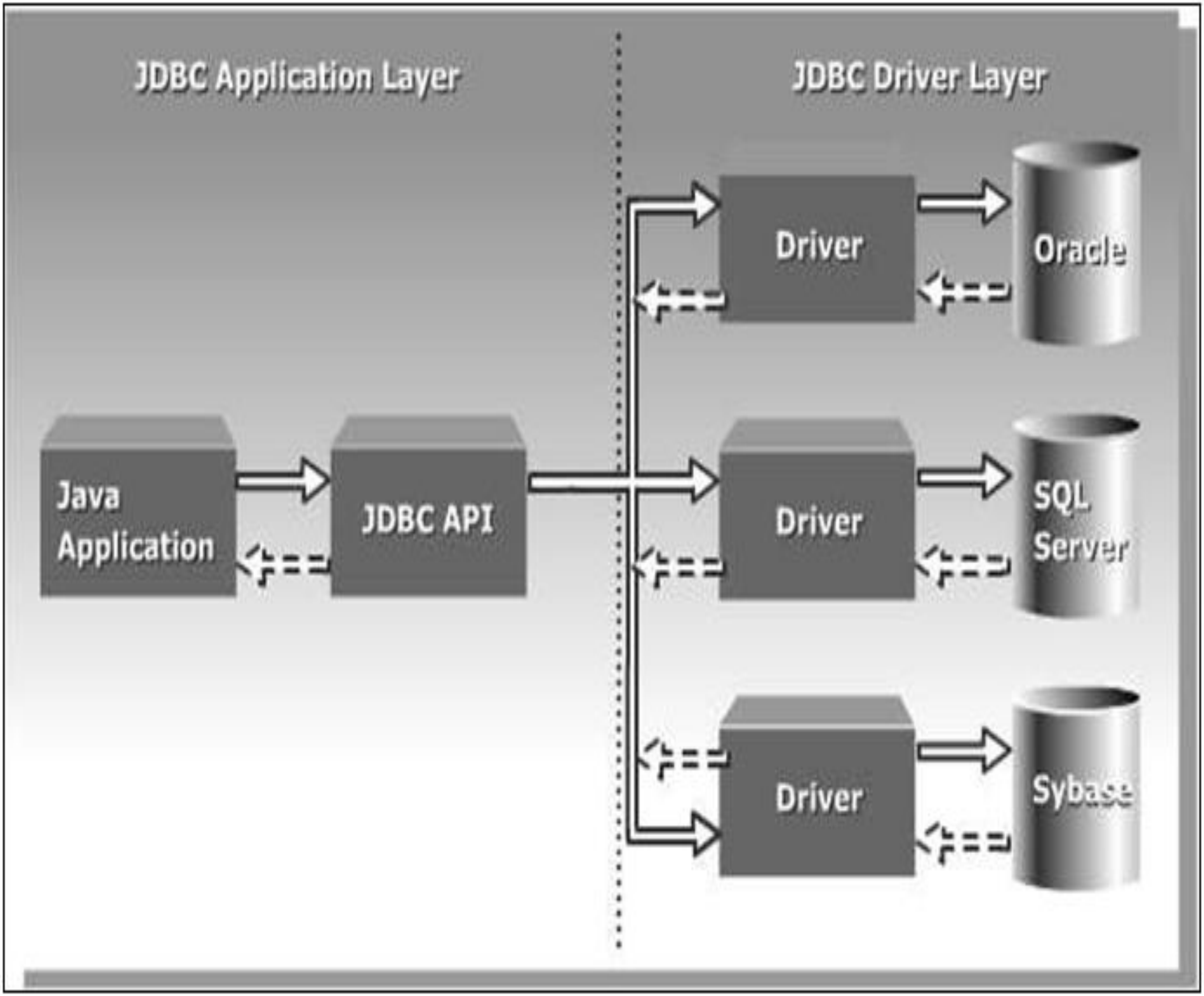
JDBC
Java
Database



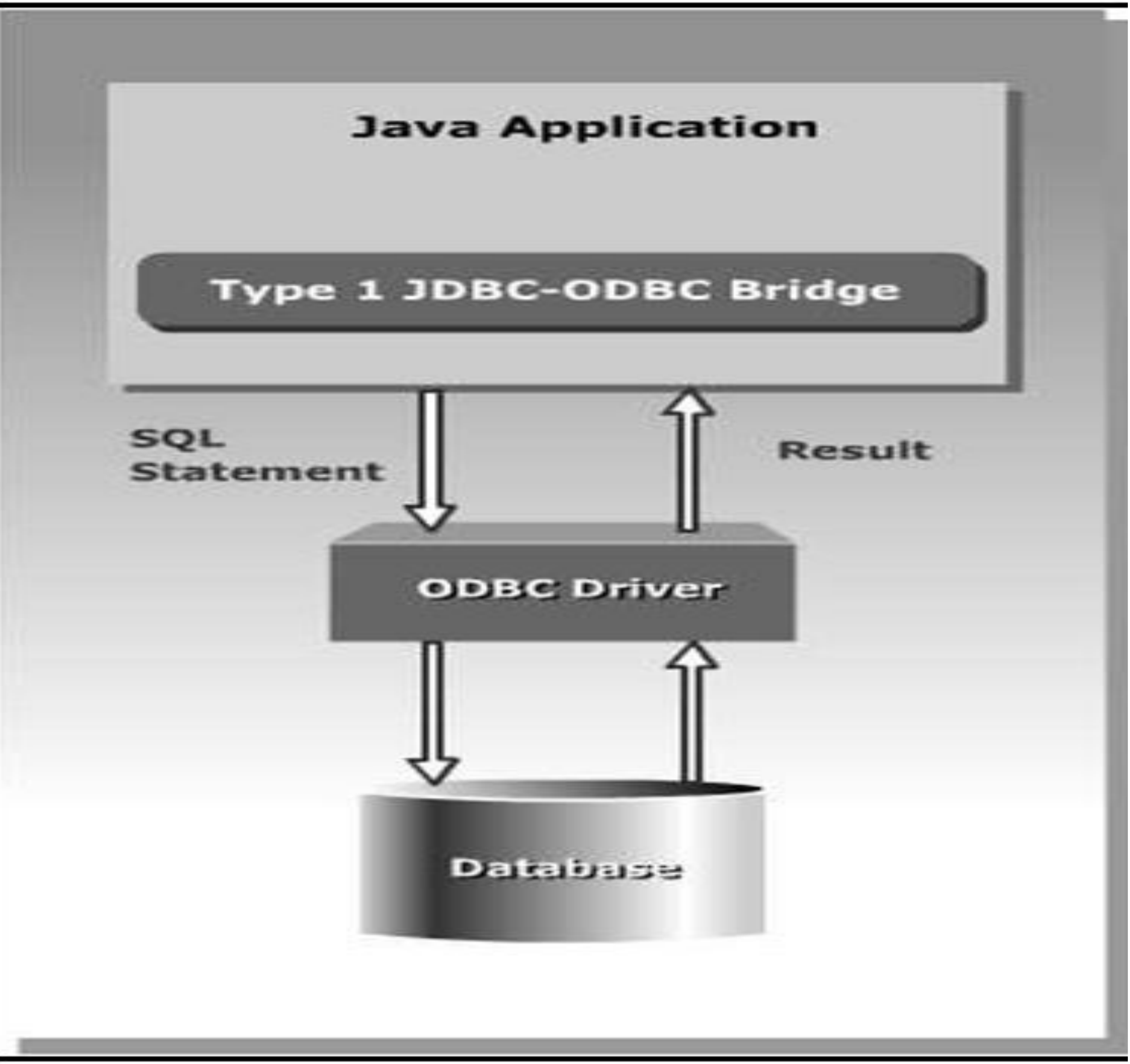
The Database Connectivity Using the JDBC API



The following figure shows the JDBC architecture.

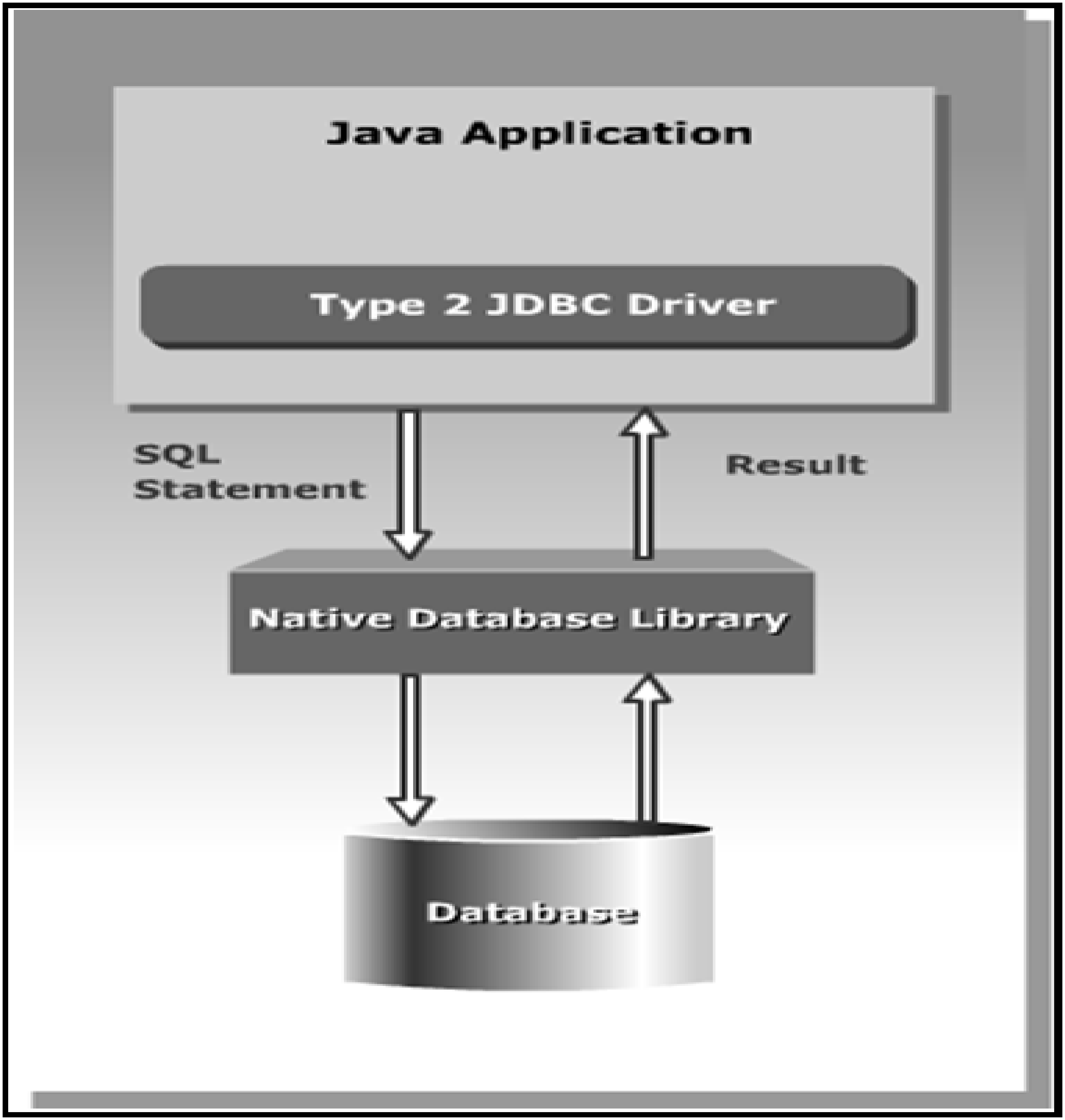


The JDBC Architecture



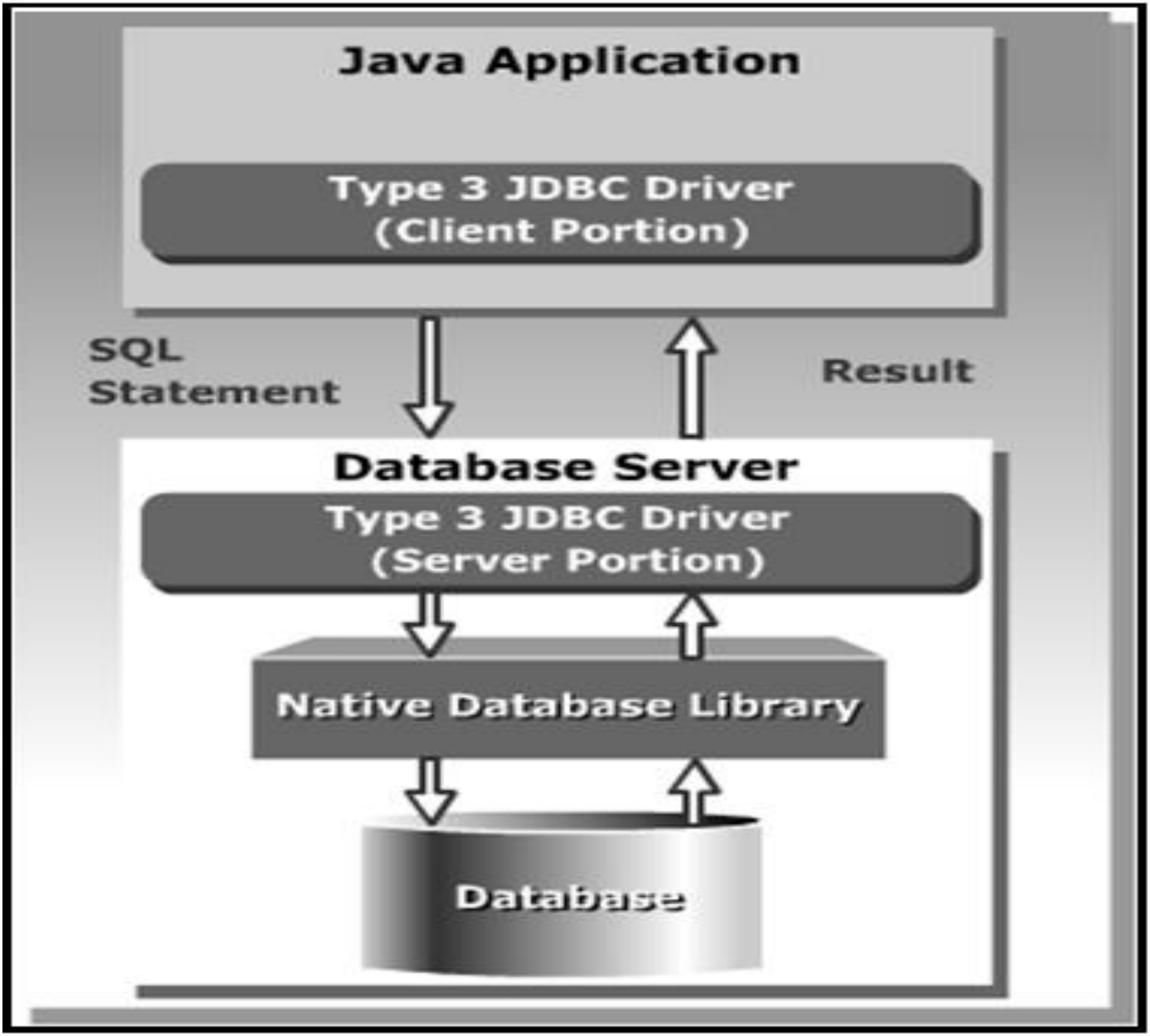
The JDBC-ODBC Bridge Driver

The following figure shows how the Native-API driver works.

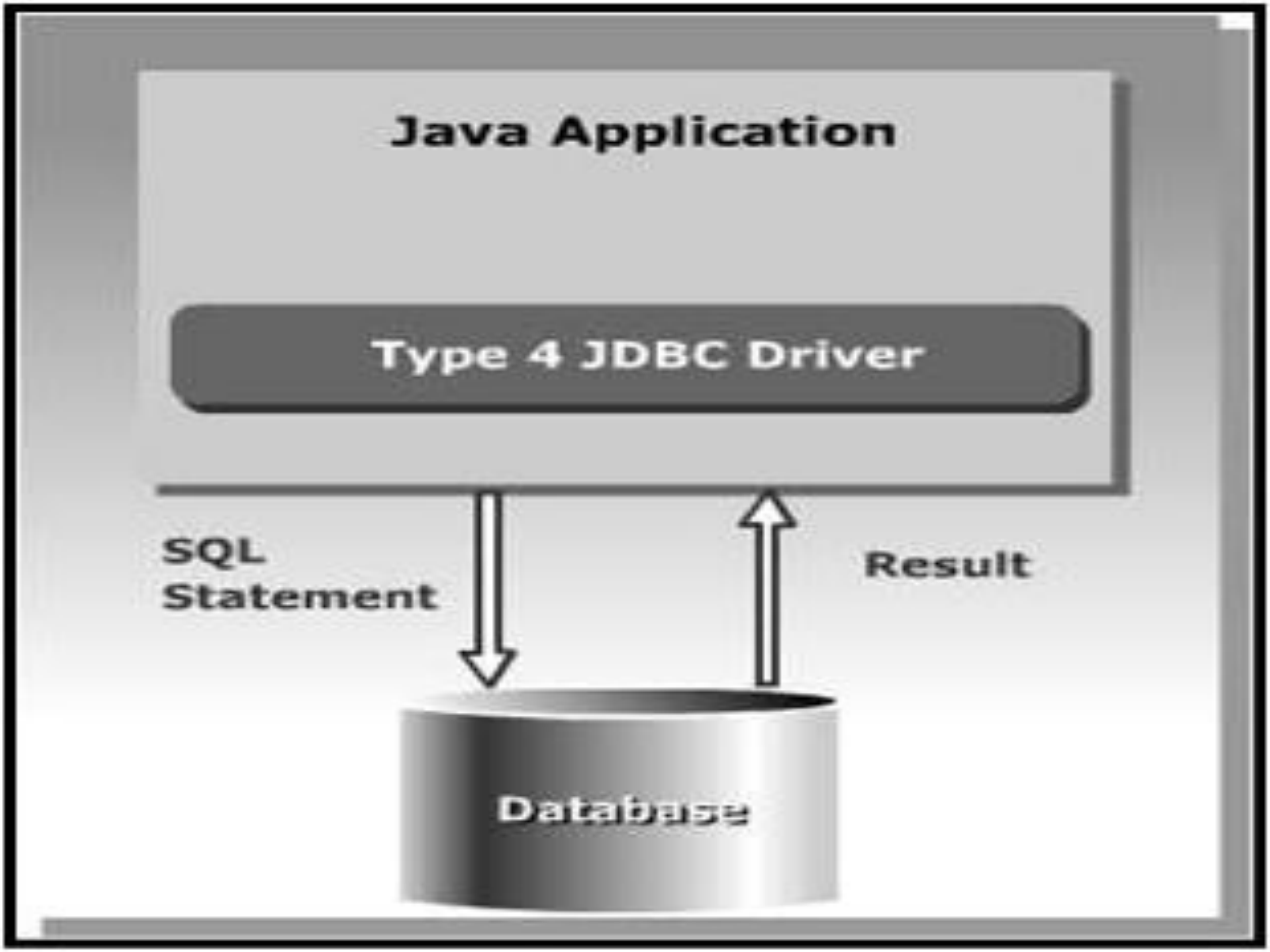


The Native-API Driver

The following figure shows how the Network Protocol driver works.



The Network Protocol Driver



The Native Protocol Driver

Introduction to JDBC

Java Database Connectivity (JDBC) is an API that executes SQL statements. It consists of a set of classes and interfaces written in the Java programming language. The interface is easy to connect to any database using JDBC. The combination of Java and JDBC makes the process of disseminating(Spreading) information easy and economical.

This chapter introduces the JDBC architecture. It gives an overview of the different types of drivers supported by JDBC. It elaborates the methods to establish the connection with SQL Server using the Type 4 JDBC driver.

Identifying the Layers in the JDBC Architecture

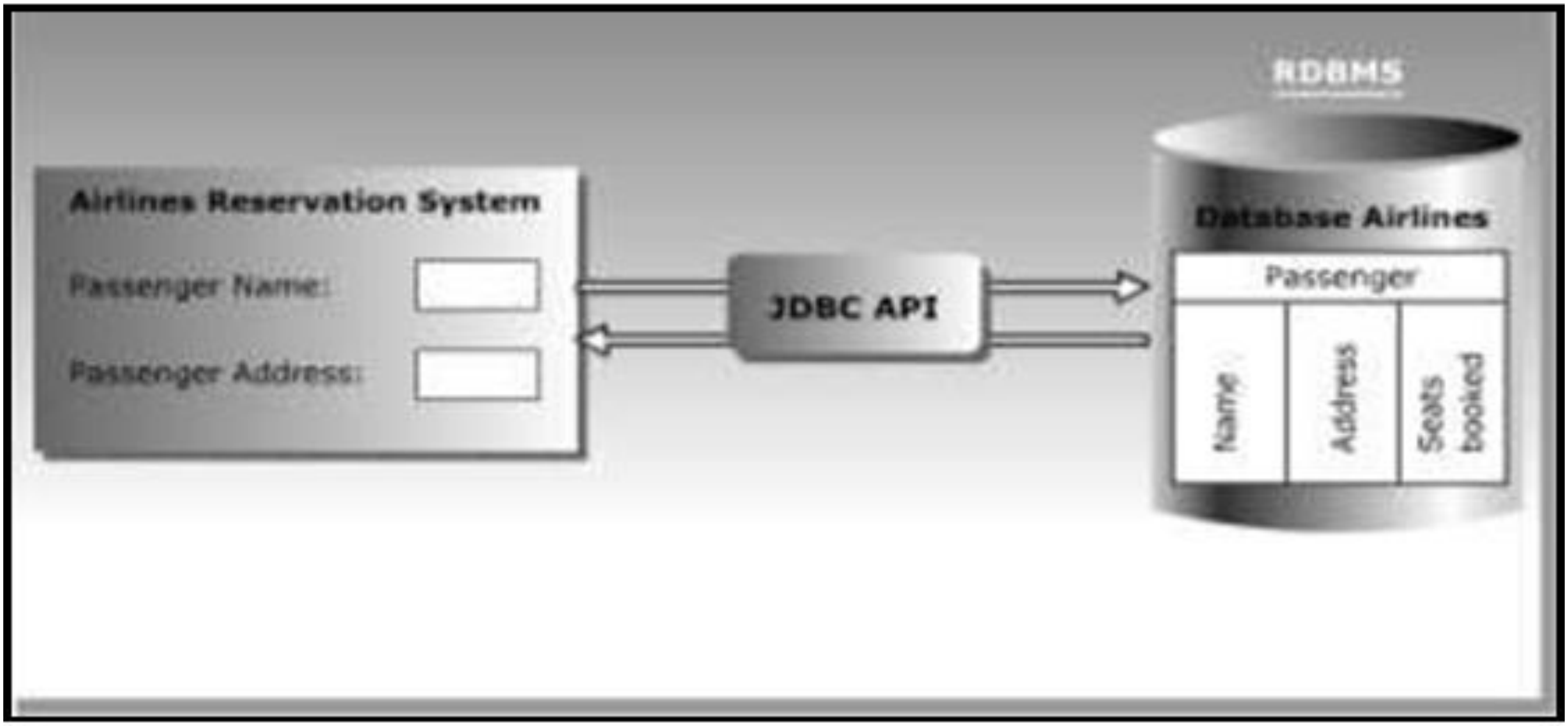
Consider a scenario where you have to develop an application for an airlines company that maintains a record of daily transactions. You install SQL Server as an RDBMS, design the airlines database, and ask the airlines personnel to use it. Will the database alone be of any use to the airlines personnel?

The answer will be negative. The task of updating data in the SQL Server database by using SQL statements alone will be a tedious process. An application will need to be developed that is user friendly and provides the options to retrieve, add, and modify data at the touch of a key to a client.

Therefore, you need to develop an application that communicates with a database to perform the following tasks:

- Store and update the data in the database.
- Retrieve the data stored in the database and present it to users in a proper format.

The following figure shows Airline Reservation System developed in Java that interacts with the Airlines database using the JDBC API.



The Database Connectivity Using the JDBC API

JDBC Architecture

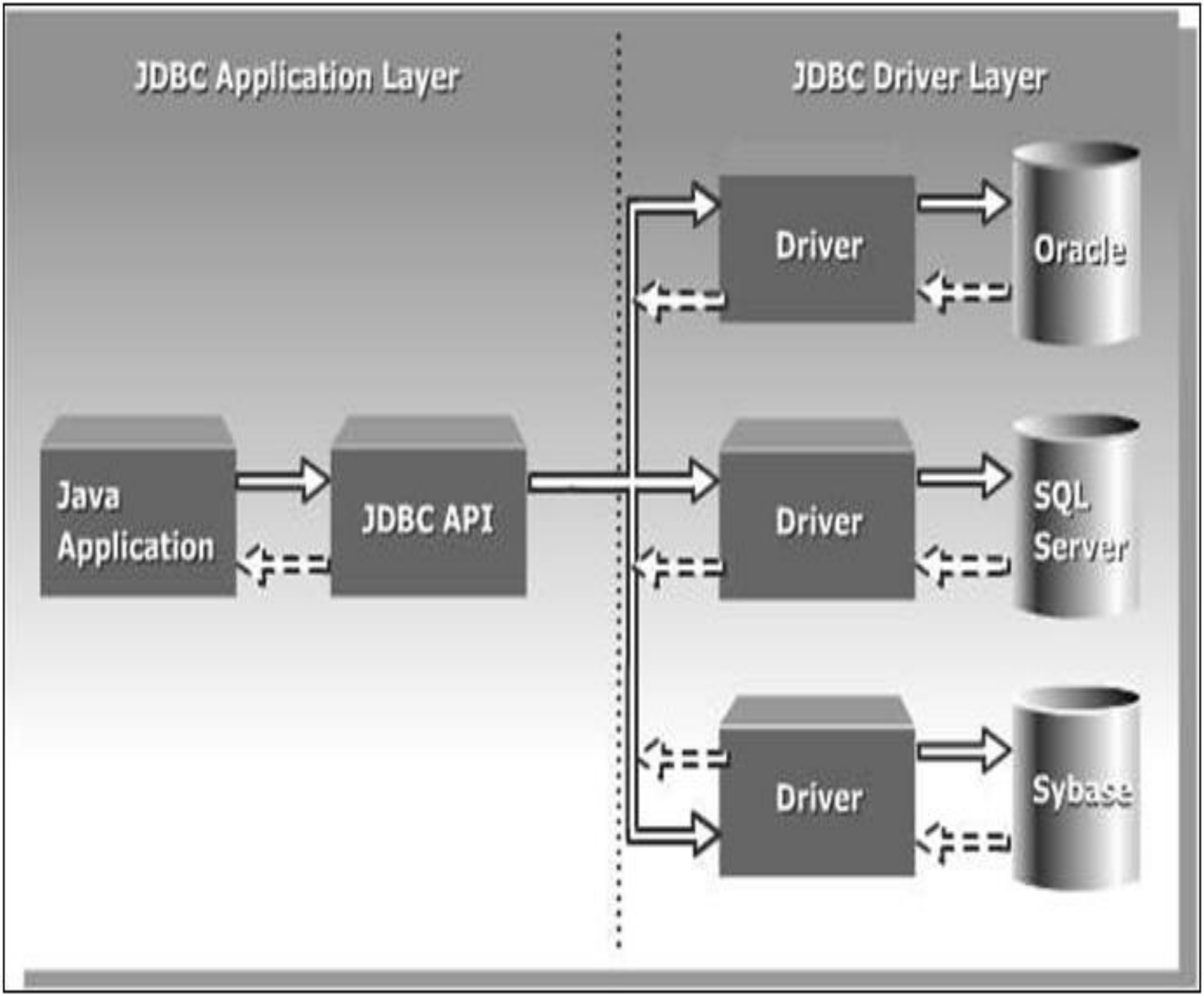
Java applications cannot communicate directly with a database to submit data and retrieve the results of queries. This is because a database can interpret only SQL statements and not Java language statements. Therefore, We need a mechanism to translate Java statements into SQL statements. The JDBC architecture provides the mechanism for this kind of translation.

The JDBC architecture has the following two layers:

JDBC application layer: Signifies a Java application that uses the JDBC API to interact with the JDBC drivers. A JDBC driver is the software that a Java application uses to access a database.

JDBC driver layer: Acts as an interface between a Java application and a database. This layer contains a driver, such as the SQL Server driver or the Oracle driver, which enables connectivity to a database. A driver sends the request of a Java application to the database. Once this request is processed, the database sends the response back to the driver. The response is then translated and sent to the JDBC API by the driver. The JDBC API finally forwards this response to the Java application.

The following figure shows the JDBC architecture.



The JDBC Architecture

Just a minute:

Identify the two layers of the JDBC architecture.

Answer:

The two layers of the JDBC architecture are:

1. *JDBC application layer*
2. *JDBC driver layer*

Identifying the Types of JDBC Drivers

While developing JDBC applications, you need to use JDBC drivers to convert queries into a form that a particular database can interpret. The JDBC driver also retrieves the result of SQL statements and converts the result into equivalent JDBC API class objects that the Java application uses.

Because the JDBC driver only takes care of the interactions with the database, any change made to the database does not affect the application. JDBC supports the following types of drivers:

➢ JDBC-ODBC Bridge driver	(Type 1)
➢ Native-API driver	(Type 2)
➢ Network Protocol driver	(Type 3)
➢ Native Protocol driver	(Type 4)

The JDBC-ODBC Bridge Driver

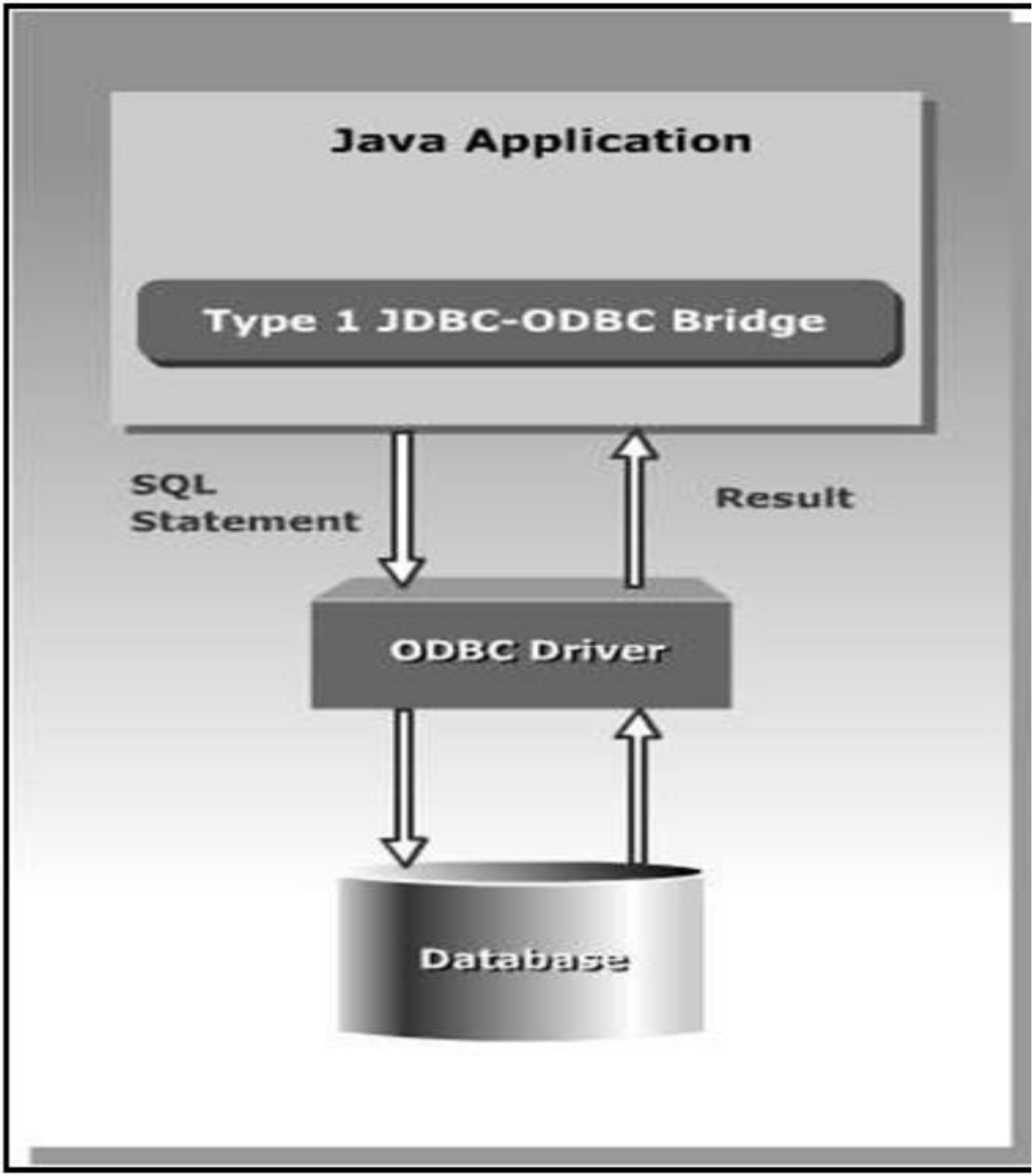
The JDBC-ODBC Bridge driver is called the Type 1 driver. The JDBC-ODBC Bridge driver converts the JDBC method calls into the *Open Database Connectivity (ODBC)* function calls.

ODBC is an open standard API to communicate with databases. The JDBC-ODBC Bridge driver enables a Java application to use any database that supports the ODBC driver.

A Java application cannot interact directly with the ODBC driver. Therefore, the application uses the JDBC-ODBC Bridge driver that works as an interface between the application and the ODBC driver.

To use the JDBC-ODBC Bridge driver, you need to have the ODBC driver installed on the client computer. The JDBC-ODBC Bridge driver is usually used in standalone applications.

The following figure shows how the JDBC-ODBC Bridge driver works.



The JDBC-ODBC Bridge Driver

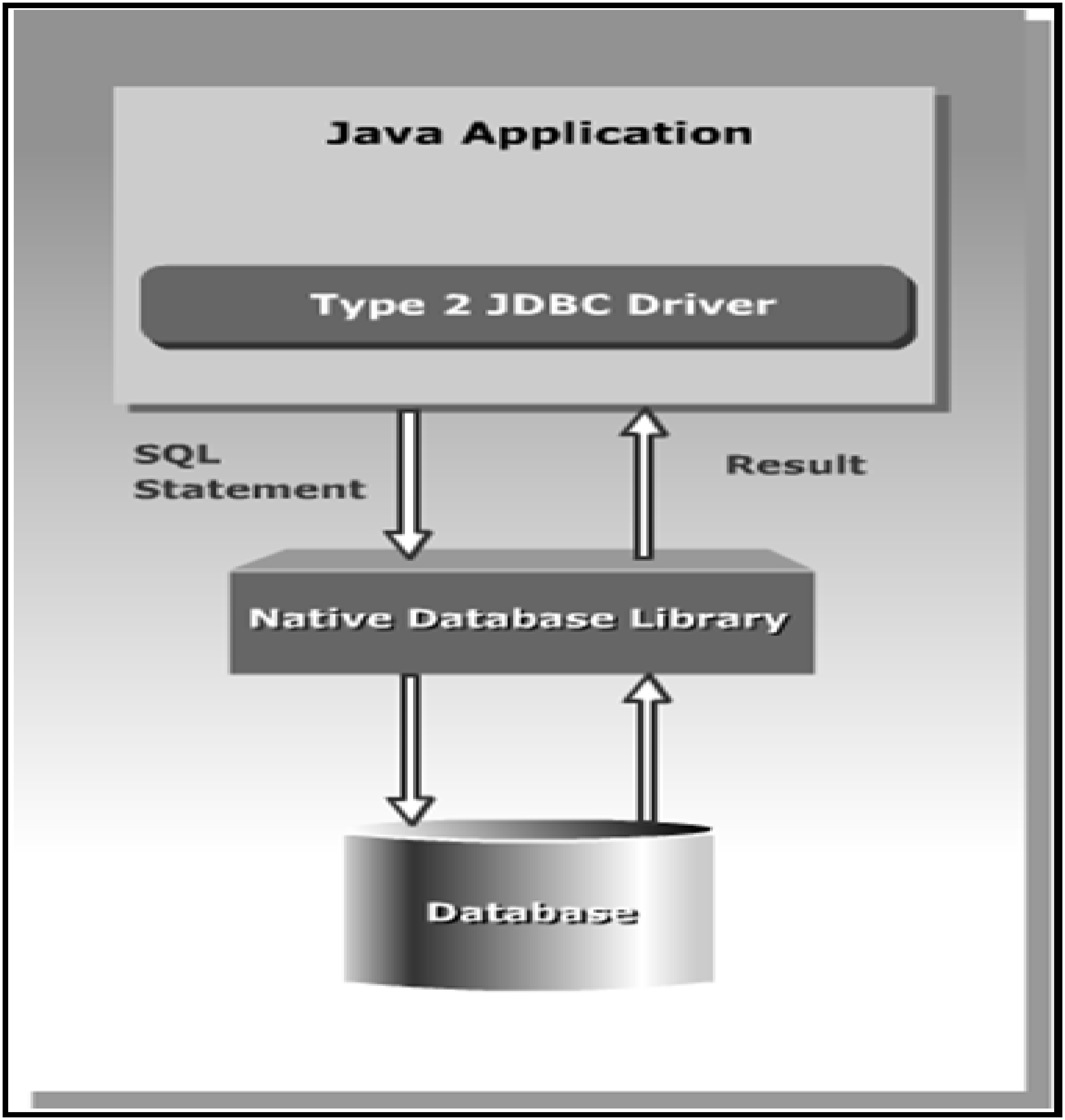
The Native-API Driver

The Native-API driver is called the Type 2 driver. It uses the local native libraries provided by the database vendors to access databases.

The JDBC driver maps the JDBC calls to the native method calls, which are passed to the local native *Call Level Interface (CLI)*. This interface consists of functions written in the C language to access databases. To use the Type 2 driver, CLI needs to be loaded on the client computer.

Unlike the JDBC-ODBC Bridge driver, the Native-API driver does not have an ODBC intermediate layer. As a result, this driver has a better performance than the JDBC-ODBC Bridge driver and is usually used for network-based applications.

The following figure shows how the Native-API driver works.



The Native-API Driver

The Network Protocol Driver

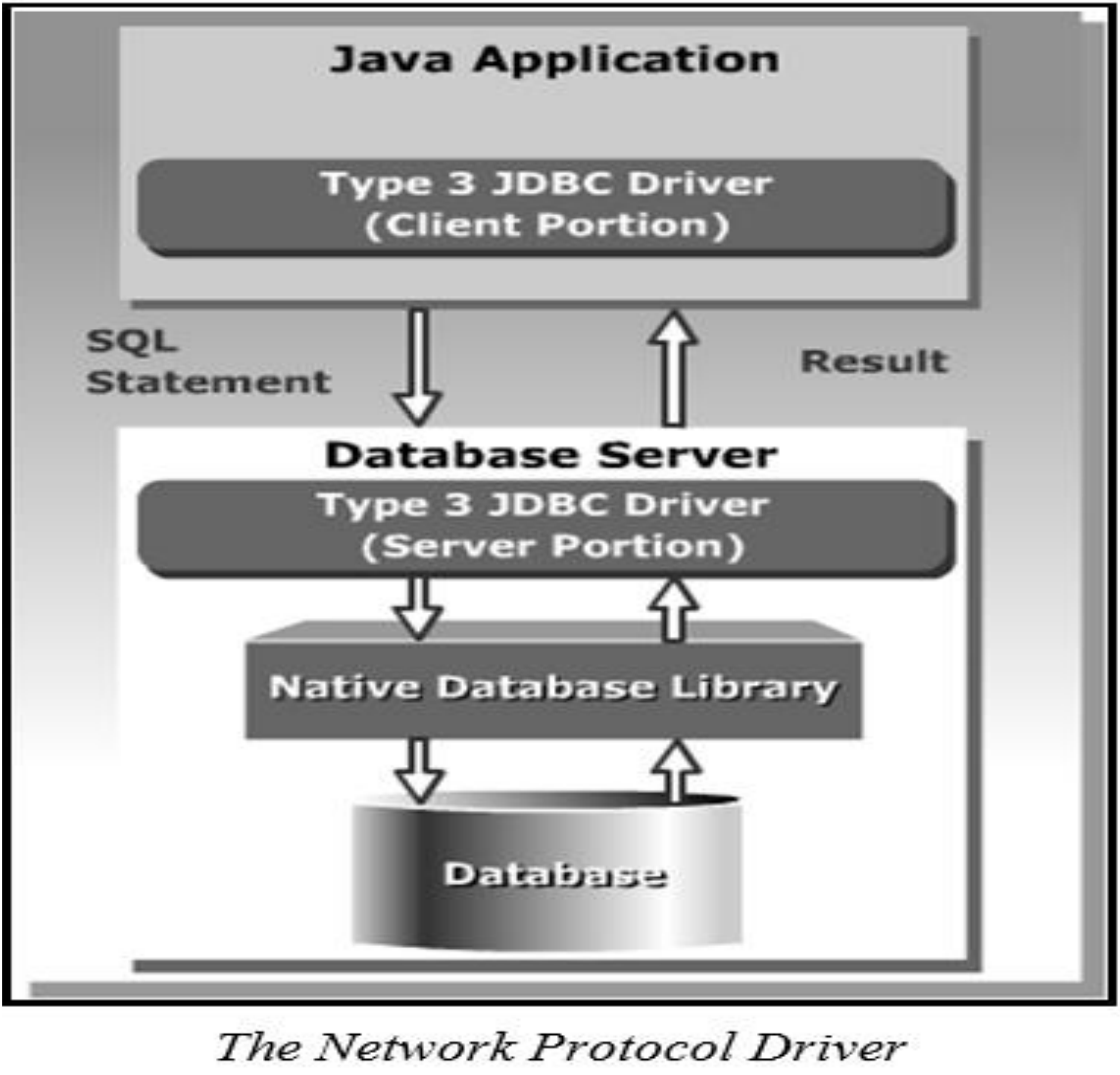
The Network Protocol driver is called the Type 3 driver. The Network Protocol driver consists of client and server portions.

The client portion contains pure Java functions, and the server portion contains Java and native methods.

The Java application sends JDBC calls to the Network Protocol driver client portion, which in turn, translates JDBC calls into database calls.

The database calls are sent to the server portion of the Network Protocol driver that forwards the request to the database. When you use the Network Protocol driver, CLI native libraries are loaded on the server.

The following figure shows how the Network Protocol driver works.

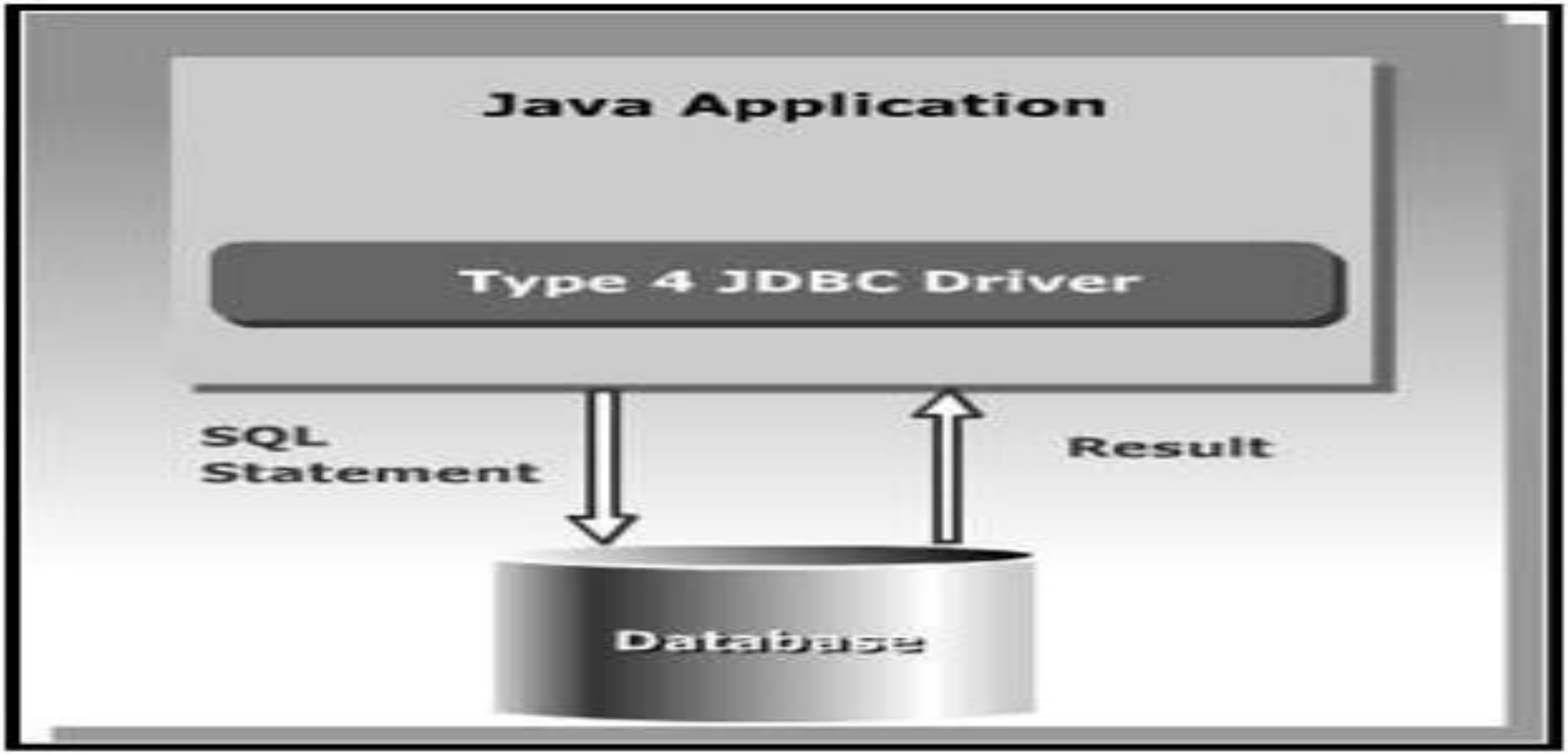


The Network Protocol Driver

The Native Protocol Driver

The Native Protocol driver is called the Type 4 driver. It is a Java driver that interacts with the database directly using a vendor-specific network protocol.

Unlike the other JDBC drivers, you do not require to install any vendor-specific libraries to use the Type 4 driver. Each database has the specific Type 4 drivers. The following figure shows how the Native Protocol driver works.



The Native Protocol Driver

Using JDBC API

You need to use database drivers and the JDBC API while developing a Java application to retrieve or store data in a database. The JDBC API classes and interfaces are available in the java.sql and javax.sql packages.

The classes and interfaces perform a number of tasks, such as establish and close a connection with a database, send a request to a database, retrieve data from a database, and update data in a database.

The classes and interfaces that are commonly used in the JDBC API are:

- The DriverManager class: Loads the driver for a database.
- The Driver interface: Represents a database driver. All JDBC driver classes must implement the Driver interface.
- The Connection interface: Enables you to establish a connection between a Java application and a database.
- The Statement interface: Enables you to execute SQL statements.
- The ResultSet interface: Represents the information retrieved from a database.
- The SQLException class: Provides information about exceptions that occur while interacting with databases.

To query a database and display the result using Java applications, you need to:

1. Load a driver.

2. Connect to a database.

3. Create and execute JDBC statements.

4. Handle SQL exceptions.

Loading a Driver

The first step to develop a JDBC application is to load and register the required driver using the driver manager. You can load and register a driver by:

❑ Using the forName() method of the java.lang.Class class.

❑ Using the registerDriver() static method of the DriverManager class.

Using the forName() Method

The forName() method loads the JDBC driver and registers the driver. The following signature is used to load a JDBC driver to access a database:

```
Class.forName("package.sub-package.sub-package.DriverClassName");
```

You can load the JDBC-Type 4 driver for SQL Server using the following code snippet:

```
Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver");
```

Using the registerDriver() Method

You can create an instance of the JDBC driver and pass the reference to the registerDriver() method. The following syntax is used for creating an instance of the JDBC driver:

```
Driver d=new <driver name>;
```

You can use the following code snippet to create an instance of the JDBC driver:

```
Driver d = new
com.microsoft.sqlserver.jdbc.SQLServerDriver();
```

Once you have created the Driver object, call the registerDriver() method to register the driver, as shown in the following code snippet:

```
DriverManager.registerDriver(d);
```

Connecting to a Database

You need to create an object of the Connection interface to establish a connection of the Java application with a database.

You can create multiple Connection objects in a Java application to access and retrieve data from multiple databases.

The DriverManager class provides the getConnection() method to create a Connection object.
The getConnection() method is an overloaded method that has the following three forms:

- public static Connection getConnection(String url)
- public static Connection getConnection(String url, Properties info)
- public static Connection getConnection(String url, String user, String password)

The getConnection (String url)method accepts the JDBC URL of the database, which you need to access as a parameter. You can use the following code snippet to connect a database using the getConnection() method with a single parameter:

```
String url =
"jdbc:sqlserver://localhost;user=MyUserName;password=password@123";

Connection con = DriverManager.getConnection(url);
```

The following signature is used for a JDBC Uniform Resource Location (URL) that is passed as a parameter to the getConnection() method:

```
<protocol>:<subprotocol>:<subname>
```

A JDBC URL has the following three components:

- Protocol: Indicates the name of the protocol that is used to access a database. In JDBC, the name of the access protocol is always jdbc.

- Subprotocol: Indicates the vendor of Relational Database Management System (RDBMS). For example, if you use the JDBC-Type 4 driver of SQL Server to access its database, the name of subprotocol will be sqlserver.

- Subname: Indicates *Data Source Information* that contains the database information, such as the location of the database server, name of a database, user name, and password, to access a database server.

Creating and Executing JDBC Statements

Once a connection is created, you need to write the JDBC statements that are to be executed. You need to create a Statement object to send requests to a database and retrieve results from the same.

The Connection object provides the createStatement() method to create a Statement object. You can use the following code snippet to create a Statement object:

```
Connection con=DriverManager.getConnection(
"jdbc:sqlserver://sqlserver01;databaseName=
Library;user=sa;password=cimagepatna");

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

You can use static SQL statements to send requests to a database. The SQL statements that do not contain runtime parameters are called static SQL statements.

You can send SQL statements to a database using the Statement object. The Statement interface contains the following methods to send the static SQL statements to a database:

ResultSet executeQuery(String str): Executes an SQL statement and returns a single object of the type, ResultSet. This object provides you the methods to access data from a result set. The executeQuery() method should be used when you need to retrieve data from a database table using the SELECT statement. The syntax to use the executeQuery() method is:

```
Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery(<SQL statement>);
```

In the preceding syntax, stmt is a reference to the object of the Statement interface. The executeQuery() method executes an SQL statement, and the result retrieved from a database is stored in rs that is the ResultSet object.

- **int executeUpdate(String str):** Executes SQL statements and returns the number of rows that are affected after processing the SQL statement.

- When you need to modify data in a database table using the Data Manipulation Language (DML) statements, INSERT, DELETE, and UPDATE, you can use the executeUpdate() method. The syntax to use the executeUpdate() method is:

```
Statement stmt = con.createStatement();
int count = stmt.executeUpdate(<SQL statement>);
```

In the preceding syntax, the executeUpdate() method executes an SQL statement and the number of rows affected in a database is stored in count that is the int type variable.

boolean execute(String str): Executes an SQL statement and returns a boolean value. You can use this method when you are dynamically executing an unknown SQL string. The execute() method returns true if the result of the SQL statement is an object of ResultSet else it returns false. The syntax to use the execute() method is:

```
Statement stmt = con.createStatement();
stmt.execute(<SQL statement>);
```

You can use the DML statements, INSERT, UPDATE, and DELETE, in Java applications to modify the data stored in the database tables.

You can also use the Data Definition Language (DDL) statements, CREATE, ALTER, and DROP, in Java applications to define or change the structure of database objects.

Querying a Table

Using the SELECT statement, you can retrieve data from a table. The SELECT statement is executed using the executeQuery() method and returns the output in the form of a ResultSet object. You can use the following code snippet to retrieve data from the Authors table:

```
String str = "SELECT * FROM Authors";

Statement stmt = con.createStatement();

ResultSet rs = stmt.executeQuery(str);
```

In the preceding code snippet, str contains the SELECT statement that retrieves data from the Authors table.

The result is stored in the ResultSet object, rs.

When you need to retrieve selected rows from a table, the condition to retrieve the rows is specified in the WHERE clause of the SELECT statement. You can use the following code snippet to retrieve selected rows from the Authors table:

```
String str = "SELECT * FROM Authors WHERE
city='London'";

Statement stmt = con.createStatement();

ResultSet rs = stmt.executeQuery(str);
```

In the preceding code snippet, the executeQuery() method retrieves the details from the Authors table for a particular city.

Inserting Rows in a Table

You can add rows to an existing table using the INSERT statement. The executeUpdate() method enables you to add rows in a table. You can use the following code snippet to insert a row in the Authors table:

```
String str = " INSERT INTO Authors (au_id, au_name,
phone, address, city, state, zip) VALUES ('A004' , 'Ringer
Albert' , '8018260752' , ' 67 Seventh Av. ' , 'Salt Lake
City' , 'UT' , '100000078' )";

Statement stmt = con.createStatement();
int count = stmt.executeUpdate(str);
```

In the preceding code snippet, str contains the INSERT statement that you need to send to a database. The object of the Statement interface, stmt, executes the INSERT statement using the executeUpdate() method and returns the number of rows inserted in the table to the count variable.

Updating Rows in a Table

You can modify the existing information in a table using the UPDATE statement. You can use the following code snippet to modify a row in the Authors table:

```
String str = "UPDATE Authors SET address='10932
Second Av.' where au_id= 'A001'";

Statement stmt = con.createStatement();
int count = stmt.executeUpdate(str);
```

In the preceding code snippet, str contains the UPDATE statement that you need to send to a database.

The Statement object executes this statement using the executeUpdate() method and returns the number of rows modified in the table to the count variable.

Deleting Rows from a Table

You can delete the existing information from a table using the DELETE statement. You can use the following code snippet to delete a row from the Authors table:

```
String str = "DELETE FROM Authors WHERE
au_id='A005'";

Statement stmt = con.createStatement();

int count = stmt.executeUpdate(str);
```

In the preceding code snippet, str contains the DELETE statement that you need to send to a database.

The Statement object executes this statement using the executeUpdate() method and returns the number of rows deleted from the table to the count variable.

Handling SQL Exceptions

The java.sql package provides the SQLException class, which is derived from the java.lang.Exception class.

SQLException is thrown by various methods in the JDBC API and enables you to determine the reason for the errors that occur while connecting a Java application to a database.

You can catch SQLException in a Java application using the try and catch exception handling block.

The SQLException class provides the following error information:

- **Error message**: Is a string that describes error.

- **Error code**: Is an integer value that is associated with an error. The error code is vendor specific and depends upon the database in use.

- **SQL state**: Is an *XOPEN* error code that identifies the error. Various vendors provide different error messages to define the same error. As a result, an error may have different error messages.

The XOPEN error code is a standard message associated with an error that can identify the error across multiple databases.

The SQLException class contains various methods that provide error information. Some of the methods in the SQLException class are:

- int getErrorCode(): Returns the error code associated with the error occurred.
- String getSQLState(): Returns SQL state for the SQLException object.
- SQLException getNextException(): Returns the next exception in

  the chain of exceptions. You can use the following code

  snippet to catch SQLException:

```
try
{
    String str = "DELETE FROM Authors WHERE au_id='A002'";
    Statement stmt = con.createStatement();
    int count = stmt.executeUpdate(str);
}
catch(SQLException sqlExceptionObject)
{
    System.out.println("Display Error Code");
    System.out.println("SQL            Exception            "+
       sqlExceptionObject.getErrorCode());
}
```

In the preceding code snippet, if the DELETE statement throws SQLException, then it is handled by the catch block. sqlExceptionObject is an object of the SQLException class and is used to invoke the getErrorCode() method.

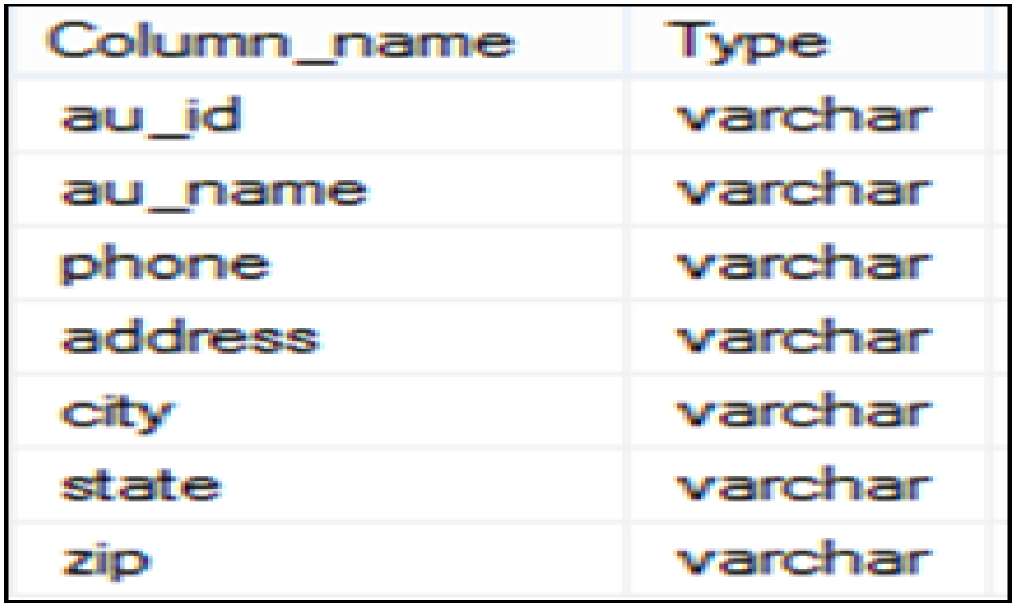
Activity : Creating a JDBC Application to Query a Database

Problem Statement

Create an application to retrieve information (author ID, name, and city) about the authors who are living in the city where the city name begins with the letter 'S'.

Prerequisite

You need to ensure that the **Library** database exists and comprises the **Authors** table. In addition, the structure of the **Authors** table should be similar to the structure, as shown in the following figure.



Column_name	Type
au_id	varchar
au_name	varchar
phone	varchar
address	varchar
city	varchar
state	varchar
zip	varchar

The Structure of the Authors Table

Accessing Result Sets

When you execute a query to retrieve data from a table using a Java application, the output of the query is stored in a ResultSet object in a tabular format. A ResultSet object maintains a *cursor* that enables you to move through the rows stored in the ResultSet object. By default, the ResultSet object maintains a cursor that moves in the forward direction only.

As a result, it moves from the first row to the last row in ResultSet. In addition, the default ResultSet object is not updatable, which means that the rows cannot be updated in the default object. The cursor in the ResultSet object initially points before the first row.

Types of Result Sets

You can create various types of ResultSet objects to store the output returned by a database after executing SQL statements.
The various types of ResultSet objects are:

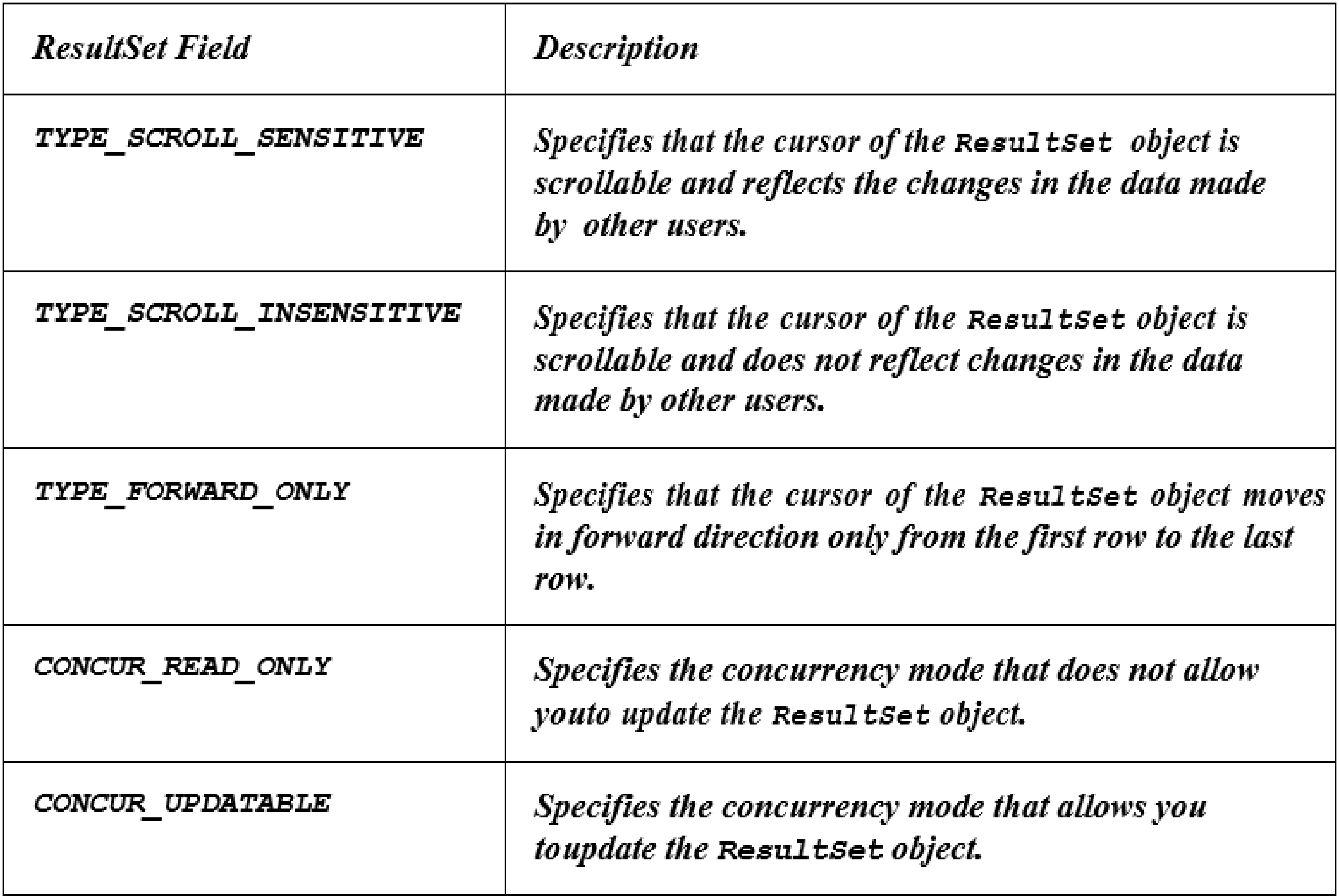
Read only: Allows you to only read the rows in a ResultSet object.

Forward only: Allows you to move the result set cursor from the first row to the last row in the forward direction only.

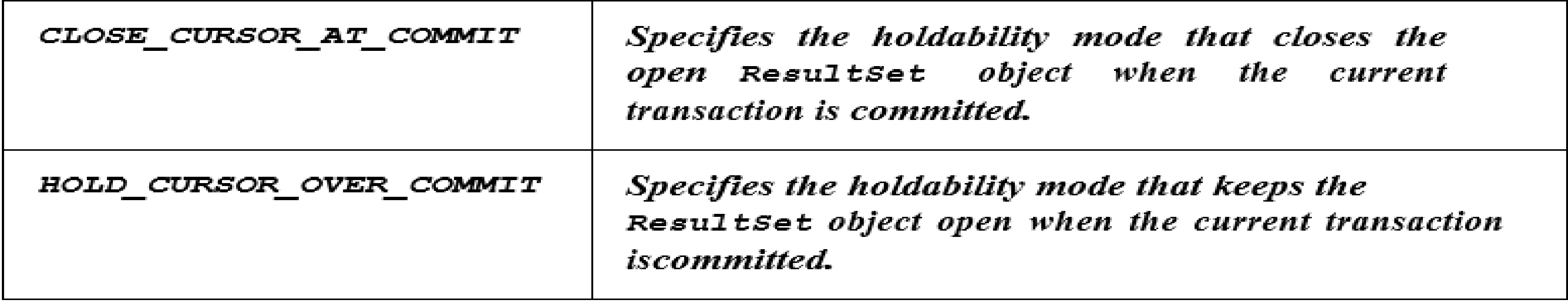
Scrollable: Allows you to move the result set cursor forward or backward through the result set.
Updatable: Allows you to update the result set rows retrieved from a database table.

You can specify the type of a ResultSet object using the createStatement() method of the Connection interface. The createStatement() method accepts the ResultSet fields as parameters to create different types of the ResultSet object.

The following table lists various fields of the ResultSet interface.



ResultSet Field	*Description*
TYPE_SCROLL_SENSITIVE	*Specifies that the cursor of the* **ResultSet** *object is scrollable and reflects the changes in the data made by other users.*
TYPE_SCROLL_INSENSITIVE	*Specifies that the cursor of the* **ResultSet** *object is scrollable and does not reflect changes in the data made by other users.*
TYPE_FORWARD_ONLY	*Specifies that the cursor of the* **ResultSet** *object moves in forward direction only from the first row to the last row.*
CONCUR_READ_ONLY	*Specifies the concurrency mode that does not allow youto update the* **ResultSet** *object.*
CONCUR_UPDATABLE	*Specifies the concurrency mode that allows you toupdate the* **ResultSet** *object.*



CLOSE_CURSOR_AT_COMMIT	*Specifies the holdability mode that closes the open ResultSet object when the current transaction is committed.*
HOLD_CURSOR_OVER_COMMIT	*Specifies the holdability mode that keeps the ResultSet object open when the current transaction iscommitted.*

The Fields of the ResultSet Interface

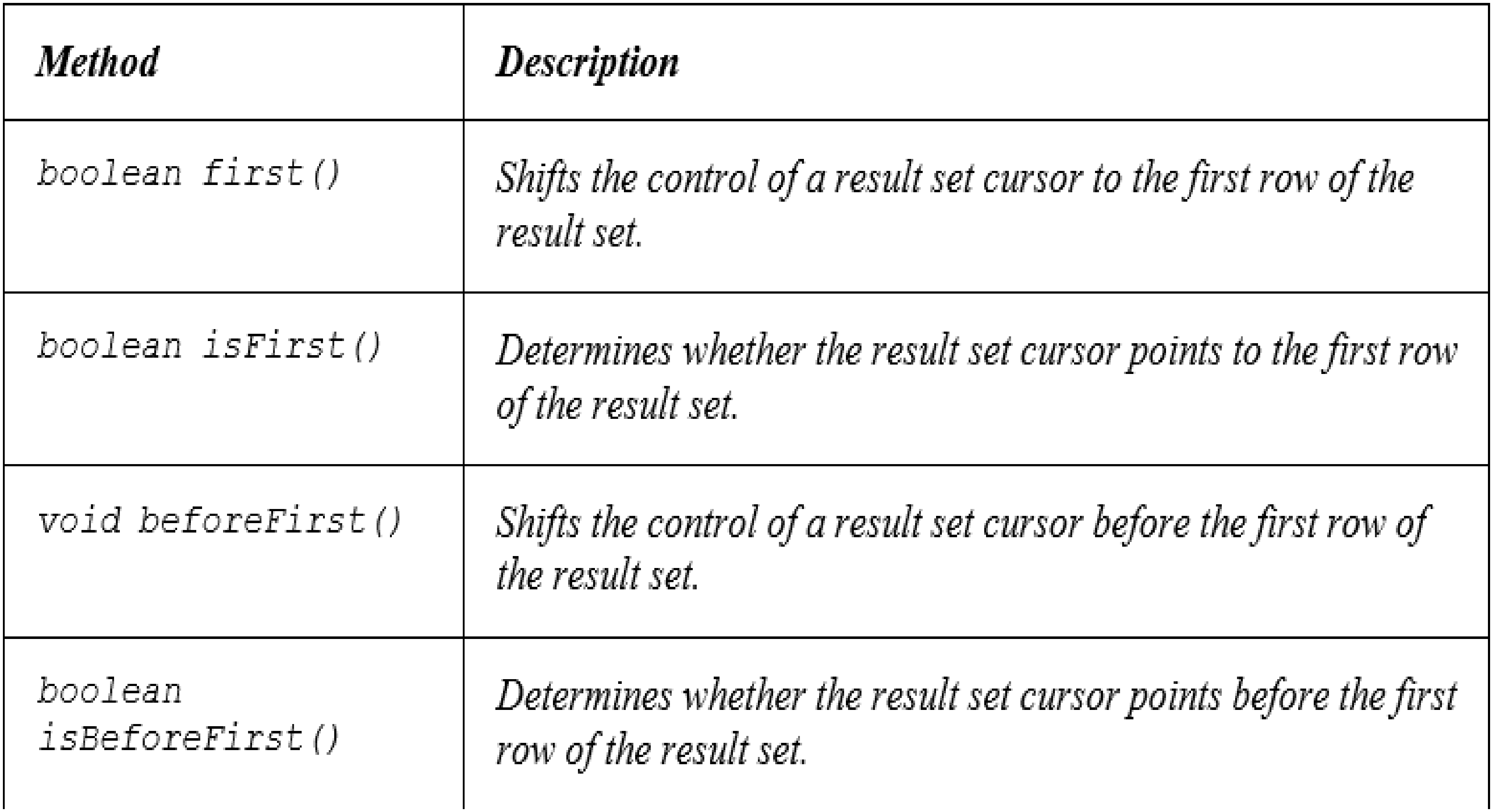
The createStatement() method has the following overloaded forms:

- Statement createStatement(): Does not accept any parameter. This method creates a Statement object for sending SQL statements to the database. It creates the default ResultSet object that only allows forward scrolling.
- Statementcreate Statement(int resultSetType, int resultSetConcurrency): Accepts two parameters. The first parameter indicates the ResultSet type that determines whether or not a result set cursor is scrollable or forward only. The second parameter indicates the concurrency mode for the result set that determines whether the data in result set is updateable or read only. This method creates a ResultSet object with the given type and concurrency.
- Statement createStatement(int resultSetType, int resultSetConcurrency, int resultSetHoldability): Accepts three parameters. Apart from the ResultSet types and the concurrency mode, this method also accepts a third parameter. This parameter indicates the holdability mode for the open result set object. This method creates a ResultSet object with the given type, concurrency, and the holdability mode.

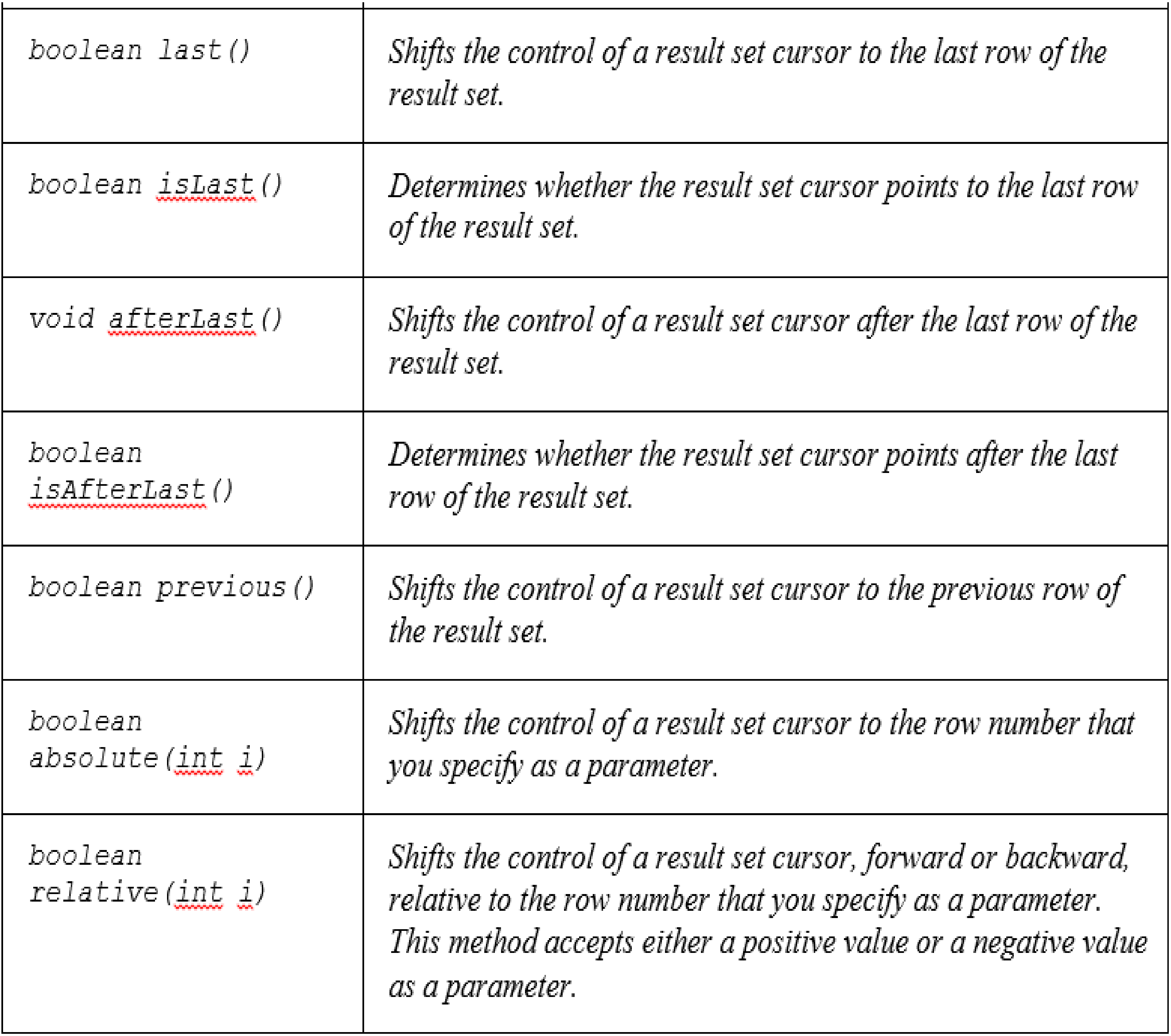
Methods of the ResultSet Interface

The ResultSet interface contains various methods that enable you to move the cursor through the result set.

The following table lists some methods of the ResultSet interface that are commonly used.



Method	***Description***
`boolean first()`	*Shifts the control of a result set cursor to the first row of the result set.*
`boolean isFirst()`	*Determines whether the result set cursor points to the first row of the result set.*
`void beforeFirst()`	*Shifts the control of a result set cursor before the first row of the result set.*
`boolean isBeforeFirst()`	*Determines whether the result set cursor points before the first row of the result set.*



`boolean last()`	*Shifts the control of a result set cursor to the last row of the result set.*
`boolean isLast()`	*Determines whether the result set cursor points to the last row of the result set.*
`void afterLast()`	*Shifts the control of a result set cursor after the last row of the result set.*
`boolean isAfterLast()`	*Determines whether the result set cursor points after the last row of the result set.*
`boolean previous()`	*Shifts the control of a result set cursor to the previous row of the result set.*
`boolean absolute(int i)`	*Shifts the control of a result set cursor to the row number that you specify as a parameter.*
`boolean relative(int i)`	*Shifts the control of a result set cursor, forward or backward, relative to the row number that you specify as a parameter. This method accepts either a positive value or a negative value as a parameter.*

The Methods of the ResultSet Interface

You can create a scrollable result set that scrolls backward or forward through the rows in the result set.

You can use the following code snippet to create a read-only scrollable result set:

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

You can determine the location of the result set cursor using the methods in the ResultSet interface.

You can use the following code snippet to determine if the result set cursor is before the first row in the result set:

```
if(rs.isBeforeFirst()==true)
System.out.println("Result set cursor is before the first row in
the result set");
```

In the preceding code snippet, rs is the ResultSet object that calls the isBeforeFirst() method.

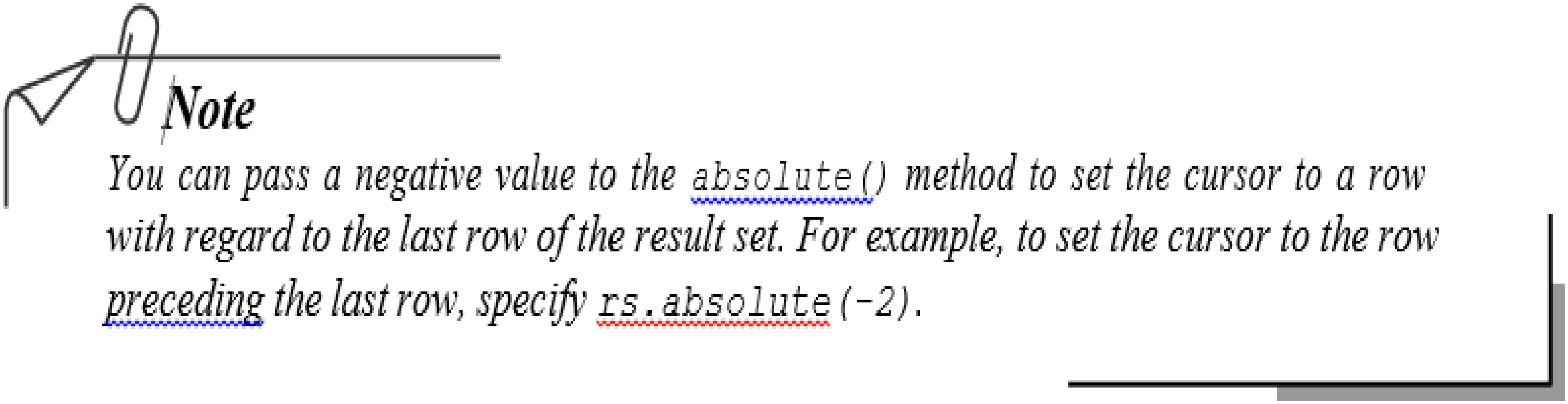
You can move to a particular row, such as first or last, in the result set using the methods in the ResultSet interface. You can use the following code snippet to move the result set cursor to the first row in the result set:

```
if(rs.first()==true)
System.out.println(rs.getString(1) + ", " +
rs.getString(2)+ ", " + rs.getString(3));
```

In the preceding code snippet, rs is the ResultSet object that calls the first() method.
Similarly, you can move the result set cursor to the last row in the result set using the last() method.

To move to any particular row of a result set, you can use the absolute() method. For example, if the result set cursor is at the first row and you want to scroll to the fourth row, you should enter 4 as a parameter when you call the absolute() method, as shown in the following code snippet:

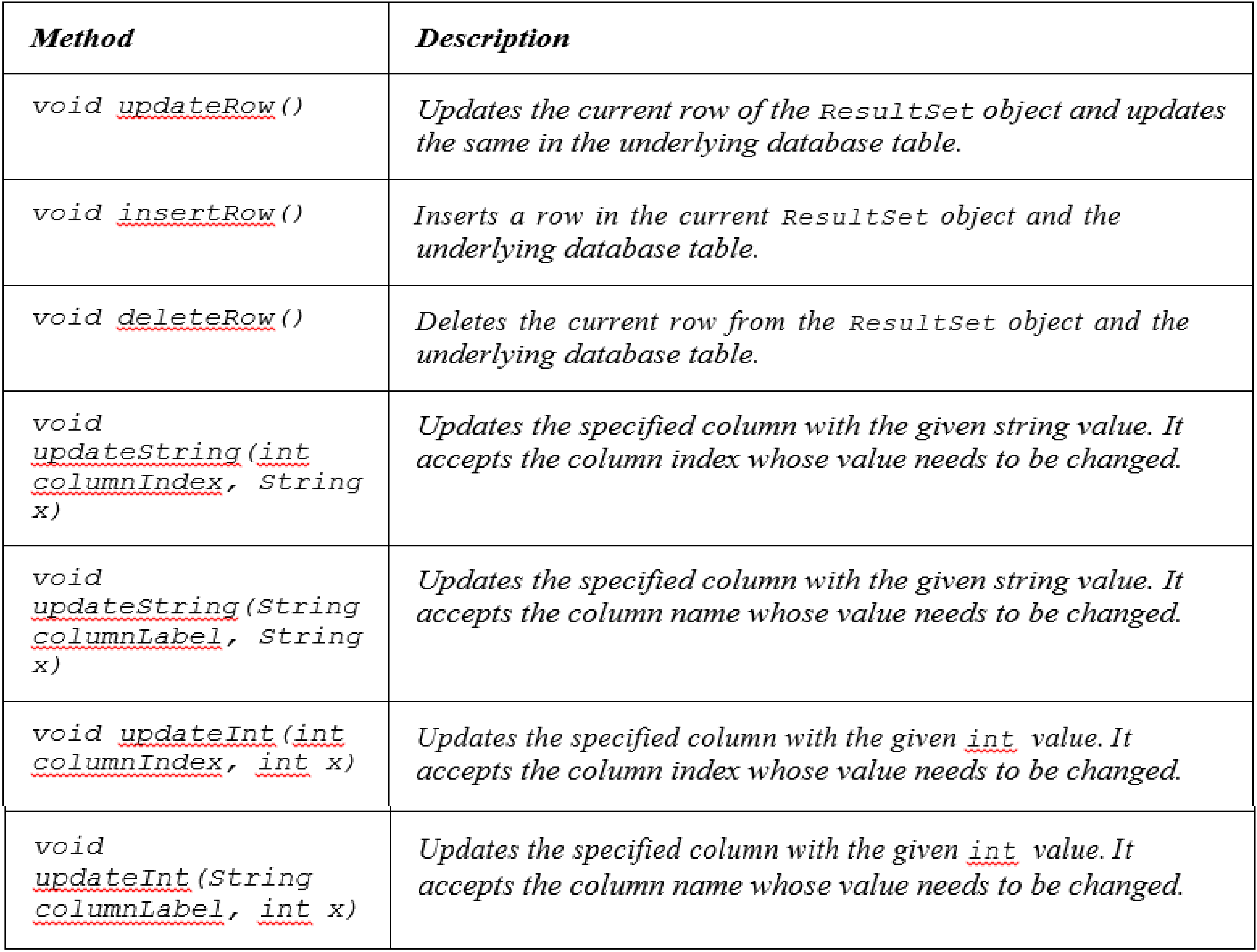
```
System.out.println("Using absolute() method");
rs.absolute(4);
int rowcount = rs.getRow();
System.out.println("row number is " + rowcount);
```



Note

You can pass a negative value to the `absolute()` *method to set the cursor to a row with regard to the last row of the result set. For example, to set the cursor to the row preceding the last row, specify* `rs.absolute(-2)`.

JDBC allows you to create an updateable result set that enables you to modify the rows in the result set. The following table lists some of the methods used with the updatable result set.



Method	*Description*
void updateRow()	*Updates the current row of the* ResultSet *object and updates the same in the underlying database table.*
void insertRow()	*Inserts a row in the current* ResultSet *object and the underlying database table.*
void deleteRow()	*Deletes the current row from the* ResultSet *object and the underlying database table.*
void updateString(int columnIndex, String x)	*Updates the specified column with the given string value. It accepts the column index whose value needs to be changed.*
void updateString(String columnLabel, String x)	*Updates the specified column with the given string value. It accepts the column name whose value needs to be changed.*
void updateInt(int columnIndex, int x)	*Updates the specified column with the given* int *value. It accepts the column index whose value needs to be changed.*
void updateInt(String columnLabel, int x)	*Updates the specified column with the given* int *value. It accepts the column name whose value needs to be changed.*

The Methods Used With the Updatable ResultSet

You can use the following code snippet to modify the information of the author using the updatable result set:

```
Statement stmt = con.createStatement();
stmt =
con.createStatement(ResultSet.TYPE_SCROLL_
SENSITIVE, ResultSet.CONCUR_UPDATABLE);
ResultSet rs = stmt.executeQuery("SELECT
au_id, city, state FROM Authors WHERE
au_id='A004’");
rs.next();
rs.updateString("state",
"NY");
rs.updateString("city",
"Columbia"); rs.updateRow();
```

In the preceding code snippet, the row is retrieved from the Authors table where the author id is A004.

In the retrieved row, the value in the state column is changed to NY and the value in the city column is changed to Columbia.

Just a minute:

Identify the field of the `ResultSet` *interface that allows the cursor to be scrollable and reflects the changes in the data.*

1. *TYPE_SCROLL_SENSITIVE*
2. *TYPE_SCROLL_INSENSITIVE*
3. *CONCUR_READ_ONLY*
4. *CONCUR_UPDATAB*

Answer:

1. *TYPE_SCROLL_SENSITIVE*

Creating Applications Using the PreparedStatement Object

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();
```

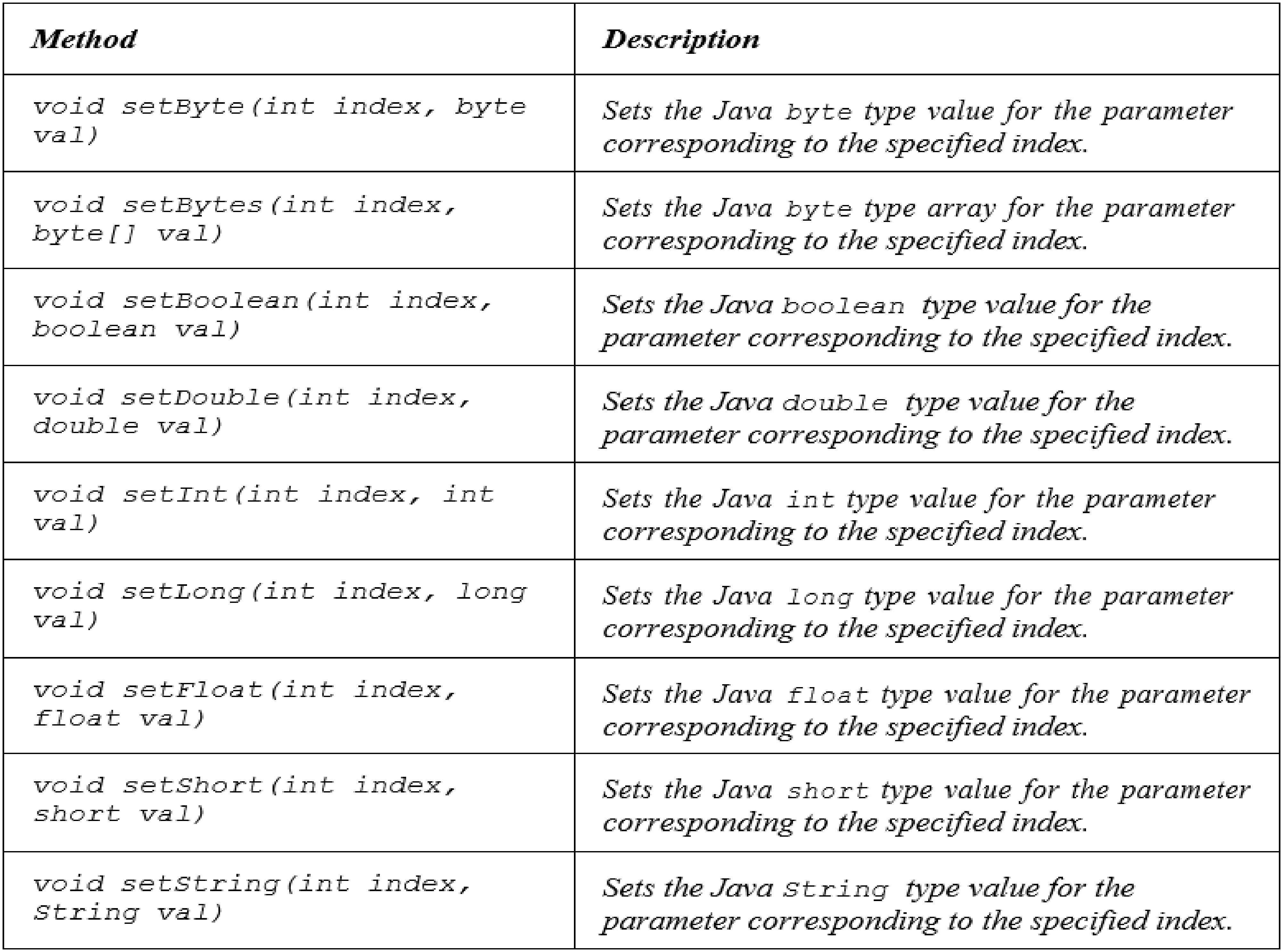
In the preceding code snippet, the first parameter of the setString()method specifies the index value of the ? placeholder, and the second parameter is used to set the value of the ? placeholder.

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.*

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 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.

Deleting Rows

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();
```

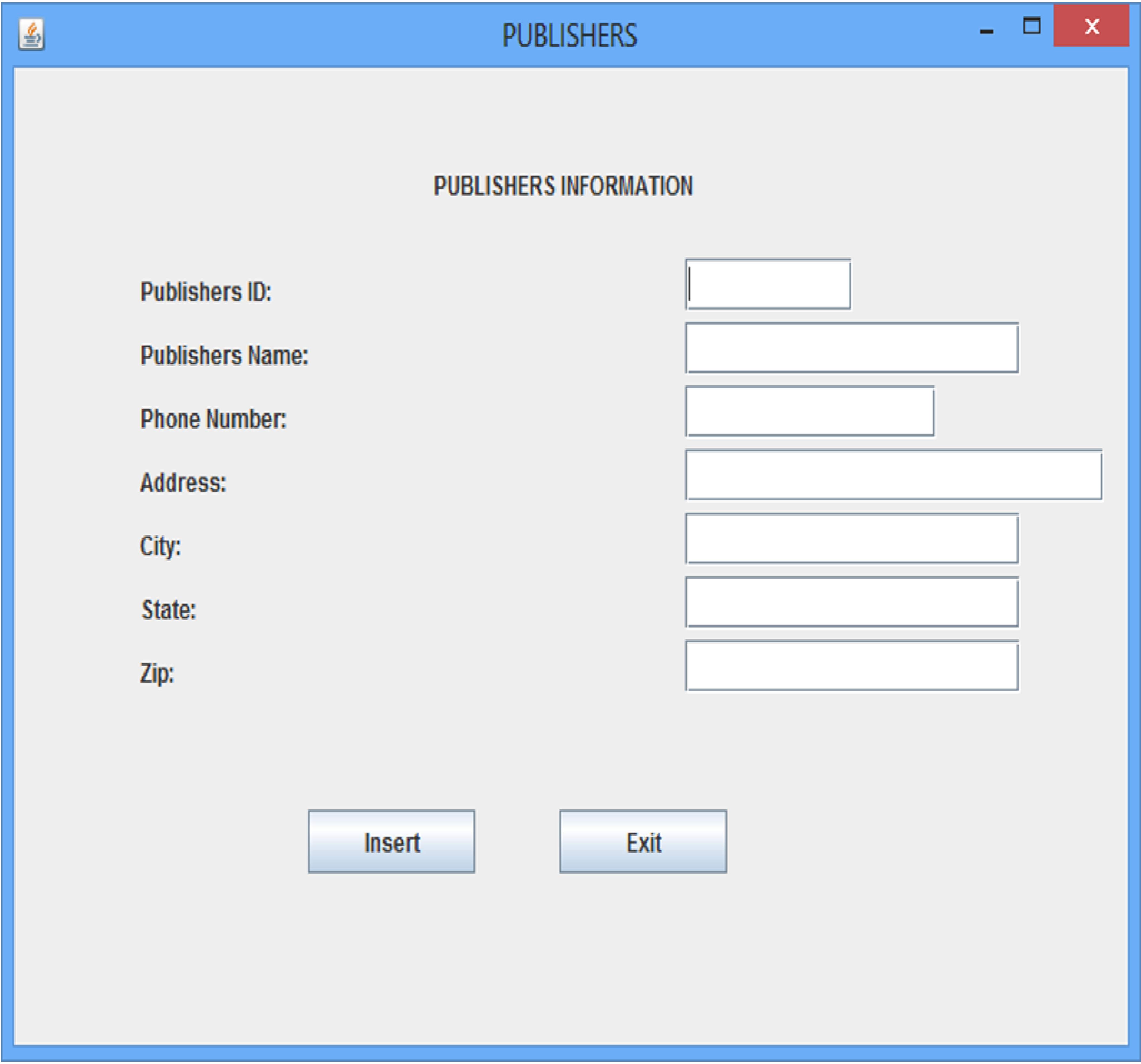
Activity: Creating an Application that Uses the PreparedStatement Object

Problem Statement

The management of City library has decided to computerize the book inventory. The library management has assigned the preceding task to a leading software development company of the US.

The Lead Analyst has assigned the development of the Book Inventory application to Mark, the Senior Software Developer.

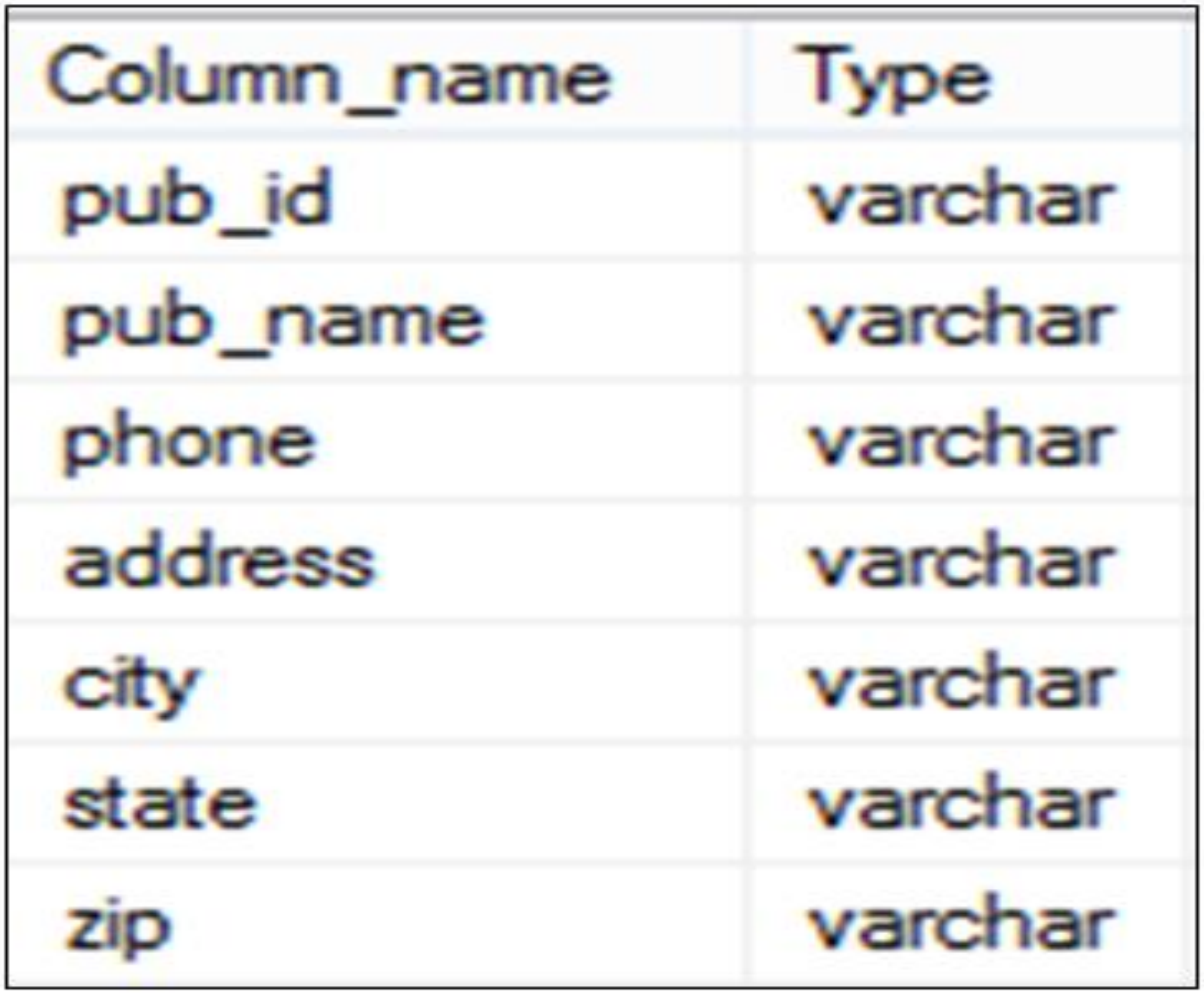
However, in the initial phase, Mark has decided to create an interface that will allow a user to add the details of a new publisher to the publishers table, as shown in the following figure.



The User Interface of the Application

Prerequisite

You need to ensure that the **Library** database exists and comprises the **Publishers** table.
The structure of the **Publishers** table should be similar to the structure, as shown in the following figure.



Column_name	Type
pub_id	varchar
pub_name	varchar
phone	varchar
address	varchar
city	varchar
state	varchar
zip	varchar

The Structure of the Publishers Table

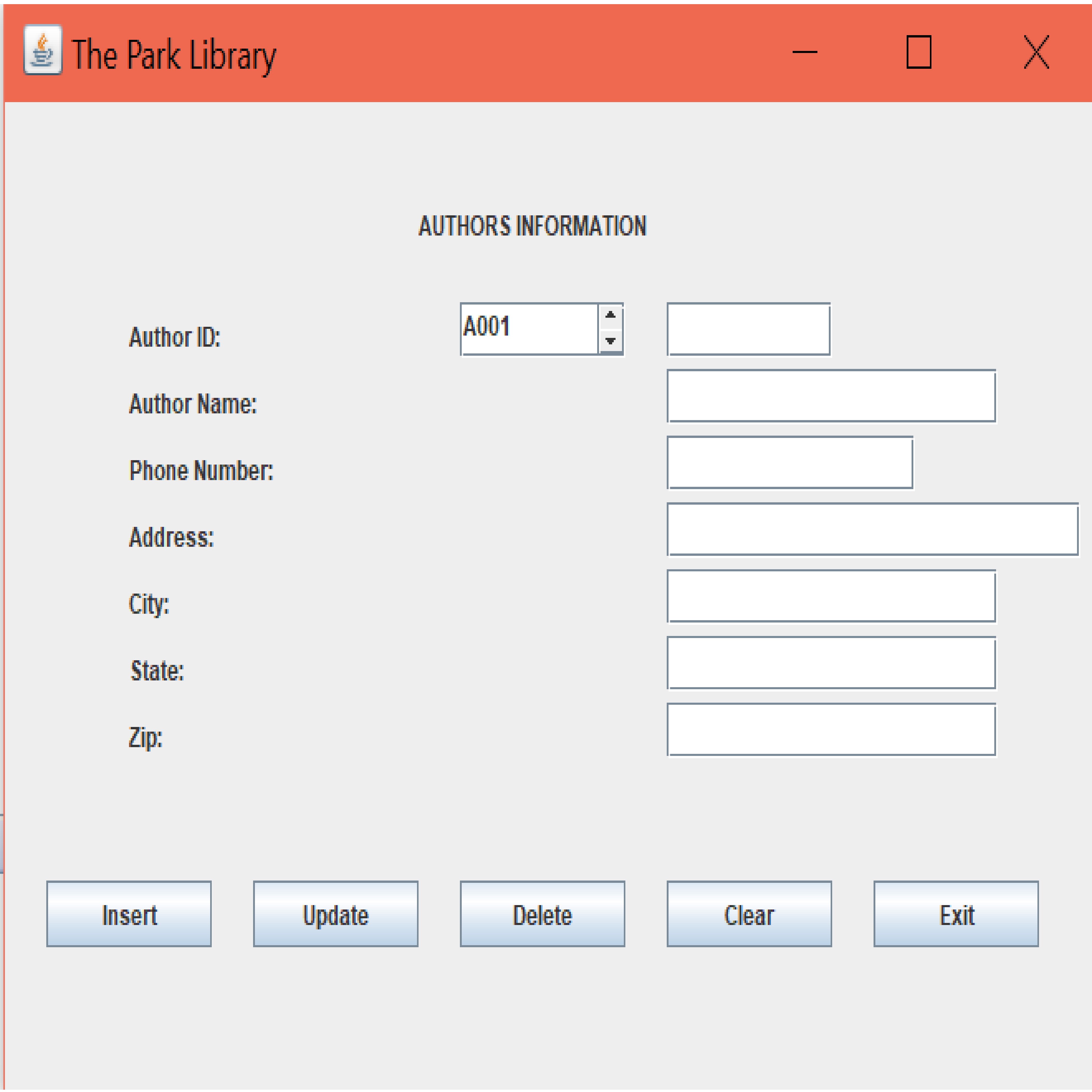
Exercise

The management of Park Library has decided to automate the task of managing the author's details in a database. For this, the library management has assigned the task to a leading software development company of the US.

The Senior Software Developer has assigned the development of the Author Management application to Jessica, the Junior Software Developer.

For this application, Jessica needs to implement the functionality to view, insert, update, and delete an author's information.

Help Jessica to achieve the preceding requirement.



AUTHORS INFORMATION

Author ID: A001

Author Name:

Phone Number:

Address:

City:

State:

Zip:

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.

Live Demo

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.

Note

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.

Live Demo