Lesson: 1.4

JDBC Transactions

Objectives:

In this lesson, you will exercise to:

- ✓ Understand JDBC Transactions.
- ✓ Committing the transactions.
- ✓ Rollback the transactions.
- ✓ Use savepoints.
- ✓ Understand transactions isolation levels.

Understand JDBC Transactions

- Transactions are used to provide data integrity, correct application semantics, and a consistent view of data during concurrent access.
- A *transaction* is a set of one or more statements that are executed together as a unit, so either all of the statements are executed or none of them are executed.
- A sequence of SQL/JDBC calls that constitute an atomic unit of work: Either all of the commands in a transaction are committed as a unit, or all of the commands are rolled back as a unit.
- When to start a new transaction is a decision made implicitly by either the JDBC driver or the underlying data source.
- Although some data sources implement an explicit "*begin transaction*" statement, there is no JDBC API to do so.
- Typically, a new transaction is started when the current SQL statement requires one and there is no transaction already in place.
- The ***java.sql.Connection*** interface provides methods for implementing transactions.
- Transactions provide ACID properties: **A**tomicity, **C**onsistency, **I**ntegrity of data, and **D**urability of database changes.
- A transaction in which commands are sent to more than one DBMS server is a *distributed transaction*.

✍ Task 12 : Explore the API

12.1 Identify the constant indicating that dirty read, non-repeatable reads and phantom read can occur.
12.2 Identify the method which retrieves the current auto-commit mode for this current *Connection* object.
12.3 Identify the method which releases this *Connection* object's database and JDBC resources immediately instead of waiting for them to be automatically released.
12.4 Method which returns true if this *Connection* object is closed, false it is still open.
12.5 Identify the method which converts the given SQL statement into the system's native SQL grammar.

Committing JDBC Transactions

- An operation that causes all of the updates made during a transaction to be permanently written into the database, completing the transaction. If a transaction is explicitly or implicitly aborted (rolled back), it is incomplete and all changes made are backed out, as if the transaction never happened.
- In the JDBC API, the method *Connection.commit* commits a transaction.
- When a connection is created, it is in auto-commit mode. This means that each individual SQL statement is treated as a transaction and will be automatically committed right after it is executed.

- The way to allow two or more statements to be grouped into a transaction is to *disable auto-commit mode*. This is demonstrated in the following line of code, where *con* is an active connection:
  ➦ **con.setAutoCommit(false);**
- Once auto-commit mode is disabled, no SQL statements will be committed until you call the method commit explicitly.
- All statements executed after the previous call to the method commit will be included in the current transaction and will be committed together as a unit.
- Whenever the *commit* method is called (either automatically when auto-commit mode is enabled or explicitly when it is disabled), all changes resulting from statements in the transaction will be made permanent.

✍ Fill in Queries:

1. Complete the method declaration: **public <*return type*> nativeSQL(*Parameter1*) throws <*Exception*>**
2. Method which retrieves whether this *Connection* object is in read-only mode.
3. Write a statement which enables auto-commit mode.
4. Method which returns the current state of this *Connection* objects's auto-commit mode.
5. Complete the method declaration: **public <*return type*> setAutoCommit(*Parameter1*) throws <*Exception*>**

Task 12 Workspace

✍ Code 1 : Test the Coding Skills

Code the Program

```
// The following code, in which con is an active connection, illustrates a transaction.

con.______________________(________); //Disabling auto-commit mode.

PreparedStatement ________________ = con.______________________(

"UPDATE Purchase SET Amount = ? WHERE User_Id = ?");

updateAmount.________________(____, 5000);

updateAmount.________________(____, 1002);

updateAmount.____________________(); //Update the table.

PreparedStatement updateAmount = con.___________________________(

"UPDATE Purchase SET Amount = Amount + ? WHERE User_Id = ?");

updateAmount.________________(___, 6550);

updateAmount.________________(___, 1005);

updateAmount.____________________(); //Update the table.

con.________________();
```

Core Note

The default auto-commit mode is for an SQL statement to be committed, when it is *completed*, not when it is *executed*.

Rollback JDBC Transactions

- Undoing the changes made by a transaction.
- This can be initiated in a JDBC application by explicitly calling the method *Connection.rollback*, and, in some cases, a rollback can be spontaneously generated by the DBMS due to an error or unresolvable deadlock between transactions.
- As mentioned earlier, calling the method rollback aborts a transaction and returns any values that were modified to the values they had at the beginning of the transaction.
- If you are trying to execute one or more statements in a transaction and get a *SQLException*, you should call the method *rollback* to abort the transaction and start the transaction over again.

- Catching a *SQLException* tells you that something is wrong, but it does not tell you what was or was not committed. Since you cannot count on the fact that nothing was committed, calling the method rollback is the only way to be sure.
- The *Connection.rollback* method has been overloaded to take a *savepoint* argument.
- If any application continues and uses the results of the transaction, however, it would be necessary to include a call to *rollback* in the *catch* block in order to protect against using possibly incorrect data.
- There is one **rollback (String savepointName)** method which rolls back work to the specified savepoint.

✍ Code 2 : Test the Coding Skills

Code the Program

// The following code, in which *con* is an active connection, illustrates a transaction using *commit* and *rollback* object.

```
try{
  //Assume a valid connection object con
  con.______________________________(___________); //Disabling auto-commit mode.
  Statement stmt = con._________________________(); //Statement Object
  String SQL = "INSERT INTO Passenger  " + "VALUES (1006, 20, 'Rita', 'Tez')";
  stmt.________________________(SQL);
  //Submit a malformed SQL statement that breaks
  String SQL = "INSERTED IN Passenger  " + "VALUES (1007, 22, 'Sita', 'Singh')";
  stmt.________________________(SQL);
  // If there is no error.
  conn._______________();
}catch(___________________ se){
  // If there is any error.
  conn.__________________();  }
```

Core Note

A rollback can occur without your invoking the method *rollback*. This can happen, for example, when your computer crashes or your program terminates unexpectedly.

Using Savepoints

- Savepoints provide finer-grained control of transactions by marking intermediate points within a transaction.
- They let you roll back a transaction to a point you have set rather than having to roll back the entire transaction.
- The *DatabaseMetaData.supportsSavepoints* method can be used to determine whether a JDBC driver and DBMS support savepoints.
- Savepoints can be either named or unnamed. Unnamed savepoints are identified by an ID generated by the underlying data source.
- ***Java.sql.Savepoint*** interface provides the additional transactional control features.
- The Connection object has two new methods that help you manage savepoints:
  - **setSavepoint(String savepointName).**
  - **releaseSavepoint(Savepoint savepointName).**

- Rolling back a transaction to a savepoint.

Sample code Fragment

```
conn.createStatement();
int rows = stmt.executeUpdate("INSERT INTO TAB1 (COL1) VALUES " + "('FIRST')");
// set savepoint
Savepoint svpt1 = conn.setSavepoint("SAVEPOINT_1");
rows = stmt.executeUpdate("INSERT INTO TAB1 (COL1) " + "VALUES ('SECOND')");
....
conn.rollback(svpt1);
....
conn.commit();
```

- There is one **rollback (String savepointName)** method which rolls back work to the specified savepoint.

✍ Task 13 : Explore the API

13.1 Identify the method which returns the numeric ID of this savepoint.
13.2 Method which retrieves the name of the savepoint that this *Savepoint* object represents.
13.3 Method which takes a *Savepoint* object as a parameter and removes it and any subsequent savepoints from the current transaction.
13.4 A transaction will be started if this method is invoked and there is not an active transaction.
13.5 Method to roll back a transaction to a *Savepoint* object.

Core Note
Rolling a transaction back to a savepoint automatically releases and makes invalid any other savepoints that were created after the savepoint in question.

✍ Code 3 : Test the Coding Skills

Code the Program

```
// The following code, in which conn is an active connection, illustrates the use of
   Savepoint object.

try{
  //Assume a valid connection object conn
  conn.____________________________(__________); //Disabling auto-commit mode.
  Statement stmt = conn._________________________(); //Statement Object
  //set a Savepoint
  Savepoint savepoint1 = conn._________________________("_________________");

  String SQL = "INSERT INTO City " +  "VALUES (106, 'HYD', 'AP', 'India')";
  stmt._____________________(SQL);
  //Submit a malformed SQL statement that breaks.
  String SQL = "INSERTED IN City " + "VALUES (107, 'SECBAD', 'AP', 'India')";
  stmt._____________________(_______);
  // If there is no error, commit the changes.
  conn.________________();
} catch(SQLException se){
  // If there is any error.
  conn.______________________(__________________);
}
```

Transaction Isolation Levels

- Transaction isolation levels specify what data is “*visible*” to the statements within a transaction.
- How locks are set is determined by what is called a *transaction isolation level*, which can range from not supporting transactions at all to supporting transactions that enforce very strict access rules.
- They directly impact the level of concurrent access by defining what interaction, if any, is possible between transactions against the same target data source.
- Possible interaction between concurrent transactions is categorized as follows:

- ➦ *Dirty reads* occur when transactions are allowed to see uncommitted changes to the data.
- ➦ *Nonrepeatable* reads and
- ➦ *Phantom* reads.

- The interface *Connection* includes five values that represent the transaction isolation levels you can use in JDBC:
  - ➦ TRANSACTION_NONE.
  - ➦ TRANSACTION_READ_UNCOMMITTED.
  - ➦ TRANSACTION_READ_COMMITTED.
  - ➦ TRANSACTION_REPEATABLE.
  - ➦ TRANSACTION_SERIALIZABLE.
- A user may call the method *setIsolationLevel* to change the transaction isolation level, and the new level will be in effect for the rest of the connection session.

✍ Task 14 : Explore the API

14.1 Method which retrieves this database's default transaction isolation level.

14.2 Method which retrieves whether this database supports the given transaction isolation level.

14.3 Identify the method which allows JDBC clients to change the transaction isolation level for a given Connection object.

14.4 Identify the constant which indicates that dirty reads, non-repeatable reads and phantom reads are prevented.

14.5 Identify the method which returns the current transaction isolation level.

✍ Fill in Queries:

1. Complete the following code to get the default transaction isolation level supported by this database: **DatabaseMetaData <*variable*> = con.<*method name*> (); int level = metadata. <*method name*>();**

2. Complete the method declaration: **public <*return type*> setTransactionIsolation (*Parameter1*) throws <*Exception*>**

3. Write the statement which will allow JDBC user to instruct the DBMS to allow a value to be read before it has been committed (a “dirty read).

4. Method which retrieves whether this database allows having multiple transactions open at once (on different connections).

5. Type of exception thrown, if a driver is unable to substitute a higher transaction level.

Task 13 Workspace

Task 14 Workspace