Chapter 9

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 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.

Objectives

In this chapter, you will learn to:

- Identify the layers in the JDBC architecture
- Identify the types of JDBC drivers
- Use JDBC API
- Access result sets

Identifying the Layers in the JDBC

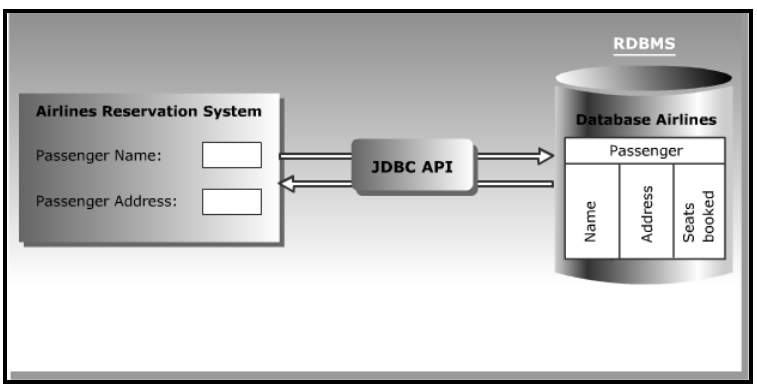
Consider a scenario where you have to develop an application for an airlines company that maintains a record of daily transactions. You install SQL Server as an RDBMS, design the airlines database, and ask the airlines personnel to use it. Will the database alone be of any use to the airlines personnel?

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

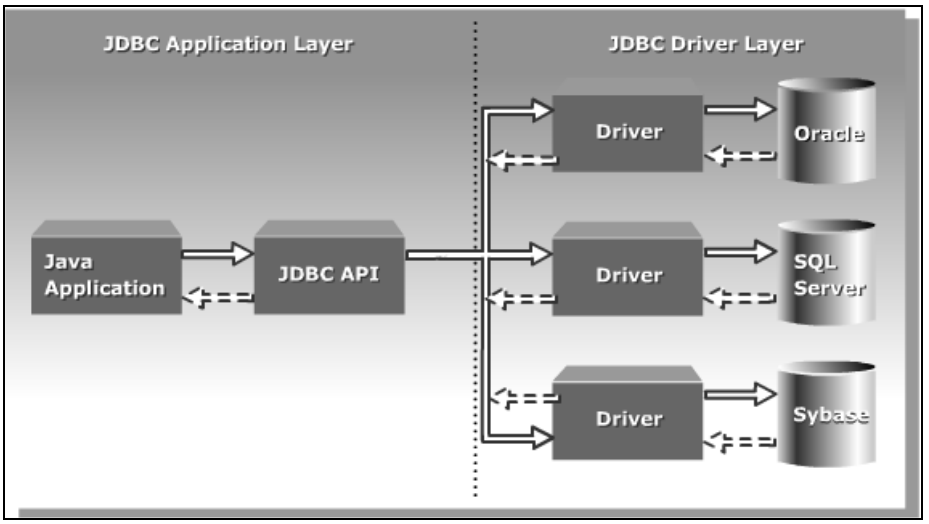
JDBC Architecture

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, you need a mechanism to translate Java statements into SQL statements. The JDBC architecture provides the mechanism for this kind of translation.

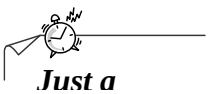
The JDBC architecture has the following two layers:

- **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

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

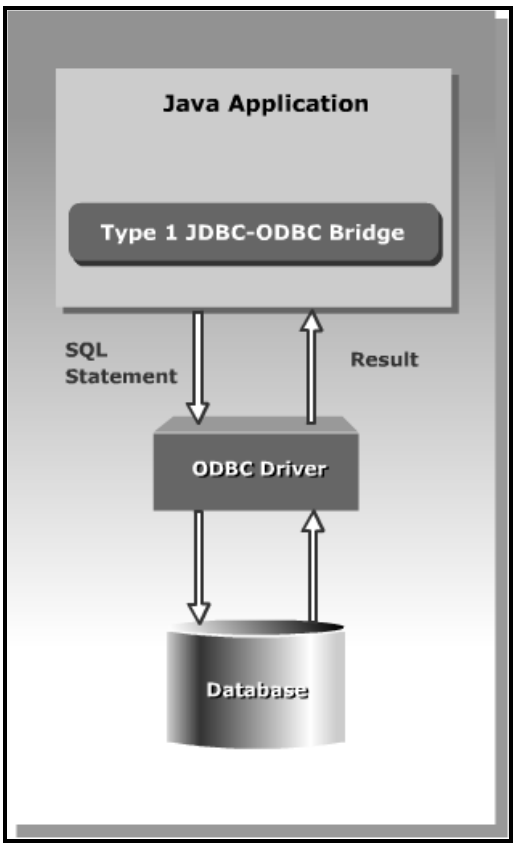
While developing JDBC applications, you need to use JDBC drivers to convert queries into a form that a particular database can interpret. The JDBC driver also retrieves the result of SQL statements and converts the result into equivalent JDBC API class objects that the Java application uses. Because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

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.

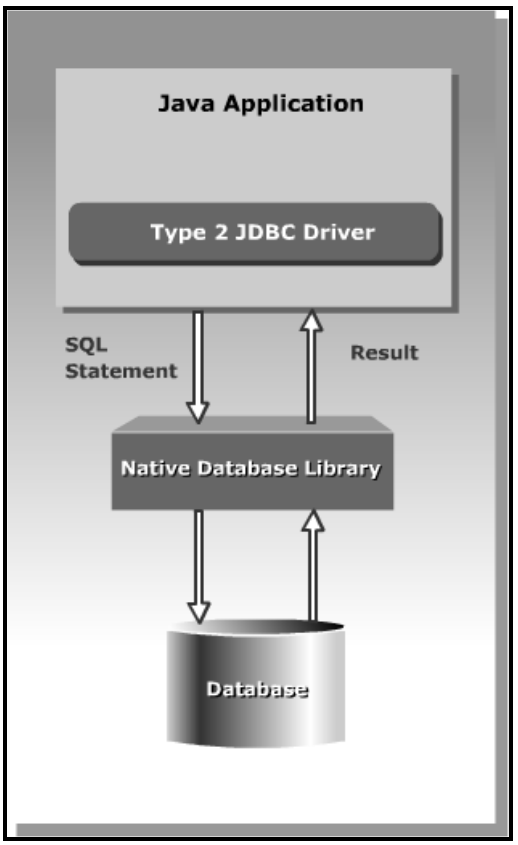


The JDBC-ODBC Bridge Driver

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.

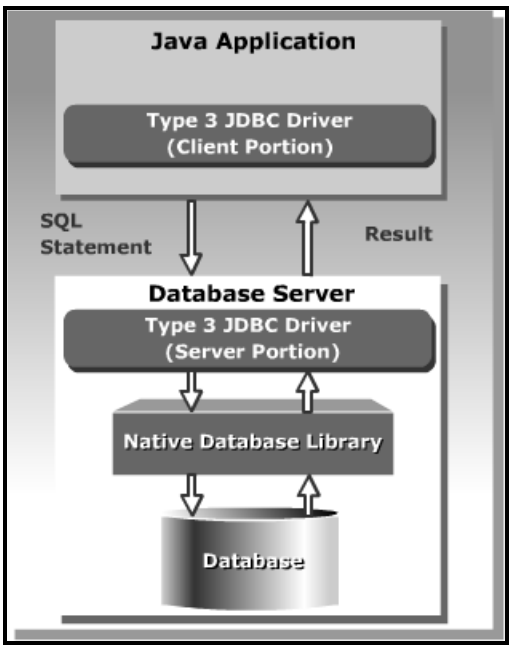


The Native-API Driver

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.

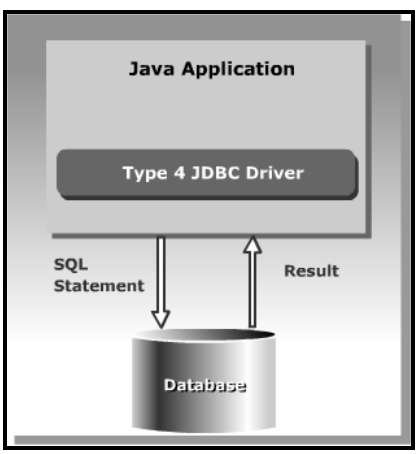
The following figure shows how the Network Protocol driver works.



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:

- Load a driver.
- Connect to a database.
- Create and execute JDBC statements.
- 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

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=Li
brary;user=user1;password=password#1234");
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.

Creating a Table

You can use the `CREATE TABLE` statement to create and define the structure of a table in a database. You can use the following code snippet in a Java application to create a table:

```
String str="CREATE TABLE Publishers"
+"(pub_id VARCHAR(5),"
+"pub_name VARCHAR(50),"
+"phone INTEGER,"
+"address VARCHAR(50), "
+"city VARCHAR(50), "
+"ZIP VARCHAR(20))";
Statement stmt=con.createStatement();
stmt.execute(str);
```

In the preceding code snippet, `str` contains the `CREATE TABLE` statement to create the `Publishers` table. The `execute()` method is used to process the `CREATE TABLE` statement.

Altering and Dropping a Table

You can use the `ALTER` statement to modify the definition of the database object. You use the `ALTER TABLE` statement to modify the structure of a table. For example, you can use this statement to add a new column in a table, change the data type and width of an existing column, and add a constraint to a column. You can use the following code snippet to use the `ALTER` statement in a Java application to add a column in the `Books` table:

```
String str="ALTER TABLE Books ADD price INTEGER";
Statement stmt=con.createStatement();
stmt.execute(str);
```

You can use the `DROP` statement to drop an object from a database. You use the `DROP TABLE` statement to drop a table from a database. You can use the following code snippet to drop the `MyProduct` table using a Java application:

```
String str="DROP TABLE MyProduct";
Statement stmt=con.createStatement();
stmt.execute(str);
```

While creating JDBC applications, you need to handle exceptions. The `execute()` method can throw `SQLException` while executing SQL statements.

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.

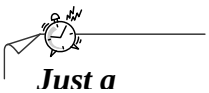
The `SQLException` class contains various methods that provide error information. Some of the methods in the `SQLException` class are:

- `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.



Just a

Identify the interface of the `java.sql` package that must be implemented by all the JDBC driver classes.

Answer:

`Driver`

Activity 9.1: Creating a JDBC Application to Query a Database

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 you to update the `ResultSet` object.*
CONCUR_UPDATABLE	*Specifies the concurrency mode that allows you to update the `ResultSet` object.*

ResultSet Field	*Description*
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 is committed.*

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.
- `Statement createStatement(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.*

Method	*Description*
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:

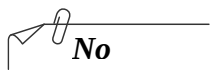
```
if(rs.first()==true)
System.out.println(rs.getString(1) + ", " + rs.getString(2)+ ", " +
rs.getString(3));
```

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);
```



No

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.*

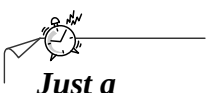
Method	*Description*
`void updateInt(String columnLabel, int x)`	*Updates the specified column with the given `int` value. It accepts the column name whose value needs to be changed.*

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

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`*

In addition to `ResultSet`, Java provides `RowSet`. The `RowSet` interface provides the connected and disconnected environments. In a connected environment the `RowSet` object uses a JDBC driver to make a connection to the underlying database and maintains the connection throughout its life time. On the other hand, in a disconnected environment, the `RowSet` object makes a connection to the underlying database only to read data from the `ResultSet` object or write back to the database. After reading or writing data, the `RowSet` object disconnects the connection.

To establish the connected or disconnected environment, the JDBC API provides the following interfaces:

- `JdbcRowSet`: Provides the connected environment that is most similar to the `ResultSet` object. It makes the `ResultSet` object scrollable and updateable.
- `CachedRowSet`: Provides the disconnected environment. It has all the capabilities of the `JdbcRowSet` object. In addition, it has the following additional features:
  - It obtains a connection to a data source and executes the query.
  - It reads the data from the `ResultSet` object and fills the data in the `CachedRowSet` object.
  - It changes the data while it is disconnected.
  - It reconnects to the data source to write the changes back to it.
  - It checks and resolves the conflicts with the data source.
- `WebRowSet`: Provides the disconnected environment. It has all the capabilities of the `CachedRowSet` object. In addition, it can convert its objects into an Extensible Markup Language (XML) document. Further, it can also use the XML document to populate its object.
- `JoinRowSet`: Provides the disconnected environment. It has all the capabilities of the `WebRowSet` object. In addition, it allows you to create `SQL JOIN` between the `RowSet` objects.
- `FilteredRowSet`: Provides the disconnected environment. It has all the capabilities of the `JoinRowSet` object. In addition, it applies filter to make the selected data available.

The preceding interfaces can be used for specific need, such as connected or disconnected. However, there is another way to provide the `RowSet` implementation. For this, the `RowSetProvider` class provides APIs to get a `RowSetFactory` implementation that can be used to instantiate a proper `RowSet` implementation. It provides the following two methods:

- `static RowSetFactory newFactory()`: Is used to create an instance of the `RowSetFactory` implementation.
- `static RowSetFactory newFactory(String factoryClassName, ClassLoader c1)`: Is used to create an instance of the `RowSetFactory` from the specified factory class name.

The `RowSetFactory` interface defines the implementation of a factory that is used to obtain different types of `RowSet` implementations. It provides the following five methods:

- `CachedRowSet createCachedRowSet()`: Is used to create an instance of `CachedRowSet`.
- `FilteredRowSet createFilteredRowSet()`: Is used to create an instance of `FilteredRowSet`.
- `JdbcRowSet createJdbcRowSet()`: Is used to create an instance of `JdbcRowSet`.
- `JoinRowSet createJoinRowSet()`: Is used to create an instance of `JoinRowSet`.
- `WebRowSet createWebRowSet()`: Is used to create an instance of `WebRowSet`.

Practice Questions

1. Which one of the following methods is used to move the cursor to a particular row in a result set?
   a. `absolute()`
   b. `first()`
   c. `isFirst()`
   d. `relative()`

2. Which one of the following components of the JDBC URL indicates the vendor of RDBMS?
   a. Subname
   b. Supername
   c. Subprotocol name
   d. Superprotocol name

3. Identify the field of the `ResultSet` interface that specifies the cursor to be scrollable and does not reflect the changes in the data.
   a. `TYPE_SCROLL_INSENSITIVE`
   b. `TYPE_FORWARD_ONLY`
   c. `TYPE_SCROLL_SENSITIVE`
   d. `TYPE_READ_ONLY`

4. The `Connection` object provides the _____ method to create a `Statement` object.

5. Which one of the following Java drivers interacts with the database directly using a vendor-specific network protocol?
   a. JDBC-ODBC Bridge driver
   b. Native-API driver
   c. Network Protocol driver
   d. Native Protocol driver

Summary

In this chapter, you learned that:

- The JDBC architecture has the following two layers:
  - JDBC application layer
  - JDBC driver layer
- JDBC supports the following four types of drivers:
  - JDBC-ODBC Bridge driver
  - Native-API driver
  - Network Protocol driver
  - Native Protocol driver
- The classes and interfaces of the JDBC API are defined in the `java.sql` and `javax.sql` packages.
- You can load a driver and register it with the driver manager by using the `forName()` method or the `registerDriver()` method.
- A `Connection` object establishes a connection between a Java application and a database.
- A `Statement` object sends a request and retrieves results to/ from a database.
- You can insert, update, and delete data from a table using the DML statements in Java applications.
- You can create, alter, and drop tables from a database using the DDL statements in Java applications.
- Once the SQL statements are executed, a `ResultSet` object stores the result retrieved from a database.
- You can create various types of the `ResultSet` objects, such as read only, updatable, and forward only.