UNIT4 JDBC PART 2

Structure

4.0 INTRODUCTION

In this unit, we will discuss the concept of the ResultSet interface, which contains tabular data of a database query. The ResultSet objects maintain a cursor and associated navigational methods to manage the data. There are three different ResultSet types that control the cursor movement and allow access to the row data through relative and absolute positions. The ResultSet concurrency provides read-only and updatable-mode, which contains access for the modification of the data. The ResultSet holdability ensure the database update within the same context by either closing the instance on commit or holding up the connection to accommodate changes in the same ResultSet instance.

The JDBC transaction will also be discussed in this unit. The transaction concept is a fundamental concept from the batch processing point of view, where the related queries are executed in a batch, and it is essential to execute either all the queries of the set or none. If even one of the queries fails, it may lead the database to the inconsistence state. To consolidate our understanding of the topic, we will also discuss the transaction isolation and save points within the context of the Transaction. Transaction isolation is a critical concept in transaction management to control and satisfy the ACID properties. Also, the idea of save points or checkpoints has been discussed to ensure that in case of any failure, we may not require complete rollback but merely using the checkpointing system, we can keep our time and efforts.

The concept of JDBC RowSet discussed where the RowSet interface provides a wrapper around a ResultSet object and thus provide all the ResultSet capabilities. In addition to that, it also offers support to the JavaBeans component model. The RowSet interface is worthy for databases where there is no driver support for specific ResultSet properties. At the end of this unit, the idea of Java Data Objects and the advantages over the JDBC API has been explained in brief.

4.1 OBJECTIVES

After going through this unit, you will be able to:

- explain the concept of ResultSet interface,
- understand the basic concepts of Transaction,
- use of commit() and rollback() methods appropriately for any transaction,
- describe the concept of transaction isolation levels for concurrent execution,
- explain the concept of SavePoints,
- write the program using batch processing in JDBC,
- write the program using the RowSet interface to fetch the data in the tabular form, and
- understand the basics of Java Data Objects.

4.2 JDBC RESULTSET INTERFACE

Resultset Interface is used to store and fetched the executed outcome of any SQL statement. The query execution requires the data to be read from the database and subsequently keep the result after processing. The ResultSet acts like the storage; it stores the data or result of the query after the execution of the SQL statements. The java application may use this result after fetching it from the ResultSet.

We can create a ResultSet as

Statement jdbcstmt = jdbcconn.createStatement();

ResultSet jdbcresultset = jdbcstmt.executeQuery ("select * from Emp");

The ResultSet contains the tabular data where each record includes equal columns. The data is accessed through the cursor maintained by the ResultSet object, and initially, the cursor is positioned before the first row, and after calling the next() for the first time, it points at the first record. The cursor position is maintained at the current row by the Resultset object, and to access the subsequent rows, either we use next() method or previous() method.

The data in the ResultSet is accessed using the cursor, and the cursor moves to the subsequent rows in the forward direction is taken care of by next() method, and it moves to the previous rows through previous() method of the ResultSet object. These methods return true when the next record is available and return false when no more rows are in the ResultSet object that implies the cursor is pointing after the last record. Also, the next() and previous() methods need not be manually closed as they get automatically closed after the Statement object is committed.

We have seen in the previous unit that to access the data using ResultSet we can iterate through the ResultSet.

```
// Retrieve data using ResultSet
while(jdbcresultset.next())
{
System.out.println("id : " + jdbcresultset.getInt("id") + " First : " +
jdbcresultset.getString("first")  + " Last : " + jdbcresultset.getString("last") +  " Age : "
+ jdbcresultset.getInt("age") + " City : " + jdbcresultset.getString("city") " Salary : " +
jdbcresultset.getInt("salary"));
}
```

We can access the data of the record column-wise. For that, we require the getter methods depending on the column's data type we are accessing like; for integer datatype, we can call getInt() method or for string data type, we can use getString() method. We can access the data calling the column name or column index.

The ResultSet has three attributes - Type, Concurrency and Holdability, which we generally set while creating Statement or Prepared Statement.

4.2.1 ResultSet Types

The ResultSet offers different Types based on the cursor movement which provide few important characteristics but not all types may be available with all the databases and drivers which means not all methods works with all ResultSet Types. Few of the Navigation methods are first(), last(), next(), previous(), relative(), absolute(), afterLast(), beforeLast() etc. We can check their availability using DatabaseMetaData.supportsResultSetType(int type) method.

The three ResultSet types are TYPE_FORWARD_ONLY, TYPE_SCROLL_SENSITIVE and TYPE_SCROLL_INSENSITIVE. The TYPE_FORWARD_ONLY is the default ResultSet type, it confirms the movement, or it allows the user to iterate only in the forward direction like first row, second row 2, third row and so on.

TYPE_SCROLL_SENSITIVE and TYPE_SCROLL_INSENSITIVE allow the user to iterate in both directions and also permits to access relative positions concerning the current position, or even we can access an absolute position.

But, the TYPE_SCROLL_SENSITIVE is sensitive enough to reflect any change in the record. At the same time, it is already open, i.e. it provides a dynamic view of the underlying data, whereas TYPE_SCROLL_INSENSITIVE would not reflect any change in the database record with the opened Resultset. An insensitive ResultSet only provides a static view of the underlying data, i.e. columns values of rows. Subsequently, we will require a new ResultSet object to view the database changes.

Let us summarize the three ResultSet types: 1) The ResultSet.TYPE_FORWARD_ONLY is neither scrollable and nor sensitive. 2) The ResultSet.TYPE_SCROLL_SENSITIVE is scrollable and sensitive to the database changes, and 3) The ResultSet.TYPE_SCROLL_INSENSITIVE is scrollable but not sensitive to the underlying changes to the database.

4.2.2 ResultSet Concurrency

The ResultSet Concurrency evaluates the ResultSet status through the concurrency mode, which provides the two levels. The first mode, where the ResultSet is updatable, i.e. ResultSet.CONCUR_UPDATABLE() and we can update the columns of each row, or we can insert new rows. The other level is the read-only, i.e. ResultSet.CONCUR_READ_ONLY() which is a default ResultSet concurrency. To incorporate the ResultSet Concurrency, we may require to use the database lock mechanisms.

Again, not all the JDBC drivers support the concurrency modes so that we can check the supported mode through DatabaseMetaData.supportsResultSetConcurrency(int concurrency) method.

The ResultSet Concurrency is independent of scrollability, but generally, we need both ResultSet.TYPE_SCROLL_SENSITIVE and ResultSet Concurrency to satisfy so that we can reach to a particular row to update. There are six ResultSet categories in total, which are as follows:

ResultSet.TYPE_FORWARD_ONLY ()/ ResultSet.CONCUR_READ_ONLY()

ResultSet.TYPE_FORWARD_ONLY() / ResultSet.CONCUR_UPDATABLE()

ResultSet.TYPE_SCROLL_SENSITIVE / ResultSet.CONCUR_READ_ONLY()

ResultSet.TYPE_SCROLL_SENSITIVE / ResultSet.CONCUR_UPDATABLE()

ResultSet.TYPE_SCROLL_INSENSITIVE / ResultSet.CONCUR_READ_ONLY()

ResultSet.TYPE_SCROLL_INSENSITIVE / ResultSet.CONCUR_UPDATABLE()

Let us take an example to understand the ResultSet interface where we fetch the last record, all records, absolute row record and relative row record. In this example, when we try to update the record of the last row, we get the error as due to ResultSet.CONCUR_READ_ONLY(), records are not updatable.

Example 4.2.1

```
import java.sql.*;
import java.sql.DriverManager;
class scrollableRS
{
public static void main(String args[])
{
Connection jdbcconn = null;
try
{
Class.forName("com.mysql.cj.jdbc.Driver");
jdbcconn =
DriverManager.getConnection("jdbc:mysql://localhost:3306/emp", "root",
"bpit");

// Scrollable ResultSet but insensitive to any change made by others
// Read-only Resultset, not updatable
Statement jdbcstmt =
jdbcconn.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
ResultSet.CONCUR_READ_ONLY);
{
ResultSet jdbcresultset = jdbcstmt.executeQuery("select * from
emp10");

jdbcresultset.last();
System.out.println("--- Show the Last Row ---");
System.out.println(jdbcresultset.getRow() + ": " +
+jdbcresultset.getInt("id") + ", "
+ jdbcresultset.getString("first") + ", " +
jdbcresultset.getString("last") + ", "
+ jdbcresultset.getInt("age") + ", " + jdbcresultset.getString("city")
+ ", "
+ jdbcresultset.getInt("salary"));

jdbcresultset.beforeFirst();
System.out.println("--- Show all the Rows ---");
while (jdbcresultset.next())
{
System.out.println(jdbcresultset.getRow() + ": " +
+jdbcresultset.getInt("id") + ", "
+ jdbcresultset.getString("first") + ", " +
jdbcresultset.getString("last") + ", "
+ jdbcresultset.getInt("age") + ", " + jdbcresultset.getString("city")
+ ", "
```

```
+ jdbcresultset.getInt("salary"));
}

jdbcresultset.absolute(5); // starting row number is 1
System.out.println("--- Show Absolute Row 5 ---");
System.out.println(jdbcresultset.getRow() + ": " +
+jdbcresultset.getInt("id") + ", "
                                    +
jdbcresultset.getString("first") + ", " +
jdbcresultset.getString("last") + ", "
+ jdbcresultset.getInt("age") + ", " + jdbcresultset.getString("city")
+ ", "
+ jdbcresultset.getInt("salary"));

jdbcresultset.relative(-2);
System.out.println("--- Show Relative Row -2 ---");
System.out.println(jdbcresultset.getRow() + ": " +
+jdbcresultset.getInt("id") + ", "
+ jdbcresultset.getString("first") + ", " +
jdbcresultset.getString("last") + ", "
+ jdbcresultset.getInt("age") + ", " + jdbcresultset.getString("city")
+ ", "
+ jdbcresultset.getInt("salary"));

// Update a row
jdbcresultset.last();
System.out.println("--- Try to Update a row ---");
System.out.println(jdbcresultset.getRow() + ": " +
+jdbcresultset.getInt("id") + ", "
+ jdbcresultset.getString("first") + ", " +
jdbcresultset.getString("last") + ", "
+ jdbcresultset.getInt("age") + ", " + jdbcresultset.getString("city")
+ ", "
+ jdbcresultset.getInt("salary"));
jdbcresultset.updateInt("salary", 9000); // update cells via column
name
jdbcresultset.updateInt("age", 50);
jdbcresultset.updateRow(); // update the row in the data source
System.out.println(jdbcresultset.getRow() + ": " +
+jdbcresultset.getInt("id") + ", "
+ jdbcresultset.getString("first") + ", " +
jdbcresultset.getString("last") + ", "
+ jdbcresultset.getInt("age") + ", " + jdbcresultset.getString("city")
+ ", "
+ jdbcresultset.getInt("salary"));

}
jdbcconn.close();
}
catch (Exception e)
{
//Handle errors for Class.forName
System.out.println(e);
} // end try
System.out.println("Executed!");
}// end main

} // end Example
```

Output:

```
--- Show the Last Row ---
8: 621, Kushagr, Mahajan, 31, Pune, 6000
--- Show all the Rows ---
1: 501, Aman, Sharma, 40, Delhi, 8000
2: 502, Ajay, Tandon, 40, Mumbai, 9000
3: 505, Suraj, Gupta, 45, Pune, 8000
4: 509, Gaurav, Pant, 49, Delhi, 5000
5: 512, Gaurav, Mehta, 44, Delhi, 5000
6: 521, Aman, Sharma, 33, Delhi, 8000
7: 525, Raghav, Gupta, 32, Pune, 8000
8: 621, Kushagr, Mahajan, 31, Pune, 6000
--- Show Absolute Row 5 ---
5: 512, Gaurav, Mehta, 44, Delhi, 5000
--- Show Relative Row -2 ---
3: 505, Suraj, Gupta, 45, Pune, 8000
--- Try to Update a row ---
8: 621, Kushagr, Mahajan, 31, Pune, 6000
com.mysql.cj.jdbc.exceptions.NotUpdatable: Result Set not
updatable.This result set must come from a statement that was created
with a result set type of ResultSet.CONCUR_UPDATABLE, the query must
select only one table, can not use functions and must select all
primary keys from that table. See the JDBC 2.1 API Specification,
section 5.6 for more details.
Executed!
```

4.2.3 ResultSet Holdability

The ResultSet Holdability evaluates the ResultSet instance if it has a closed status when the underlying connection commit(). The commit() method used to ensure the consistent database states. In the case of ResultSet Holdability, there are two types of holdability. The first one is where all the ResultSet instances are closed on commit(), i.e. CLOSE_CURSORS_OVER_COMMIT and the other one where all the ResultSet instances are not closed but kept open to accommodate the updates with the same ResultSet on commit(), i.e. HOLD_CURSORS_OVER_COMMIT.

The ResultSet Interface methods are as follows:

boolean next(): The next() method is used for advancing the cursor to next row.

boolean previous(): The previous() method precedes the cursor to the previous row.

boolean first(): The first() method moves the cursor to the first row.

boolean last(): The last() method move the cursor to the last row.

boolean absolute(int row): The absolute() method moves the cursor to the specific row.

boolean relative(int row): The relative() method moves the cursor to the relative row number from the current row.

Again, not all the JDBC drivers support the concurrency modes so that we can check the supported mode through a DatabaseMetaData.supportsResultSetHoldability(int holdability) method. The Connection interface getHoldability() method is used to return an integer value 1 or 2. The ResultSet.HOLD_CURSORS_OVER_COMMIT returns the value 1, and the ResultSet.CLOSE_CURSORS_AT_COMMIT returns the value 2.

We can check the Default cursor holdability using DatabaseMetaData.getResultSetHoldability() and also check the driver support for HOLD_CURSORS_OVER_COMMIT and CLOSE_CURSORS_AT_COMMIT using DatabaseMetaData.supportsResultSetHoldability(ResultSet.HOLD_CURSORS_OVER_COMMIT) and DatabaseMetaData.supportsResultSetHoldability(ResultSet.CLOSE_CURSORS_AT_COMMIT) respectively.

Let us take an example where the holdability value is set to HOLD_CURSORS_OVER_COMMIT. In this example, first, we have to set the auto-commit to false and then retrieve the employee table's content to a ResultSet object and insert a new row in the ResutlSet and to the employee table. At the end, we will commit the execution, but due to the holdability property HOLD_CURSORS_OVER_COMMIT, we can check that the ResultSet object is open.

Example 4.2.3:

```
import java.sql.*;
import java.sql.DriverManager;
class holdabilityRS
{
 public static void main(String args[])
 {
  Connection jdbcconn = null;
  try
  {
   Class.forName("com.mysql.cj.jdbc.Driver");
   jdbcconn =
   DriverManager.getConnection("jdbc:mysql://localhost:3306/emp",
   "root", "admin");
   // By default auto commit is true so setting the auto commit false
   jdbcconn.setAutoCommit(false);
   // Holdability is set to HOLD_CURSORS_OVER_COMMIT
   jdbcconn.setHoldability(ResultSet.HOLD_CURSORS_OVER_COMMIT);
   // Creating a Statement object
   Statement jdbcstmt =
   jdbcconn.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
   ResultSet.CONCUR_UPDATABLE);
   // Execute query to fetch the data
   ResultSet jdbcresultset = jdbcstmt.executeQuery("select * from
   emp10");
   System.out.println("Employee table");
   while (jdbcresultset.next())
   {
    System.out.print("ID: " + jdbcresultset.getInt("id") + ", ");
    System.out.print("First_Name: " + jdbcresultset.getString("first")
    + ", ");
    System.out.print("Last_Name: " + jdbcresultset.getString("last"));
    System.out.print("Age: " + jdbcresultset.getInt("age") + ", ");
    System.out.print("City: " + jdbcresultset.getString("city") + ",
    ");
    System.out.print("Salary: " + jdbcresultset.getInt("salary"));
    System.out.println("");
   }
   // Insert new row
   jdbcresultset.moveToInsertRow();
   jdbcresultset.updateInt(1, 622);
   jdbcresultset.updateString(2, "Poonam");
   jdbcresultset.updateString(3, "Sharma");
   jdbcresultset.updateInt(4, 34);
   jdbcresultset.updateString(5, "Delhi");
```

```
   jdbcresultset.updateInt(6, 6000);
   jdbcresultset.insertRow();
   // commit transaction
   jdbcconn.commit();
   boolean status = jdbcresultset.isClosed();
   if (status)
   {
    System.out.println("ResultSet object close");
   }
   else
   {
    System.out.println("ResultSet object open");
   }
  }
  catch (Exception e)
  {
   //Handle errors for Class.forName
   System.out.println(e);
  }
  // end try
  System.out.println("Executed!");
 }
 // end main
}
// end Example
```

Output:

```
Employee table
ID: 501, First_Name: Aman, Last_Name: SharmaAge: 40, City: Delhi,
Salary: 8000
ID: 502, First_Name: Ajay, Last_Name: TandonAge: 40, City: Mumbai,
Salary: 9000
ID: 505, First_Name: Suraj, Last_Name: GuptaAge: 45, City: Pune,
Salary: 8000
ID: 509, First_Name: Gaurav, Last_Name: MehtaAge: 44, City: Delhi,
Salary: 5000
ID: 512, First_Name: Gaurav, Last_Name: MehtaAge: 44, City: Delhi,
Salary: 5000
ID: 521, First_Name: Aman, Last_Name: SharmaAge: 33, City: Delhi,
Salary: 8000
ID: 525, First_Name: Suraj, Last_Name: GuptaAge: 45, City: Pune,
Salary: 8000
ID: 621, First_Name: Kushagr, Last_Name: MahajanAge: 31, City: Pune,
Salary: 6000
ResultSet object open
Executed!
```

4.2.4 ResultSet MetaData

ResultSetMetaData interface provides data about the data available in the ResultSet. It includes all the column and their characteristics. We can get the metadata object through the method from the ResultSet object.

syntax:

public ResultSetMetaData getMetaData()

The ResultSetMetaData interface supports the following methods:

ResultSetMetaData.getColumnCount(): This method used to fetch the number of columns in the ResultSet object.

ResultSetMetaData.getColumnName(int indexno): This method used to fetch the name of the columns from the ResultSet object.

ResultSetMetaData getColumnTypeName (int indexno): This method used to get the column datatype from the ResultSet object.

ResultSetMetaData getColumnDisplaySize (int indexno): This method used to get the column size from the ResultSet object.

Let us take an example to understand the concept of ResultSetMetaData:

Example 4.2.4:

```
import java.sql.*;
import java.sql.DriverManager;
public class resultMeta
{
 public static void main(String[] args)
 {
  Connection jdbcconn = null;
  try
  {
   Class.forName("com.mysql.cj.jdbc.Driver");
   jdbcconn =
   DriverManager.getConnection("jdbc:mysql://localhost:3306/emp",
   "root", "admin");
   Statement jdbcstmt = jdbcconn.createStatement();

   ResultSet jdbcresultset = jdbcstmt.executeQuery("Select * from
   Emp");
   ResultSetMetaData jdbcrsmd = jdbcresultset.getMetaData();

   int countno = jdbcrsmd.getColumnCount();
   System.out.println("Total number of columns are " + countno);
   // Display column name, data type and the column size
   for (int i = 1; i <= countno; i++)
   {
    System.out.println(jdbcrsmd.getColumnName(i) + " " +
    jdbcrsmd.getColumnTypeName(i) + " "
    + jdbcrsmd.getColumnDisplaySize(i));

    System.out.println();
   }
   jdbcconn.close();
  }
  catch (Exception e)
  {
   System.out.println(e);
  }
  System.out.println("Success!");
 }
}
```

Output:

```
Total number of columns are 6
id INT 10

first VARCHAR 255
```

```
last VARCHAR 255

age INT 10

city VARCHAR 255

salary INT 10


Success!
```

☞ Check Your Progress 1

1. What is a ResultSet Object, and how is it used to fetch the data from the database?

2. Which is the default ResultSet type in the JDBC application?

3. What is ResultSet holdability in JDBC?

4. How to iterate the data in both forward and backward direction?

4.3 JDBC TRANSACTIONS

A transaction is a logical unit that may contain one or more SQL query executed in totality to perform a specific task. Generally, any change to the database state is due to a transaction execution like creating, updating or deleting a tuple from the table. It is essential to run the transactions in a controlled environment to ensure system integrity and consistency.

We can understand the transaction concept by taking a simple example of updating the fund transfer from one account to another. If the balance from one account is deducted but couldn't transfer to another, then we would like the first execution to roll back; otherwise, the amount is lost in the cyber-space. This example clarifies that at times we don't like even the first statement should take effect if the second statement didn't execute correctly; else, we will end up with inconsistent data.

The mechanism to ensure that either all dependent queries or batch of queries execute successfully or none is through transactions. In general, a transaction is referred to as a single unit, and the transaction management supports ACID properties – Atomicity, Consistency, Isolation and Durability.

4.3.1 Rollback() and commit()

In JDBC by default, after the query execution result will be automatically saved, i.e. after the connection establishment with the database, the connection remains in an auto-commit mode. The auto-commit mode here leads to each statement treated as a transaction, and all updates due to query execution are made permanent. However, at times, to improve the efficiency, we may require to group two or more statements, and for that, we need to turn-off the auto-commit mode. This turning-off of the auto-commit mode is achieved through the setAutoCommit() method of the connection interface.

syntax:

jdbcconn.setAutoCommit(false);

When we disable the auto-commit mode, no SQL query will commit automatically, and we need to explicitly call the commit() method. This explicit call of jdbcconn.commit() method is through the connection object. Also, if any problem occurs during commit, then a simple set of queries will roll back using jdbcconn.rollback() method.

We can summarize the necessary step needed for any transaction management in JDBC as:

1. We need to change the auto-commit option as false for our connection object.
2. When all the statements in a transaction are completed, we use commit() method to make all the changes permanent.
3. When there is some error while executing statements in a transaction, we need rollback all the changes using rollback() method.

Let us take a simple example to understand the concept of transaction commit and rollback. In this example, we consider the transaction case where the amount from one account is transferred to another account. The MYSQL database is used in this example with the schema "transaction" and table name "account". The two methods used for transferring amounts are debit() to deduct and credit() to add the amount and

combinedly form a transaction. If we get some error while executing any one of these methods, there will be rollback through connection.rollback() else if all goes well it will commit using connection.commit(). In this example, account number 309 is not available, so the query debited the amount from account 1 is also rollback.

Example 4.3.1:

```
import java.sql.*;
import java.sql.DriverManager;
public class transaction
{
public static void main(String[] args)
{
transaction jdbctran = new transaction();
jdbctran.transAmount(1, 309, 10000);
}
// Establishing connection using getConnection() Method
public static Connection getConnection()
{
Connection jdbcconn = null;
try
{
Class.forName("com.mysql.cj.jdbc.Driver");
jdbcconn =
DriverManager.getConnection("jdbc:mysql://localhost:3306/transaction",
"root", "admin");
}
catch (ClassNotFoundException e)
{
e.printStackTrace();
}
catch (SQLException e)
{
e.printStackTrace();
}
return jdbcconn;
}

// Transfer Amount @parameter from_Account, to_Account, value
public void transAmount(int from_Account, int to_Account, int value)
{
Connection jdbcconn = null;
try
{
jdbcconn = getConnection();
// For Transaction execution we make Auto-commit disable
jdbcconn.setAutoCommit(false);
// Start the transaction
debit(jdbcconn, from_Account, value);// deduct the amount from the
account
credit(jdbcconn, to_Account, value); // add the amount to the account
// Complete the transaction
jdbcconn.commit(); // Commit the transaction
System.out.println("Transaction Completed successfully");
}
catch (SQLException e)
{
e.printStackTrace();
if (jdbcconn != null)
{
try
```

```
{
System.out.println("Transaction Rollback due to error");
jdbcconn.rollback();
}
catch (SQLException e1)
{
e1.printStackTrace();
}
}
// if condition
}
finally
{
if (jdbcconn != null)
{
// close the connection
try
{
jdbcconn.close();
}
catch (SQLException e)
{
e.printStackTrace();
}
}
// if condition
}
// finally
}


private void debit(Connection jdbcconn, int accountno, int
debit_amount) throws SQLException
{

String debitSQL = "update account as Tab_Deduct JOIN "
+ " (select accountno, (amount - ?) as balance from account" + " where
accountno = ?) As tab "
+ " ON  Tab_Deduct.accountno = tab.accountno" + " set
Tab_Deduct.amount = tab.balance"
+ " where Tab_Deduct.accountno = ?";

PreparedStatement jdbcprepstmt = null;
try
{
jdbcprepstmt = jdbcconn.prepareStatement(debitSQL);
jdbcprepstmt.setInt(1, debit_amount);
jdbcprepstmt.setInt(2, accountno);
jdbcprepstmt.setInt(3, accountno);
int countno = jdbcprepstmt.executeUpdate();
if (countno == 0)
{
throw new SQLException("Account number not found " + accountno);
}
}
finally
{
if (jdbcprepstmt != null)
{
jdbcprepstmt.close();
}
}
```

```
}
private void credit(Connection jdbcconn, int accountno, int
credit_amount) throws SQLException
{
String creditSQL = "update account as Tab_Credit JOIN "
+ " (select accountno, (amount + ?) as balance from account" + " where
accountno = ?) As tab "
+ " ON  Tab_Credit.accountno = tab.accountno" + " set
Tab_Credit.amount = tab.balance"
+ " where Tab_Credit.accountno = ?";

PreparedStatement jdbcprepstmt = null;
try
{
jdbcprepstmt = jdbcconn.prepareStatement(creditSQL);
jdbcprepstmt.setInt(1, credit_amount);
jdbcprepstmt.setInt(2, accountno);
jdbcprepstmt.setInt(3, accountno);
int countno = jdbcprepstmt.executeUpdate();
if (countno == 0)
{
throw new SQLException("Account number not found " + accountno);
}
}
finally
{
if (jdbcprepstmt != null)
{
jdbcprepstmt.close();
}
}
}
}
```

Output:

```
java.sql.SQLException: Account number not found 309
Transaction Rollback due to error
        at transaction.credit(transaction.java:97)
        at transaction.transAmount(transaction.java:34)
        at transaction.main(transaction.java:8)
```

4.3.2 JDBC Transaction Isolations

The auto-commit disable is essential during the transaction mode as otherwise, we need to hold database locks during the execution of multiple queries and may end up with some sort of deadlock situation. This situation avoided using the concept of transactions, and even transaction management provides some kind of protection for conflicting statements. Generally, the transaction management uses the lock mechanism to deal with the problem of conflicting statements. DBMS uses locks to check transactional access and manage concurrency control by avoiding problems like dirty read etc.

During the concurrent access of the database generally, the dirty read, non-repeatable reads and phantom reads make the database inconsistent. The problem of dirty read occurs when a transaction reads an uncommitted value. The non-repeatable read problem occurs when one Transaction reread the data after an update made by another

transaction. The phantom reads problem happens when records are added or removed by another transaction; then, the transaction is executed on the database. These problems are resolved using a locking mechanism by applying locks and is controlled by the JDBC transaction Isolation level.

There are different transaction levels ranging from TRANSACTION_NONE, which does not support transactions to TRANSACTION_SERIALIZABLE, which strictly supports transactions following access rules and handles problems like dirty read non-repeatable reads or phantom reads.

JDBC provides five different transaction-level through Connection interface.

TRANSACTION_NONE: This level does not support Transaction and is denoted by 0.

TRANSACTION_READ_UNCOMMITTED: - This level support transaction but has no control over dirty read, non-repeatable and phantom reads and is denoted by 1.

TRANSACTION_READ_COMMITTED: - This level support transaction and prevent dirty read but allows non-repeatable and phantom reads. This level is denoted by 2.

TRANSACTION_REPEATABLE_READ: - This level support transaction and allow phantom read but prevent dirty and non-repeatable reads. This level is denoted by 4

TRANSACTION_SERIALIZABLE: - This level support transaction and strictly prevents dirty read, non-repeatable reads or phantom reads. This level is denoted by 8.

Let us take an example to understand the isolation level numbers and set the level for the underlying database.

Example 4.3.2:

```
import java.sql.*;

public class transIsolation
{
public static void main(String args[])
{
Connection jdbcconn = null;
try
{
Class.forName("com.mysql.cj.jdbc.Driver");
jdbcconn =
DriverManager.getConnection("jdbc:mysql://localhost:3306/transaction",
"root", "admin");
System.out.println("Connection established:- " + jdbcconn);
System.out.println("Transactions levels for the Connection interface
\"TRANSACTION_NONE\":- "
+ Connection.TRANSACTION_NONE);
System.out.println("Transactions levels for the Connection interface
\"TRANSACTION_READ_UNCOMMITTED\":- "
+ Connection.TRANSACTION_READ_UNCOMMITTED);
System.out.println("Transactions levels for the Connection interface
\"TRANSACTION_READ_COMMITTED\":- "
+ Connection.TRANSACTION_READ_COMMITTED);
System.out.println("Transactions levels for the Connection interface
\"TRANSACTION_REPEATABLE_READ\":- "
+ Connection.TRANSACTION_REPEATABLE_READ);
System.out.println("Transactions levels for the Connection interface
\"TRANSACTION_SERIALIZABLE\":- "
```

```
+ Connection.TRANSACTION_SERIALIZABLE);
// Setting the transaction isolation level
jdbcconn.setTransactionIsolation(Connection.TRANSACTION_SERIALIZABLE);
int setLevel = jdbcconn.getTransactionIsolation();
System.out.println("Set the Transaction isolation level the current
database is:- " + setLevel);
jdbcconn.close();
}
catch (Exception e)
{
System.out.println(e);
}
System.out.println("Success!");
}
}
```

Output:

```
Connection established:- com.mysql.cj.jdbc.ConnectionImpl@49ec71f8
Transactions levels for the Connection interface "TRANSACTION_NONE":-
0
Transactions levels for the Connection interface
"TRANSACTION_READ_UNCOMMITTED":- 1
Transactions levels for the Connection interface
"TRANSACTION_READ_COMMITTED":- 2
Transactions levels for the Connection interface
"TRANSACTION_REPEATABLE_READ":- 4
Transactions levels for the Connection interface
"TRANSACTION_SERIALIZABLE":- 8
Set the Transaction isolation level the current database is:- 8
Success!
```

Again, not all the JDBC drivers support the transaction isolation levels so that we can check the sustained status through DatabaseMetaData.supportsTransactionIsolationLevel method. If a driver request of setTransactionIsolation is not supporting a specific isolation level, then the driver can substitute a higher and more restrictive transaction isolation level. Also, if the substitution to higher transaction level fails, it throws an SQLException.

4.3.3 JDBC Savepoints

The process of rollback is a saviour at times, but it's not very efficient when a transaction containing a large number of queries to rollback. The Savepoint is one option that sets a Savepoint, i.e. checkpoint in the Transaction and ensures that rollback to only the marked checkpoint and not of the complete Transaction. The connection interface in the current Transaction offers two methods to set Savepoint. The setSavepoint() and setSavepoint (String name) method returns the Savepoint object and creates unnamed Savepoint and given name Savepoint respectively. The rollback() method is overloaded and takes the Savepoint argument. The Savepoint is automatically released on commit, but if we need to release it, we can use the releaseSavepoint() method, which takes the parameter Savepoint.

Let us extend our transaction example to understand the concept of Savepoint. In this example, we preserve the transaction data in another table all_transactions with the first column from_account, second column as to_account, and the transfer value as the third column. The Savepoint is created when the transfer of the amount is completed from

one account to another account. Now, suppose we encounter some error while preserving the transaction details in a new table. In that case, the rollback operation will take place till the checkpoint only and the portion of queries before the Savepoint need not to rollback. This process will avoid the rollback of the complete Transaction.

Example 4.3.3:

```
import java.sql.*;
import java.sql.DriverManager;
import java.sql.Savepoint;
public class transSavepoint
{
public static void main(String[] args)
{
transSavepoint jdbctran = new transSavepoint();
jdbctran.transAmount(4, 3, 10000);
}
// Establishing connection using getConnection() Method
public static Connection getConnection()
{
Connection jdbcconn = null;
try
{
Class.forName("com.mysql.cj.jdbc.Driver");
jdbcconn =
DriverManager.getConnection("jdbc:mysql://localhost:3306/transaction",
"root", "admin");
}
catch (ClassNotFoundException e)
{
e.printStackTrace();
}
catch (SQLException e)
{
e.printStackTrace();
}
return jdbcconn;
}

// Transfer Amount @parameter from_Account, to_Account, value
public void transAmount(int from_Account, int to_Account, int value)
{
Connection jdbcconn = null;
Savepoint jdbcsp = null;
try
{
jdbcconn = getConnection();
// For Transaction execution we make Auto-commit disable
jdbcconn.setAutoCommit(false);
// Start the transaction
debit(jdbcconn, from_Account, value);// deduct the amount from the
account
credit(jdbcconn, to_Account, value); // add the amount to the account
// Complete the transaction
// setting-up save point
jdbcsp = jdbcconn.setSavepoint("DebitCreditDoneSP");
// Preserving transaction
preserveTrans(jdbcconn, from_Account, to_Account, value);
jdbcconn.commit(); // Commit the transaction
}
catch (SQLException e)
```

```
{
e.printStackTrace();
if (jdbcconn != null)
{
try
{
// complete rollback if no Savepoint
if (jdbcsp == null)
{
System.out.println("Transaction Rollback due to error");
jdbcconn.rollback();
}
else
{
System.out.println("Transaction Rollback to Savepoint");
jdbcconn.rollback(jdbcsp);
// commit the transaction till the Savepoint
jdbcconn.commit();
}
}
catch (SQLException e1)
{
e1.printStackTrace();
}
}
// if condition
}
finally
{
if (jdbcconn != null)
{
// closing the connection
try
{
jdbcconn.close();
}
catch (SQLException e)
{
e.printStackTrace();
}
}
// if condition
}
// finally
}

private void debit(Connection jdbcconn, int accountno, int
debit_amount) throws SQLException
{
String debitSQL = "update account as Tab_Deduct JOIN "
+ " (select accountno, (amount - ?) as balance from account" + " where
accountno = ?) As tab "
+ " ON  Tab_Deduct.accountno = tab.accountno" + " set
Tab_Deduct.amount = tab.balance"
+ " where Tab_Deduct.accountno = ?";

PreparedStatement jdbcprepstmt = null;
try
{
jdbcprepstmt = jdbcconn.prepareStatement(debitSQL);
jdbcprepstmt.setInt(1, debit_amount);
jdbcprepstmt.setInt(2, accountno);
```

```
jdbcprepstmt.setInt(3, accountno);
int countno = jdbcprepstmt.executeUpdate();
System.out.println("Amount debited and  Count of rows updated " +
countno);
if (countno == 0)
{
throw new SQLException("Account number not found " + accountno);
}
}
finally
{
if (jdbcprepstmt != null)
{
jdbcprepstmt.close();
}
}
}

private void credit(Connection jdbcconn, int accountno, int
credit_amount) throws SQLException
{
String creditSQL = "update account as Tab_Credit JOIN "
+ " (select accountno, (amount + ?) as balance from account" + " where
accountno = ?) As tab "
+ " ON  Tab_Credit.accountno = tab.accountno" + " set
Tab_Credit.amount = tab.balance"
+ " where Tab_Credit.accountno = ?";

PreparedStatement jdbcprepstmt = null;
try
{
jdbcprepstmt = jdbcconn.prepareStatement(creditSQL);
jdbcprepstmt.setInt(1, credit_amount);
jdbcprepstmt.setInt(2, accountno);
jdbcprepstmt.setInt(3, accountno);
int countno = jdbcprepstmt.executeUpdate();
System.out.println("Amount credited and Count of  rows updated  " +
countno);
if (countno == 0)
{
throw new SQLException("Account number not found " + accountno);
}
}
finally
{
if (jdbcprepstmt != null)
{
jdbcprepstmt.close();
}
}

}

private void preserveTrans(Connection jdbcconn, int from_account, int
to_account, int value) throws SQLException
{
String preserve = "Insert into all_transaction (from_account,
to_account, value) values ( ?, ?, ?)";
PreparedStatement jdbcprepstmt = null;
try
{
jdbcprepstmt = jdbcconn.prepareStatement(preserve);
```

```
jdbcprepstmt.setInt(1, from_account);
jdbcprepstmt.setInt(2, to_account);
jdbcprepstmt.setInt(3, value);
int countno = jdbcprepstmt.executeUpdate();
System.out.println("Count of rows inserted " + countno);
if (countno == 0)
{
throw new SQLException(
"Data insertion Problem - " + " From Account: " + from_account + " To
Account: " + to_account);
}
}
finally
{
if (jdbcprepstmt != null)
{
jdbcprepstmt.close();
}
}
}
}
```

Output:

```
Amount debited and  Count of rows updated 1
Amount credited and Count of  rows updated  1
java.sql.SQLSyntaxErrorException: Table 'emp.all_transaction' doesn't
exist
      at
com.mysql.cj.jdbc.exceptions.SQLError.createSQLException(SQLError.java:
120)
      at
com.mysql.cj.jdbc.exceptions.SQLError.createSQLException(SQLError.java:
97)
      at
com.mysql.cj.jdbc.exceptions.SQLExceptionsMapping.translateException(SQ
LExceptionsMapping.java:122)
      at
com.mysql.cj.jdbc.ClientPreparedStatement.executeInternal(ClientPrepare
dStatement.java:953)
      at
com.mysql.cj.jdbc.ClientPreparedStatement.executeUpdateInternal(ClientP
reparedStatement.java:1092)
      at
com.mysql.cj.jdbc.ClientPreparedStatement.executeUpdateInternal(ClientP
reparedStatement.java:1040)
      at
com.mysql.cj.jdbc.ClientPreparedStatement.executeLargeUpdate(ClientPrep
aredStatement.java:1347)
      at
com.mysql.cj.jdbc.ClientPreparedStatement.executeUpdate(ClientPreparedS
tatement.java:1025)
      at transSavepoint.preserveTrans(transSavepoint.java:130)
      at transSavepoint.transAmount(transSavepoint.java:40)
      at transSavepoint.main(transSavepoint.java:8)
Transaction Rollback to Savepoint
```

☞ Check Your Progress 2

1. What is a Transaction, and how is it maintained in the JDBC applications?

.....

.....

.....

.....

2. What is SavePoint in the JDBC application?

.....

.....

.....

.....

3. How do the dirty read, non-repeatable reads and phantom reads problems make the database inconsistent? What mechanisms we can use to resolve them?

.....

.....

.....

.....

4.4 JDBC BATCH PROCESSING

A JDBC batch contains a group of SQL statements that are processed together with one database call. JDBC batch is handy when a large number of SQL statements are executed concurrently after putting them in a set. The collection of queries executed with one call. This reduces the network communication overhead and facilitates some of the queries to run in parallel and improve the overall efficiency.

Following are the commonly used methods for Batch Processing:

addBatch(): The addBatch() method of statement interface is used to add the queries to the batch.

executeBatch(): The executeBatch() method used to process the batch.

clearBatch(): The clearBatch() method is used to remove the batch, i.e. it will remove all the queries added in the batch, but we can't remove queries selectively.

The JDBC drivers support is not required for batch processing, but not all the database supports the batch processing, we can check the supported databases through DatabaseMetaData.supportsBatchUpdates() method. If this method returns true implies that the database in context supports the batch processing else, it is not.

In batch processing, the database is executing each update separately, which implies there may be a possibility that out of many queries in the set, one of them may fail. Even if one of the queries fails from the collection, it may lead to an inconsistent database state. All successfully executed queries applied to the database but not those which fails. We can solve this unstable database problem that occurs due to the failure of any query by keeping the batch update inside the JDBC transaction. This step will ensure that the Transaction will execute in totality or none will be updated.

We can summarize the basic step needed to use the JDBC Batch Processing:

1. createStatement() method creates a Statement object.
2. setAutoCommit() method used to set the auto-commit false.
3. addBatch() method to add multiple similar queries into a batch.
4. executeBatch() method to execute all the queries within the batch.
5. commit() method used to commit all the changes.

Let us take an example to understand the concept of batch processing.

Example 4.4:

```
import java.sql.*;
import java.sql.DriverManager;
public class batchProcess
{
public static void main(String[] args)
{
Connection jdbcconn = null;
PreparedStatement jdbcprepstmt  = null;
try
{
Class.forName("com.mysql.cj.jdbc.Driver");
System.out.println("Connecting to database...");
jdbcconn =
DriverManager.getConnection("jdbc:mysql://localhost:3306/emp", "root",
"admin");
String insertSQL = "INSERT INTO EMP10"
+ "(id, first, last, age, city, salary) VALUES"
                              + "(?,?,?,?,?,?)";
jdbcprepstmt=jdbcconn.prepareStatement(insertSQL);

jdbcconn.setAutoCommit(false);

jdbcprepstmt=jdbcconn.prepareStatement(insertSQL);

jdbcprepstmt.setInt(1, 101);// 1 specify first parameter
jdbcprepstmt.setString(2, "Dinesh");
jdbcprepstmt.setString(3, "Sharma");
jdbcprepstmt.setInt(4, 40);
jdbcprepstmt.setString(5, "Delhi");
jdbcprepstmt.setInt(6, 8000);
jdbcprepstmt.addBatch();
```

```
jdbcprepstmt.setInt(1, 102);
jdbcprepstmt.setString(2, "Abhimanu");
jdbcprepstmt.setString(3, "Singh");
jdbcprepstmt.setInt(4, 50);
jdbcprepstmt.setString(5, "Mumbai");
jdbcprepstmt.setInt(6, 9000);
jdbcprepstmt.addBatch();

jdbcprepstmt.setInt(1, 105);
jdbcprepstmt.setString(2, "Abhijit");
jdbcprepstmt.setString(3, "Nayak");
jdbcprepstmt.setInt(4, 45);
jdbcprepstmt.setString(5, "Pune");
jdbcprepstmt.setInt(6, 8000);
jdbcprepstmt.addBatch();

jdbcprepstmt.executeBatch();

jdbcconn.commit();
}
catch (SQLException | ClassNotFoundException e)
{
e.printStackTrace();
if(jdbcconn != null)
{
try
{
System.out.println("Transaction Rollback due to error");
jdbcconn.rollback();
}
catch (SQLException e1)
{
e1.printStackTrace();
}
}
// if condition
}
Finally
{
if(jdbcconn != null)
{
//closing the connection
try
{
jdbcconn.close();
System.out.println("Batch executed Successfully!");
}
catch (SQLException e)
{
e.printStackTrace();
}
}
// if condition
}
// finally
}
// main
}
//example
```

Output:

```
Connecting to database...
Batch executed Successfully!
```

4.5 JDBC ROWSET INTERFACE

The JDBC RowSet interface is an extension of the RowSet interface and is a wrapper around a ResultSet object. RowSet objects inherently derive the capabilities of ResultSet. It allows to use the ResultSet as a JavaBeans component and supports JavaBeans event notification mechanism. It provides the data storage in the tabular form with some additional functionality like sending a notification to other registered components when a particular event is triggered. These additional capabilities make it much flexible and easier to use than a ResultSet.

RowSet types

The two types of RowSet objects are connected RowSet object and a disconnected RowSet object. A connected RowSet object establishes a connection with the database using a JDBC driver and sustains it till the application terminates. JDBCRowSet implementation is the only standard implementation that offers a connected RowSet object, and thus it is similar to ResultSet object.

On the contrary, a disconnected RowSet object only establishes a connection to the data source for reading data from the ResultSet or writing back to the data source. After executing the query, it disconnects and closes the connection. The other four implementations CachedRowSet, WebRowSet, JoinRowSet and FilteredRowSet, offers disconnected RowSet object. The advantage of a disconnected RowSet object is that they are lightweight as they need not to maintain a permanent connection with the data source but still have all other functionalities of the connected RowSet object. These require a connection to establish every time to ensure the updates, and so it is a little bit slower in comparison to the connected RowSet object. But still, these are not only lightweight but also serializable and thus very suitable for network transmission.

Implementation Types

RowSet interface offers five implementation classes JDBCRowSet, CachedRowSet, WebRowSet, JoinRowSet and the FilteredRowSet. The RowSet objects are derived from the ResultSet, and it is the only connected RowSet. It offers additional capabilities like Scrollability and Updatability and JavaBeans Component-based development model.

The connectivity between JDBC RowSet and the data source is preserved through its life cycle, which enables JdbcRowSet to take calls that invoke and, in return, call them on the ResultSet object. A ResultSet object can be made scrollable and updatable by the JdbcRowSet object, and this is useful when the driver and database are not offering these properties. The RowSet objects are scrollable and updatable by default and facilitate the ResultSet with these capabilities after populating the RowSet object with the content of the ResultSet, and this implies that we can traverse the records through the ResultSet object.

The RowSet objects are also used as a JavaBeans component and have properties and JavaBeans Notification Mechanism. The RowSet object property have getter and setter methods to fetch and set values like setInt(), getInt() etc. RowSet objects utilize the

JavaBeans event model, in which registered components, i.e. all listeners notified when certain events like cursor movement, insert, update, delete operations or change in row content occur.

RowSets support JavaBeans events like cursorMoved, rowChanged and rowSetChanged. The cursorMoved event initiated due to the next() or previous(), i.e. the rowChanged event-triggered due to a row insert, update, or delete. Whenever the execute method is called to create or change the RowSet, the rowSetChanged event is initiated.

An application component implements a RowSet listener, and when the event occurs, it performs a triggered action. These application components must register the listener objects with a RowSet.addRowSetListener method and implement the standard RowSetListener interface. The RowSet.removeRowSetListener method is used to unregistered the listener.

A CachedRowSet object offers all the necessary capabilities of the JDBC RowSet objects and the disconnected RowSet objects. The other three implementations are the extensions of this interface. The CachedRowSet provides the following in addition to what JDBCRowSet object provides:

1. connection establishment with the data source and query execution
2. reading data from the ResultSet and populating itself with that data
3. manipulating data during disconnected state
4. reconnection and write back to the data source
5. conflict resolution if any

A WebRowSet object provides the capability of reading and writing the XML document in addition to what CachedRowSet object provides:

A JoinRowSet object offers the ability to formulate an equivalent SQL JOIN without connecting to a data source and also provides all the capabilities that WebRowSet object offers.

A FilteredRowSet object provides the filtering capability to enable the visibility of selected data and also offers all the capabilities that WebRowSet object offers.

The event handling is used by calling the instance of **RowSetListener** in the addRowSetListener() method of JdbcRowSet interface. This interface three methods cursorMoved(RowSetEvent event, rowChanged(RowSetEvent event and rowSetChanged(RowSetEvent event).

Let us take an example to understand the concept of JDBC RowSet. In this example, we have fetched the data and performed the cursor movement task; also, we have used the event handling concept, which is not possible using ResultSet.

Example 4.5:

```
import javax.sql.RowSetEvent;
import javax.sql.RowSetListener;
import javax.sql.rowset.JdbcRowSet;
import javax.sql.rowset.RowSetProvider;
public class rowSet
{
public static void main(String[] args)
{
try
```

```
{

Class.forName("com.mysql.cj.jdbc.Driver");
System.out.println("Connecting to database...");
// RowSet Creation and Execution
JdbcRowSet jdbcRS = RowSetProvider.newFactory().createJdbcRowSet();
jdbcRS.setUrl("jdbc:mysql://localhost:3306/emp");
jdbcRS.setUsername("root");
jdbcRS.setPassword("admin");

jdbcRS.setCommand("select * from emp10");
jdbcRS.execute();

// Listener is added
jdbcRS.addRowSetListener(new MyListener());

while (jdbcRS.next())
{
// Event generated for cursor Movement
System.out.println("Id: " + jdbcRS.getInt(1));
System.out.println("first: " + jdbcRS.getString(2));
System.out.println("Salary: " + jdbcRS.getInt(6));
}
}
catch (Exception e)
{
//Handle errors for Class.forName
System.out.println(e);
}
// end try
System.out.println("Executed!");
}
// end main
}
// end Example

// listener implementation
class MyListener implements RowSetListener
{
public void cursorMoved(RowSetEvent event)
{
System.out.println("Cursor is Moved");
}

public void rowChanged(RowSetEvent event)
{
System.out.println("Cursor is Changed");
}
public void rowSetChanged(RowSetEvent event)
{
System.out.println("RowSet is changed");
}
}
```

```
Connecting to database...
Cursor is Moved
Id: 501
first: Aman
Salary: 8000
Cursor is Moved
Id: 502
```

```
first: Ajay
Salary: 9000
Cursor is Moved
Id: 505
first: Suraj
Salary: 8000
Cursor is Moved
Id: 509
first: Gaurav
Salary: 5000
Cursor is Moved
Executed!
```

4.6 INTRODUCTION TO JAVA DATA OBJECT (JDO)

Java Data Objects (JDO) is a specification which enables storage and retrieval of persistence data of Java objects. The complex applications require these persistent data or the information to be available beyond the life cycle of the program for various purposes.

JDBC API and JDO API allow us to access data through the java application. In JDBC, the relational database is used as a data source, but JDO uses any data source, maybe a relational database, object-oriented database or even flat files. JDBC mostly uses structured query language, while JDO uses Java code for the purpose.
In the case of JDBC, we have to write SQL queries and fetch the ResultSet into data objects, but JDO doesn't require execution of queries, copying JDBC ResultSet into Java objects etc. as all this is taken care of by JDO.
There are various other options available to the users to store and retrieve persistent data like JDBC, object databases and entity EJBs, but they have their limitations. JDO adds many useful features which are offered by other persistence mechanisms. The JDO provides many features like ease of creating persistence classes, supporting large datasets, also the object-oriented concepts, consistency, concurrency and query capabilities etc. JDO is flexible in terms of data storage as it may be a relational or object-oriented, or simple flat-file database, and the implementation is wholly hidden from the user.

The following are the advantage of using the JDO API:

a. Portable: The application based on JDO API are portable and can run on different vendors implementations without recompiling or changing any code.
b. Data Source access is transparent: The code to access the data source is independent of the database.
c. Ease of use: The JDO API facilitates the users to emphasis on the domain object model and the JDO implementation for the persistence details.
d. High performance: JDO implementations look for efficient data access for performance optimization.
e. EJB Integration: EJB features are available for the application using the same domain object model throughout the enterprise.

The JDO offers encapsulation, transparency, and portability, and it supports various databases as per the requirements of the application. The JDO API consists of fewer interfaces and a standard for object persistence which makes it simple to implement.

These all features offer greater flexibility to choose deployment environment and fast development.

☞ Check Your Progress 3

1. What is a JDBC RowSet, and how it is different from ResultSet?

.....

.....

.....

.....

2. What is batch processing in JDBC, and how it improves efficiency?

.....

.....

.....

.....

3. What is Java Data Object API, and how it is different from JDBC API?

.....

.....

.....

.....

4.7 SUMMARY

In this unit, we have seen how the JDBC ResultSet interface is used to store the tabular data after the query execution. The data access using cursor subsequently used by the java application. The cursor movement is controlled through various navigational methods. The ResultSet interface also provides concurrency mode to keep the data read-only or in the updatable mode as per the requirements. The ResultSet holdability provide another essential consideration of maintaining the context of data updates through the ResultSet instance.

The batch processing requires the execution of the transaction management, where we have discussed the usage of the commit and rollback methods. The Transaction management for Java applications is needed to satisfy the ACID property and to maintain the consistency we have discussed the various transaction isolation levels. These transaction isolation levels offer multiple methods to check the database inconsistencies and are vital while concurrent access of the database is taking place.

The JDBC RowSet interface provides connected and disconnected RowSet objects. The connected RowSet objects work like the ResultSet object to maintain the connectivity throughout the execution. In contrast, the disconnected one provides the advantage of establishing the connection only while accessing the data read/write and thus saves network usage and also is very lightweight for this reason. We have also discussed various implementation types of the RowSet interface. The Java Data objects are discussed briefly and providing an insight into their usage and advantages over JDBC.

4.8 SOLUTIONS/ANSWERS TO CHECK YOUR PROGRESS

☞ Check Your Progress 1

1. ResultSet is an object which is automatically created when we execute an SQL query and manage the fetched data from the database in the JDBC applications. With this ResultSet object, a cursor is automatically created to read the data from the object. We need to check the availability of the data before accessing it, so next() method is useful in it, which will return true if the next record is available and move the cursor to that record. We can fetch the data using getter methods.

2. The default ResultSet type in JDBC application is Read only and forward only. In JDBC applications, we can use ResultSet combination of Types and concurrency, and the following are the possible cases:

ResultSet.CONCUR_READ_ONLY(), ResultSet.TYPE_FORWARD_ONLY,

ResultSet.CONCUR_READ_ONLY(), ResultSet.TYPE_SCROLL_SENSITIVE

ResultSet.CONCUR_READ_ONLY(), ResultSet.TYPE_SCROLL_INSENSITIVE.

ResultSet.CONCUR_UPDATABLE(), ResultSet.TYPE_FORWARD_ONLY,

ResultSet.CONCUR_UPDATABLE(), ResultSet.TYPE_SCROLL_SENSITIVE

ResultSet.CONCUR_UPDATABLE(), ResultSet.TYPE_SCROLL_INSENSITIVE.

We can use a particular ResutlSet object type based on application requirements, and then we can choose any of the above combinations and pass as a parameter to createStatement() method.

3. The ResultSet holdability evaluates the ResultSet instance if it has a closed status when the underlying connection commit(). There are two types of holdability:

a. CLOSE_CURSORS_OVER_COMMIT: All the ResultSet instances are closed on commit()
b. HOLD_CURSORS_OVER_COMMIT: All the ResultSet instances are not closed but kept open to accommodate the updates with the same ResultSet on commit().

4. The ResultSet object will use ResultSet.TYPE_SCROLL_SENSITIVE and the following two methods to iterate the data in the forward direction and backward direction

Boolean next() and Boolean previous()

The following example explains this concept:

```
import java.sql.*;
import java.sql.DriverManager;
public class forwBack
{
public static void main(String[] args)
{
Connection jdbcconn = null;
Try
{
Class.forName("com.mysql.cj.jdbc.Driver");
jdbcconn = DriverManager.getConnection("jdbc:mysql://localhost:3306/emp", "root",
"admin");
Statement jdbcstmt =
jdbcconn.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,ResultSet.CONCUR_UPD
ATABLE);

ResultSet jdbcresultset = jdbcstmt.executeQuery("Select * from Emp");
System.out.println("data in forward direction");

System.out.println("ENO      ENAME      ESAL      EADDR");

System.out.println("**********************************");
while(jdbcresultset.next())
{
//Retrieve data
System.out.println("id : " + jdbcresultset.getInt("id") + " First : " +
jdbcresultset.getString("first")  + " Last : " + jdbcresultset.getString("last") +  " Age : " +
jdbcresultset.getInt("age") + " City : " + jdbcresultset.getString("city") + " Salary : " +
jdbcresultset.getInt("salary"));
}

System.out.println("data in forward direction");

System.out.println("ENO      ENAME      ESAL      EADDR");

while(jdbcresultset.previous())
{
//Retrieve data
System.out.println("id : " + jdbcresultset.getInt("id") + " First : " +
jdbcresultset.getString("first")  + " Last : " + jdbcresultset.getString("last") +  " Age : " +
jdbcresultset.getInt("age") + " City : " + jdbcresultset.getString("city") + " Salary : " +
jdbcresultset.getInt("salary"));
}
jdbcconn.close();
}
catch(Exception e)
{
System.out.println(e);
}
System.out.println("Success!");
}
}
```

☞ Check Your Progress 2

1. A transaction is a logical unit containing one or more SQL queries and is executed in totality to perform a specific task. This mechanism is to ensure that either all dependent queries or batch of queries executed successfully or none. In general, transaction management supports ACID properties – Atomicity, Consistency, Isolation and

Durability. By default, every query execution result automatically saves in JDBC applications, i.e. after the connection establishment with the database, the connection remains in an auto-commit mode. But at times, to improve the efficiency, we may require to group two or more statements to form as a part of the Transaction, and for that, we need to turn-off the auto-commit mode. This turning-off of the auto-commit mode is through setAutoCommit() method of the connection interface. This non-commit nature maintained the transactions in our JDBC applications.

2. The SavePoint option sets a Savepoint, i.e. checkpoint in the Transaction, which ensures that if rollback is required in the Transaction, it will only be up to the checkpoint instead of the complete rollback of the Transaction.

3. The concurrent access of the database may lead to the problem of the dirty read, non-repeatable reads and phantom reads which may make the database inconsistent. When a transaction reads an uncommitted value, it is a dirty read problem. When a transaction re-read a data item that is updated by another transaction, it is the non-repeatable read problem. The phantom reads problem happens when records are added or removed by another transaction, and then the transaction is executed on the database. These problems were resolved using a locking mechanism and controlled by the JDBC transaction Isolation level.

There are different transaction levels ranging from TRANSACTION_NONE, which does not support transactions to TRANSACTION_SERIALIZABLE, which strictly supports transactions following access rules and handles problems like dirty read, non-repeatable reads or phantom reads. The other Isolation level is TRANSACTION_READ_COMMITTED which support transactions and prevent dirty read but allows non-repeatable and phantom reads. The other one is the TRANSACTION_READ_UNCOMMITTED which support transactions but has no control over dirty read, non-repeatable and phantom reads. The fifth isolation level is TRANSACTION_REPEATABLE_READ which also supports transactions and allow phantom read but prevent dirty and non-repeatable reads.

☞ Check Your Progress 3

1. The JDBC RowSet interface is an extension of the ResultSet interface. It is a wrapper around a ResultSet object, and inherently derive the capabilities of ResultSet and thus offer more flexibility. A RowSet is broadly divided into Connected RowSet Objects and Disconnected RowSet Objects.

The Connected RowSet Object always remains connected with the database, and it is very similar to the ResultSet object. The Disconnected RowSet Objects are not always connected to a database which makes it very lightweight and serializable.

2. A JDBC batch contains a group of SQL statements that are processed together with one database call. This is very useful when a large number of SQL statements are executed together after putting them in a batch. The batch queries are executed with one call, and this not only reduce the network communication overhead but also some of the queries may run in parallel and thus improve the overall efficiency.

3. JDO API implicitly enables the user to access a database, offering an abstraction of persistence through a standard interface-based Java model. JDBC API and JDO API allow us to access data through the java application. In JDBC, the relational database is used as a data source, but JDO may be using any data source may be a relational database, object-oriented database or even flat files. JDBC mostly uses structured query language, while JDO uses Java code for the purpose. In the case of JDBC, we

have to write SQL queries and fetch the ResultSet into data objects, but JDO doesn't require execution of queries copying JDBC ResultSet into Java objects etc. as all this is taken care by JDO.

4.9 REFERENCES/FURTHER READING

- Herbert Schildt "Java The Complete Reference", McGraw-Hill,2017
- Horstmann, Cay S., and Gary Cornell, " *Core Java: Advanced Features*" Vol. 2. Pearson Education, 2013.
- Prasanalakshmi, "*Advanced Java Programming*", CBS Publishers 2015
- Sagayaraj, Denis, Karthik and Gajalakshmi ,"*Java Programming – for Core and Advanced Users*", Universities Press 2018
- Sharan, Kishori, "Beginning Java 8 APIs, Extensions and Libraries: Swing, JavaFX, JavaScript, JDBC and Network Programming APIs", Apress, 2014.
- Parsian, Mahmoud, "*DBC metadata, MySQL, and Oracle recipes: a problem-solution approach"* Apress, 2006.
- https://docs.oracle.com/javase/tutorial/jdbc/basics/index.html
- https://www.roseindia.net/jdbc/