Unit -4 Database Connectivity

A database is an organized collection of data. There are many different strategies for organizing data to facilitate easy access and manipulation. A database management system(DBMS) provides mechanisms for storing, organizing, retrieving and modifying data form any users. Database management systems allow for the access and storage of data without concern for the internal representation of data.

Today's most popular database systems are relational databases.A language called SQL—pronounced "sequel," or as its individual letters—is the international standard language used almost universally with relational databases to perform queries (i.e., to request information that satisfies given criteria) and to manipulate data.

Some popular relational database management systems (RDBMSs) are Microsoft SQL Server, Oracle, Sybase, IBM DB2, Informix, PostgreSQL and MySQL. The JDK now comes with a pure-Java RDBMS called Java DB—Oracles's version of Apache Derby.

Java programs communicate with databases and manipulate their data using the Java Database Connectivity (JDBC) API. A JDBC driver enables Java applications to connect to a database in a particular DBMS and allows you to manipulate that database using the JDBC API.

JDBC Introduction

The JDBC API is a Java API that can access any kind of tabular data, especially data stored in a Relational Database.
JDBC helps to write Java applications that manage these three programming activities:

- Connect to a data source, like a database
- Send queries and update statements to the database
- Retrieve and process the results received from the database in answer to your query

JDBC includes four components:

- The JDBC API
  The JDBC API provides programmatic access to relational data from the Java programming language. Using the JDBC API, applications can execute SQL statements, retrieve results, and propagate changes back to an underlying data source. The JDBC API can also interact with multiple data sources in a distributed, heterogeneous environment.The JDBC API is part of the Java platform, which includes the Java Standard Edition (Java SE) and the Java Enterprise Edition (Java EE). The JDBC 4.0 API is divided into two packages: java.sql and javax.sql. Both packages are included in the Java SE and Java EE platforms.
- JDBC Driver Manager
  The JDBC DriverManager class defines objects which can connect Java applications to a JDBC driver. DriverManager has traditionally been the backbone of the JDBC architecture. It is quite small and simple.

- JDBC Test Suite

The JDBC driver test suite helps you to determine that JDBC drivers will run your program. These tests are not comprehensive or exhaustive, but they do exercise many of the important features in the JDBC API.

- JDBC-ODBC Bridge
  The Java Software bridge provides JDBC access via ODBC drivers. Note that you need to load ODBC binary code onto each client machine that uses this driver. As a result, the ODBC driver is most appropriate on a corporate network where client installations are not a major problem, or for application server code written in Java in a three-tier architecture.

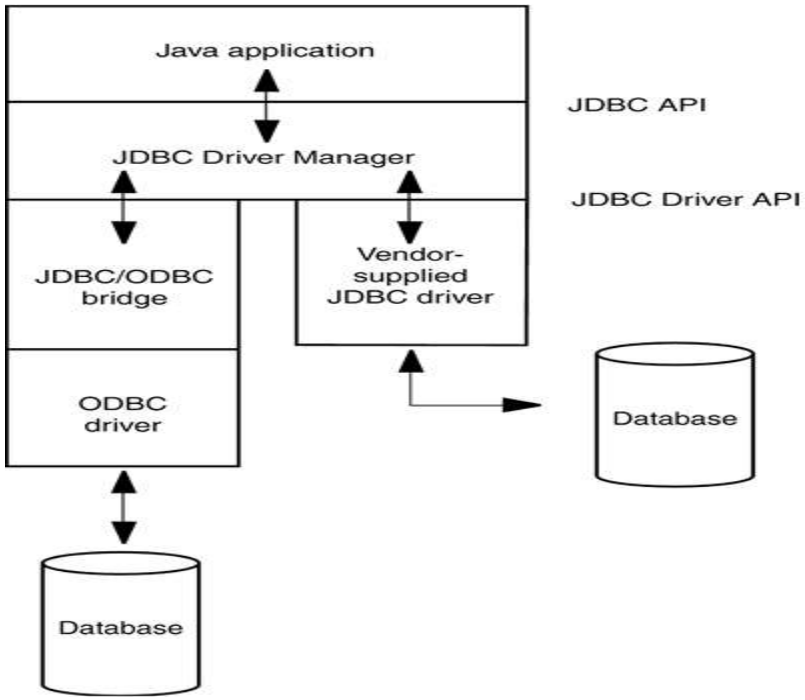

fig.JDBC-to-database communication path

JDBC Driver Types

There are 4 types of JDBC drivers:

1. **JDBC-ODBC bridge driver (Type 1 Driver)**
   - ➔ The JDBC-ODBC bridge driver uses ODBC driver to connect to the database.
   - ➔ The JDBC-ODBC bridge driver converts JDBC method calls into the ODBC function calls.
   - ➔ This is now discouraged because of thin driver.

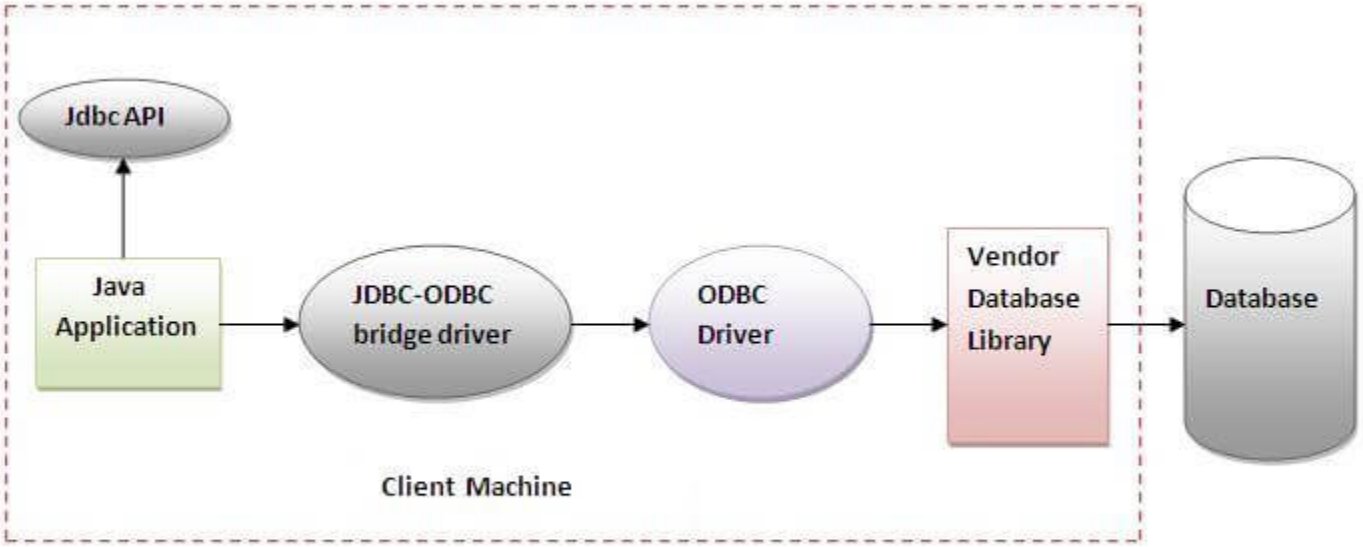

Figure- JDBC-ODBC Bridge Driver

Advantages

- ➔ easy to use.
- ➔ can be easily connected to any database.

Disadvantages

- ➔ Performance degraded because JDBC method call is converted into the ODBC function calls.
- ➔ The ODBC driver needs to be installed on the client machine.

2. **Native-API driver (partially java driver/ Type 2 Driver)**
   - ➔ The Native API driver uses the client-side libraries of the database.
   - ➔ The driver converts JDBC method calls into native calls of the database API.
   - ➔ It is not written entirely in java.

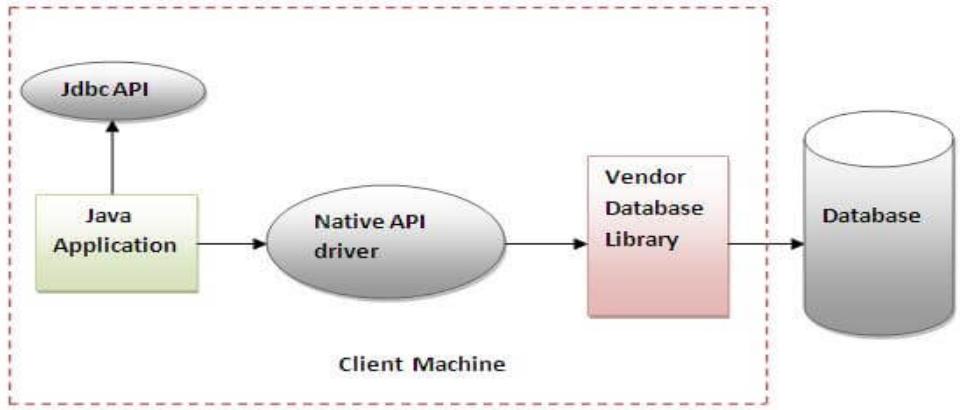

Figure- Native API Driver

Advantage

➔ performance upgraded than JDBC-ODBC bridge driver.

Disadvantage

➔ The Native driver needs to be installed on the each client machine.

➔ The Vendor client library needs to be installed on client machine.

3. **Network Protocol driver (fully java driver/ Type 3 Driver)**

➔ The Network Protocol driver uses middleware (application server) that converts JDBC calls directly or indirectly into the vendor-specific database protocol.

➔ It is fully written in java.

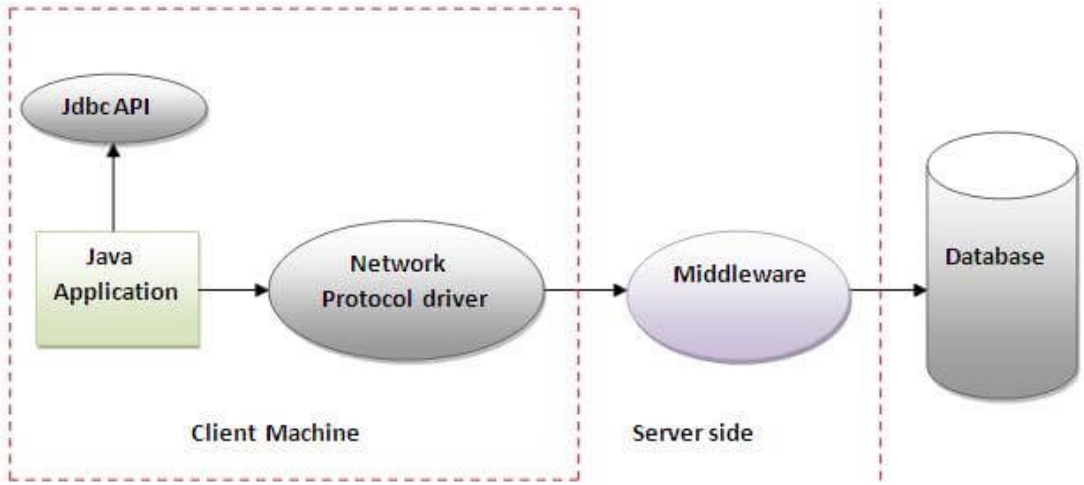

Figure- Network Protocol Driver

Advantage

➔ No client side library is required because of application server that can perform many tasks like auditing, load balancing, logging etc.

Disadvantages

➔ Network support is required on client machine.

➔ Requires database-specific coding to be done in the middle tier.

➔ Maintenance of Network Protocol driver becomes costly because it requires database-specific coding to be done in the middle tier.

4. **Thin driver (fully java driver/ Type 4 Driver)**

➔ The thin driver converts JDBC calls directly into the vendor-specific database protocol.

➔ That is why it is known as thin driver.

➔ It is fully written in Java language.

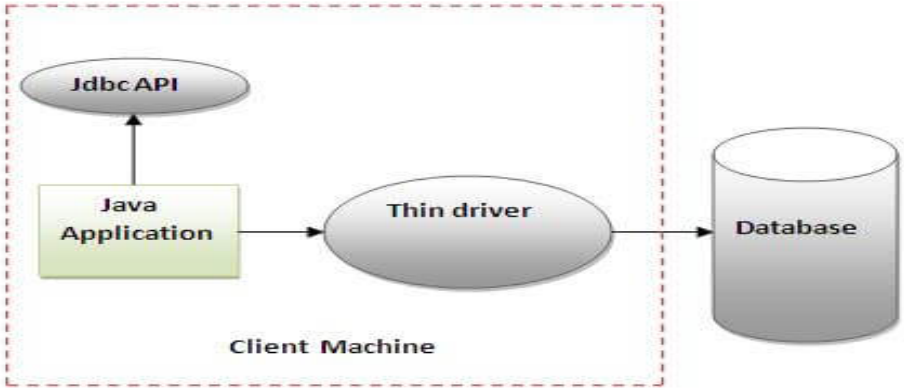

Figure- Thin Driver

Advantage

➔ Better performance than all other drivers.

➔ No software is required at client side or server side.

Disadvantage

➔ Drivers depends on the Database.

Instructions for Setting Up a MySQL User Account

Step 1: Open MySQL Command-Line Client

Open the Command Prompt or PowerShell on your Windows machine.

Navigate to the MySQL bin directory. The default path is usually C:\Program Files\MySQL\MySQL Server X.X\bin, where X.X is the version number.

Type the following command and press Enter to log in to MySQL:

mysql -u root -p

You will be prompted to enter the root password. Enter the password and press Enter.

Step 2: Create a New MySQL User

Once you are logged in, you can create a new user with the following SQL command:

CREATE USER 'username'@'localhost' IDENTIFIED BY 'password';
Replace username with the desired username and password with the desired password.

Step 3: Grant Privileges to the User
After creating the user, you need to grant the necessary privileges. For example, to grant all privileges on a specific database:
GRANT ALL PRIVILEGES ON database_name.* TO 'username'@'localhost';

Replace database_name with the name of the database to which you want to grant access.

Step 4: Run the following command to apply the changes:
FLUSH PRIVILEGES;

Step 5: Test the New User logging in with the specified credentials:
mysql -u username -p
Enter the password when prompted.

Connecting to the Database

- An object that implements interface Connection manages the connection between the Java program and the database.

- Connection objects enable programs to create SQL statements that manipulate databases. The program initializes connection with the result of a call to static method getConnection of class DriverManager (package java.sql), which attempts to connect to the database specified by its URL.
- Method get-Connection takes three arguments—a String that specifies the database URL, a String that specifies the username and a String that specifies the password.
- The URL locates the database (possibly on a network or in the local file system of the computer).
- The URL **jdbc:mysql://localhost/books** specifies the protocol for communication (jdbc), the subprotocol for communication (mysql) and the location of the database (//localhost/books, where localhost is the host running the MySQL server and books is the database name).
- The subprotocol mysql indicates that the program uses a MySQL-specific subprotocol to connect to the MySQL database.
- If the DriverManager cannot connect to the database, method getConnection throws a SQLException (package java.sql).

Figure below lists the JDBC driver names and database URL formats of several popular RDBMSs.

RDBMS	Database URL format
MySQL	`jdbc:mysql://`*hostname*`:`*portNumber*`/`*databaseName*
ORACLE	`jdbc:oracle:thin:@`*hostname*`:`*portNumber*`:`*databaseName*
DB2	`jdbc:db2:`*hostname*`:`*portNumber*`/`*databaseName*
PostgreSQL	`jdbc:postgresql://`*hostname*`:`*portNumber*`/`*databaseName*
Java DB/Apache Derby	`jdbc:derby:`*dataBaseName* (embedded) `jdbc:derby://`*hostname*`:`*portNumber*`/`*databaseName* (network)
Microsoft SQL Server	`jdbc:sqlserver://`*hostname*`:`*portNumber*`;databaseName=`*dataBaseName*
Sybase	`jdbc:sybase:Tds:`*hostname*`:`*portNumber*`/`*databaseName*

```java
import java.sql.*;

class ConnectionDemo{
    public static void main(String[] args) throws SQLException {
        String url = "jdbc:mysql://localhost:3306/test";
        String uname = "admin";
        String pass = "admin";
        Connection conn = DriverManager.getConnection(url, uname, pass);

        if(conn != null){
            System.out.println("Successfully connected");
        }

    }
}
```

Fig : example program showing JDBC connection with mysql

Creating a Statement for Executing Queries
Connection method createStatement is invoked to obtain an object that implements interface Statement (package java.sql). The program uses the Statement object to submit SQL statements to the Database.

Executing a Query
The Statement object's executeQuery method is used to submit a query that selects all the author information from table Authors. This method returns an object that implements interface ResultSet and contains the query results. The ResultSet methods enable the program to manipulate the query result.

Processing a Query's ResultSet
The metadata describes the ResultSet's contents. Programs can use metadata programmatically to obtain information about the ResultSet's column names and types. ResultSetMetaData method getColumnCount is used to retrieve the number of columns in the ResultSet.

Retrieving and Modifying Values from Result Sets
A ResultSet object is a table of data representing a database result set, which is usually generated by executing a statement that queries the database. A ResultSet object can be created through any object that implements the Statement interface, including PreparedStatement, CallableStatement, and RowSet.

You access the data in a ResultSet object through a cursor. Note that this cursor is not a database cursor. This cursor is a pointer that points to one row of data in the ResultSet. Initially, the cursor is positioned before the first row. The method ResultSet.next moves the cursor to the next row. This method returns false if the cursor is positioned after the last row. This method repeatedly calls the ResultSet.next method with a while loop to iterate through all the data in the ResultSet.

ResultSet Interface

In Java, the ResultSet interface is a part of the JDBC (Java Database Connectivity) API. JDBC is a Java-based API that provides a standard interface for connecting to relational databases and executing SQL queries. The ResultSet interface specifically represents the result set of a database query, which is essentially a table of data representing the result of a database query, typically generated by executing a SQL statement.

Here are some key points about the ResultSet interface:

1. Obtaining a ResultSet:
   You obtain a ResultSet object by executing a SQL query through a Statement object.
   For example:
   ```
   Statement statement = connection.createStatement();
   ResultSet resultSet = statement.executeQuery("SELECT * FROM my_table");
   ```
2. Navigating through the resultset
   The ResultSet interface provides methods for navigating through the result set. You can move the cursor forward, backward, or directly to a specific row. Common methods include next(), previous(), first(), last(), etc.
3. Retrieving Data
   You can retrieve data from the ResultSet using methods like getString(), getInt(), getDouble(), etc., based on the data types of the columns in the result set. The column index or column name is used to specify which column's data you want to retrieve.
   ```
   while (resultSet.next()) {
       String name = resultSet.getString("column_name");
       int age = resultSet.getInt("column_age");
       // Process data...
   ```

```
}
```

4. Updating data
   In addition to retrieving data, the ResultSet interface allows you to update data in the result set and, consequently, in the underlying database. Use methods like updateString(), updateInt(), insertRow(), etc.
   ```
   resultSet.updateString("column_name", "new_value");
   resultSet.updateRow();
   ```

5. Closing the resultset
   ```
   resultSet.close();
   ```

ResultSet Types

1. TYPE_FORWARD_ONLY
   This is the default type. In a forward-only result set, you can only iterate through the result set in the forward direction (from the first row to the last row). You cannot move the cursor backward. It is the most lightweight and efficient type but has limitations in terms of navigation.
   ```
   Statement statement = connection.createStatement(ResultSet.TYPE_FORWARD_ONLY,
   ResultSet.CONCUR_READ_ONLY);
   ```

2. TYPE_SCROLL_INSENSITIVE
   In a scroll-insensitive result set, you can move the cursor both forward and backward. Changes made to the underlying data in the database are not reflected in the result set, and the result set does not reflect

changes made by others while it was open. This type allows for more flexible navigation, but it may not always reflect the latest changes in the database.
Statement statement = connection.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_READ_ONLY);

3. TYPE_SCROLL_SENSITIVE
   Similar to TYPE_SCROLL_INSENSITIVE, a scroll-sensitive result set allows both forward and backward navigation. However, changes made to the underlying data in the database are reflected in the result set. This means that if another transaction modifies the data, the result set is updated accordingly. This type provides the most up-to-date view of the data but may be less efficient.
   Statement statement = connection.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE, ResultSet.CONCUR_READ_ONLY);

Additionally, there are two options for specifying the concurrency mode of the result set:

1. CONCUR_READ_ONLY:
   The ResultSet object cannot be updated using the ResultSet interface.
2. CONCUR_UPDATABLE:
   The ResultSet object can be updated using the ResultSet interface.

The default ResultSet concurrency is CONCUR_READ_ONLY.

ResultSetMetaData

The ResultSetMetaData interface in Java is part of the JDBC (Java Database Connectivity) API. It provides information about the structure of the result set produced by a query. This metadata includes details about the number of columns, the names of columns, the types of columns, and other properties.

```
// Establish a connection
Connection connection = DriverManager.getConnection(url, user, password);

// Create a Statement
Statement statement = connection.createStatement();

// Execute a query and get the ResultSet
ResultSet resultSet = statement.executeQuery("SELECT * FROM your_table");

// Get ResultSetMetaData
ResultSetMetaData metaData = resultSet.getMetaData();

// Retrieve metadata information
int columnCount = metaData.getColumnCount();
System.out.println("Number of columns: " + columnCount);

// Iterate through columns and print information
for (int i = 1; i <= columnCount; i++) {
    String columnName = metaData.getColumnName(i);
    String columnType = metaData.getColumnTypeName(i);
```

```
        int columnSize = metaData.getColumnDisplaySize(i);

        System.out.println("Column " + i + ":");
        System.out.println("   Name: " + columnName);
        System.out.println("   Type: " + columnType);
        System.out.println("   Size: " + columnSize);
        System.out.println();
    }
```

Cursors

In the context of databases and JDBC (Java Database Connectivity), a cursor is a mechanism that allows you to traverse and manipulate the result set returned by a database query. Cursors are used to navigate through the rows of a result set, and they provide a way to access and process data sequentially. There are different types of cursors with varying capabilities. Here, I'll provide an overview of the two main types: forward-only and scrollable cursors.

1. Forward-only cursors
   A forward-only cursor is the simplest type of cursor. It allows you to move only in a forward direction through the result set, from the first row to the last row. You cannot move backward or jump to a specific row. This type of cursor is often the most efficient and is suitable for scenarios where you only need to process each row once.

   ```
   Statement statement = connection.createStatement(ResultSet.TYPE_FORWARD_ONLY,
   ResultSet.CONCUR_READ_ONLY);
   ```

```
ResultSet resultSet = statement.executeQuery("SELECT * FROM my_table");


while (resultSet.next()) {
    // Process data
}
```

2. Scrollable cursors

   Scrollable cursors provide more flexibility in terms of navigation. They allow you to move both forward and backward through the result set and jump to a specific row. There are two main types of scrollable cursors: scroll-insensitive and scroll-sensitive.

   - Scroll-Insensitive Cursor:
     Changes made by other transactions to the underlying data are not reflected in the result set, and the result set does not reflect changes made by itself. It provides a snapshot of the data at the time the query was executed.
     ```
     Statement statement = connection.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_READ_ONLY);
     ResultSet resultSet = statement.executeQuery("SELECT * FROM my_table");
     ```
   - Scroll-Sensitive Cursor:
     Changes made by other transactions to the underlying data are reflected in the result set. If another transaction modifies the data, the result set is updated accordingly. This provides a more up-to-date view of the data.
     ```
     Statement statement = connection.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE, ResultSet.CONCUR_READ_ONLY);
     ResultSet resultSet = statement.executeQuery("SELECT * FROM my_table");
     ```

With scrollable cursors, you can use methods like first(), last(), absolute(int row), relative(int rows), etc., to move around the result set.

Cursors are important for efficient processing of large result sets and for scenarios where you need to navigate through the data in a non-sequential manner.

Updating Rows in ResultSet Objects

In JDBC, you can update rows in a ResultSet object using the updateAAA() methods, where AAA corresponds to the data type of the column you want to update. After making changes to the row, you need to call the updateRow() method to apply those changes to the actual database. Here's an example:

```
// Establish a connection
Connection connection = DriverManager.getConnection(url, user, password);

// Create a Statement with a scroll-sensitive, updatable ResultSet
Statement statement = connection.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
ResultSet.CONCUR_UPDATABLE);

// Execute a query and get the ResultSet
ResultSet resultSet = statement.executeQuery("SELECT * FROM your_table");
```

```
// Iterate through the result set
while (resultSet.next()) {
    // Retrieve and update column values
    int id = resultSet.getInt("id");
    String name = resultSet.getString("name");
    double salary = resultSet.getDouble("salary");

    // Modify the data as needed
    name = name.toUpperCase(); // For example, converting name to uppercase

    // Update the row in the ResultSet
    resultSet.updateString("name", name);
    resultSet.updateDouble("salary", salary);

    // Apply the changes to the actual database
    resultSet.updateRow();
}
```

Inserting Rows in ResultSet Objects

In JDBC, a ResultSet object is primarily used for reading data from the database, and it's not typically used for inserting new rows. To insert new rows into a database, you would generally use a PreparedStatement or Statement object. The ResultSet is more suited for retrieving and updating existing data.

Here's an example of how to insert a new row into a database using a PreparedStatement:

```
// Establish a connection
Connection connection = DriverManager.getConnection(url, user, password);

// Define the SQL query with placeholders
String sql = "INSERT INTO your_table (column1, column2, column3) VALUES (?, ?, ?)";

// Create a PreparedStatement
PreparedStatement preparedStatement = connection.prepareStatement(sql);

// Set values for the placeholders
preparedStatement.setString(1, "value1");
preparedStatement.setInt(2, 42);
preparedStatement.setDouble(3, 123.45);

// Execute the query to insert the new row
int rowsAffected = preparedStatement.executeUpdate();

if (rowsAffected > 0) {
    System.out.println("Row inserted successfully.");
} else {
    System.out.println("Failed to insert row.");
}
```

Using Statement Objects for Batch Updates

In JDBC, you can use Statement objects for batch updates to efficiently execute multiple SQL statements in a single batch. Batch updates can be more performant than executing each statement individually, especially when dealing with large datasets. Here's an example of using Statement objects for batch updates:

```
// Establish a connection
Connection connection = DriverManager.getConnection(url, user, password);

// Create a Statement with batch updates enabled
Statement statement = connection.createStatement();

// Define multiple SQL statements for batch updates
String sql1 = "INSERT INTO your_table (column1, column2) VALUES ('value1', 42)";
String sql2 = "UPDATE your_table SET column1 = 'new_value' WHERE column2 = 42";
String sql3 = "DELETE FROM your_table WHERE column2 = 42";

// Add the statements to the batch
statement.addBatch(sql1);
statement.addBatch(sql2);
statement.addBatch(sql3);

// Execute the batch updates
statement.executeBatch();
```

PreparedStatements

A PreparedStatement enables you to create compiled SQL statements that execute more efficiently than Statements. PreparedStatements can also specify parameters, making them more flexible than Statements—you can execute the same query repeatedly with different parameter values.

The PreparedStatement is derived from the more general class, Statement. If you want to execute a Statement object many times, it usually reduces execution time to use a PreparedStatement object Instead.

Performing Parameterized Batch Update using PreparedStatement

It is also possible to have a parameterized batch update, as shown in the following code :

```
connection = DriverManager.getConnection(DATABASE_URL, "root", "" );
connection.setAutoCommit(false);
PreparedStatement pstmt = connection.prepareStatement( "INSERT INTO authors VALUES(?, ?, ?)");
pstmt.setInt(1,19);
pstmt.setString(2, "Ram");
pstmt.setString(3,"Thapa");
pstmt.addBatch();
```

```
pstmt.setInt(1,20);
pstmt.setString(2, "Sita");
pstmt.setString(3,"Devi");
pstmt.addBatch();
// ... and so on for each new

int [] updateCounts = pstmt.executeBatch();
```

The three question marks (?) in the the preceding SQL statement's last line are placeholders for values that will be passed as part of the query to the database. Before executing a PreparedStatement, the program must specify the parameter values by using the Prepared- Statement interface's set methods.For the preceding query, parameters are int and strings that can be set with Prepared- Statement method setInt and setString.
Method setInt's and setString's first argument represents the parameter number being set, and the second argument is that parameter's value.
Parameter numbers are counted from 1, starting with the first question mark (?). Interface PreparedStatement provides set methods for each supported SQL type. It's important to use the set method that is appropriate for the parameter's SQL type in the database—SQLExceptions occur when a program attempts to convert a parameter value to an incorrect type.

Transaction Processing

Transaction processing is a concept in database management that ensures the integrity, consistency, and reliability of data within a database. A transaction is a sequence of one or more operations that are executed as a single unit of work. These operations can include reading, writing, or modifying data. The properties of

transactions are often referred to by the acronym ACID, which stands for Atomicity, Consistency, Isolation, and Durability:

In Java, the JDBC (Java Database Connectivity) API provides mechanisms for managing transactions. Here's a simple example:

```
// Establish a connection
Connection connection = DriverManager.getConnection(url, user, password);

connection.setAutoCommit(false);  // Disable auto-commit to start a transaction

try {
    // Perform multiple SQL operations as part of the transaction
    Statement statement = connection.createStatement();
    statement.executeUpdate("INSERT INTO your_table (column1, column2) VALUES ('value1', 42)");
    statement.executeUpdate("UPDATE your_table SET column1 = 'new_value' WHERE column2 = 42");

    // Commit the transaction
    connection.commit();
} catch (SQLException e) {
    // Rollback the transaction in case of an exception
    connection.rollback();
    e.printStackTrace();
} finally {
    // Enable auto-commit after the transaction is complete
    connection.setAutoCommit(true);
```

```
}
```

- The setAutoCommit(false) method is called on the Connection object to disable auto-commit mode, indicating the start of a transaction.

- A series of SQL operations (insert and update) are performed using a Statement.

- The commit() method is called to commit the transaction if all the operations were successful. If an exception occurs, the rollback() method is called to undo the changes made in the transaction.

- The setAutoCommit(true) method is called after the transaction is complete to enable auto-commit mode.

It's important to properly handle exceptions and ensure that the commit() and rollback() methods are called appropriately based on the success or failure of the transaction.

RowSet Interface

RowSet interface in Java is a part of the Java Database Connectivity (JDBC) API. JDBC is a Java-based API that allows Java applications to interact with relational databases.

The RowSet interface is part of the javax.sql package and is an extension of the ResultSet interface. RowSet provides a higher-level abstraction for working with tabular data retrieved from a database. It represents a set of rows, and it can be disconnected from the database, meaning that you can work with the data in-memory without maintaining a continuous connection to the database server.

Connected and Disconnected RowSets

Connected and Disconnected RowSets are two different approaches for working with database data in Java. They are part of the Java Database Connectivity (JDBC) API and provide different strategies for handling and manipulating result sets retrieved from a database.

Connected RowSet:

- A connected RowSet maintains a live connection to the database throughout its lifecycle.
- It is more closely tied to the database, and changes made to the RowSet directly affect the underlying database.
- JdbcRowSet is an example of a connected RowSet. It is essentially a ResultSet that operates as a JavaBeans component and can be used to traverse and update the underlying database.

Disconnected RowSet:

- A disconnected RowSet retrieves data from the database and then operates independently of the database connection.
- It allows for working with data in-memory, and changes made to the RowSet do not affect the actual database until a connection is re-established.
- CachedRowSet is an example of a disconnected RowSet.

Types of RowSets:

- CachedRowSet: A type of RowSet that can operate in a disconnected mode. It stores data in memory and can be modified locally without affecting the database until a connection is re-established.
- JdbcRowSet: A RowSet implementation that is connected and scrollable. It can be used as a wrapper around a ResultSet.
- WebRowSet: An extension of CachedRowSet that adds support for XML serialization, making it suitable for web-based applications.
- FilteredRowSet
- JoinRowSet

Creating RowSet Objects for any type of RowSet

- In Java version 1.4, if we wanted to create JDBC rowset object, then we had to know the internal implemented class name provided by vendors
- But in Java 1.7, the object creation process was simplified i.e we dont need to worry about the implemented class names
- How to create?

```
RowSetFactory rsf = RowSetProvider.newFactory();
```

By using this rowset factory we can create any type of rowset objects. For example :

```
JdbcRowSet jdbcRowSet = rsf.createJdbcRowSet();
```

```
WebRowSet webRowSet = rsf.createWebRowSet();

CachedRowSet cachedRowSet = rsf.createCachedRowSet();

FilteredRowSet filteredRowSet = rsf.createFilteredRowSet();

JoinRowSet joinRowSet = rsf.createJoinRowSet();
```

- We can use the below code to know the corresponding implementation class names

```
System.out.println(jdbcRowSet.getClass().getName());
```

JdbcRowSet

- JdbcRowSet is exactly same as the ResultSet
- JdbcRowSet is not connected and non serializable as ResultSet
  Not connected = after connection is closed we cannot access the JdbcRowSet
  Non serilizable = cannot be easily converted into byte stream and cannot be shared across the network to other people
- The difference between JdbcRowSet and ResultSet is that JdbcRowSet is by default Scrollable and updatable
- Below is a simple JdbcRowSet program to fetch data from the database

```java
import java.sql.SQLException;
import javax.sql.rowset.*;

public class JdbcRowSetDemo {
    public static void main(String[] args) throws SQLException,ClassNotFoundException{
        RowSetFactory rsf = RowSetProvider.newFactory();

        JdbcRowSet jrs = rsf.createJdbcRowSet();
        jrs.setUrl("jdbc:mysql://localhost:3306/college");
        jrs.setUsername("admin");
        jrs.setPassword("admin");
        jrs.setCommand("select * from students");
        jrs.execute();

        // forward direction
        System.out.println("Forward direction-------------------------------");
        while(jrs.next()){
            System.out.println("Row:"+jrs.getRow()+"⟹ Name: "+jrs.getString(2)+" Age: "+jrs.getInt(3));
        }

        // backward direction
        System.out.println("Backward direction-------------------------------");
        while(jrs.previous()){
            System.out.println("Row:"+jrs.getRow()+"⟹ Name: "+jrs.getString(2)+" Age: "+jrs.getInt(3));
        }

        // Random direction
        System.out.println("Backward direction-------------------------------");
        jrs.first();
        System.out.println("Row:"+jrs.getRow()+"⟹ Name: "+jrs.getString(2)+" Age: "+jrs.getInt(3));
        jrs.last();
        System.out.println("Row:"+jrs.getRow()+"⟹ Name: "+jrs.getString(2)+" Age: "+jrs.getInt(3));
        jrs.absolute(4);
        System.out.println("Row:"+jrs.getRow()+"⟹ Name: "+jrs.getString(2)+" Age: "+jrs.getInt(3));

        jrs.close();
    }
}

```

CachedRowSet

- It is a child interface of RowSet
- It is by default scrollable and updatable
- It is disconnected RowSet i.e we can use RowSet without having database connection
- It is serializable
- The main advantage of CachedRowSet is we can send this RowSet object for multiple people across the network and all those people can access RowSet data without having DB connection.
- If we perform any update operation (like Update,Create,Delete) to the CachedRowSet, then to reflect those changes the connection should be established.
- Once connection is established then only those changes will be reflected in database.
- Below is the example of CachedRowSet

```java
import java.sql.*;

import javax.sql.rowset.*;

public class CachedRowSetDemo {
    public static void main(String[] args) throws SQLException,ClassNotFoundException{
        Connection conn = DriverManager.getConnection("jdbc:mysql://localhost:3306/college", "admin", "admin");
        Statement st = conn.createStatement();
        ResultSet rs = st.executeQuery("select * from students");

        RowSetFactory rsf = RowSetProvider.newFactory();
        CachedRowSet crs = rsf.createCachedRowSet();
        crs.populate(rs);

        conn.close();

        // while(rs.next()){
        //     System.out.println("Name: " + rs.getString(2)+",  Age:"+rs.getInt(3));
        // }

        while(crs.next()){
            System.out.println("Name: " + crs.getString(2)+",  Age:"+crs.getInt(3));
        }


    }
}

```

Some key points on RowSet:

- ➔ Similarities between JdbcRowSet and ResultSet (both are non serializable and connected)
- ➔ Differences between JdbcRowSet and ResultSet (JdbcRowSet is default scrollable and updatable whereas ResultSet is by default forward only and read only)
- ➔ Similarities between JdbcRowSet and CachedRowSet (both are scrollable and updatable)
- ➔ Differenced between JdbcRowSet and CachedRowSet (JdbcRowSet is connected and non serializable whereas CachedRowSet is serializable and not connected)