Object-oriented Programming in Java

Session: 10

Advanced JDBC Features

Objectives

- List and describe scrollable result sets
- List different types of ResultSet and row-positioning methods
- Explain stored procedures
- Explain how to call a stored procedure using JDBC API
- Describe the steps to update records
- Explain the steps of implementing transactions using JDBC
- List the enhancements of JDBC 4.0 API
- Explain RowSet and its type
- Describe JDBC 4.1 RowSetProvider and RowSetFactory Interfaces

Introduction

- The `ResultSet` object in JDBC API represents a SQL result set in the JDBC application.
- A default result set object cannot be updated or scrolled backward and forward.
- The characteristics of `ResultSet` are as follows:
  - **Scrollable -** It refers to the ability to move backward as well as forward through a result set.
  - **Updatable -** It refers to the ability to update data in a result set and then copy the changes to the database. This includes inserting new rows into the result set or deleting existing rows.
  - **Holdable -** It refers to the ability to check whether the cursor stays open after a COMMIT.

Scrollable ResultSet [1-3]

- A scrollable result set allows the cursor to be moved to any row in the result set.
- This capability is useful for GUI tools for browsing result sets.
- Since scrollable result sets involve overhead, they should be used only when the application needs scrolling.
- You can create a scrollable `ResultSet` through methods of the `Connection` interface.

Scrollable ResultSet [2-3]

The following table lists the methods that can be invoked on the connection instance for returning a scrollable `ResultSet`:

Method	Syntax	Code
`createStatement()`	`public Statement createStatement(int resultSetType, int resultSetConcurrency) throws SQL Exception` where, `resultSetType`: It is this argument that will help to create a scrollable `ResultSet`. It represents the constant values that can be `ResultSet.TYPE_FORWARD`, `ResultSet.TYPE_SCROLL_SENSITIVE`, or `ResultSet.TYPE_SCROLL_INSENSITIVE`. `resultSetConcurrency`: Represents one of the two `ResultSet` constants for specifying whether a result set is read-only or updatable. The constant values for concurrency type can be `ResultSet.CONCUR_READ_ONLY` or `ResultSet.CONCUR_UPDATABLE`.	`Statement st = cn.createStatement( ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_READ_ONLY); ResultSet rs = st.executeQuery("SELECT EMP_NAME, DEPT FROM EMPLOYEES");`

Scrollable ResultSet [3-3]

Method	Syntax	Code
`prepareCall()`	`public CallableStatement prepareCall(String sql, int resultSetType, int resultSetConcurrency) throws SQLException` where, `sql` represents a `String` object containing the SQL statements to be sent to the database. `resultSetType` has the same attributes as in the `createStatement` method. `resultSetConcurrency` represents one of the two `ResultSet` constants for specifying whether a result set is read-only or updatable.	`CallableStatement cs = cn.prepareCall("? = CALL EMPLOYEE(?, ?, ?)", ResultSet.TYPE_SCROLL_ INSENSITIVE, ResultSet. CONCUR_ READ_ONLY);`

Types of ResultSet Values

TYPE_FORWARD_ONLY

- A cursor that can only be used to process from the beginning of a `ResultSet` to the end of it.
- The cursor only moves forward.
- This is the default type.

TYPE_SCROLL_INSENSITIVE

- A cursor that can be used to scroll in various ways through a `ResultSet`.
- This type of cursor is insensitive to changes made to the database while it is open.

TYPE_SCROLL_SENSITIVE

- A cursor that can be used to scroll in various ways through a `ResultSet`.
- This type of cursor is sensitive to changes made to the database while it is open.

Row Positioning Methods [1-3]

Method	Description
`next()`	This method moves the cursor forward one row in the `ResultSet` from the current position. The method returns `true` if the cursor is positioned on a valid row and `false` otherwise.
`previous()`	The method moves the cursor backward one row in the `ResultSet`. The method returns `true` if the cursor is positioned on a valid row and `false` otherwise.
`first()`	The method moves the cursor to the first row in the `ResultSet`. The method returns `true` if the cursor is positioned on the first row and `false` if the `ResultSet` is empty.
`last()`	The method moves the cursor to the last row in the `ResultSet`. The method returns `true` if the cursor is positioned on the last row and `false` if the `ResultSet` is empty.
`beforeFirst()`	The method moves the cursor immediately before the first row in the `ResultSet`. There is no return value from this method.

Row Positioning Methods [2-3]

Method	Description
`afterLast()`	The method moves the cursor immediately after the last row in the `ResultSet`. There is no return value from this method.
`relative (int rows)`	The method moves the cursor relative to its current position. If row value is 0, this method has no effect. If row value is positive, the cursor is moved forward that many rows. If there are fewer rows between the current position and the end of the `ResultSet` than specified by the input parameters, this method operates same as `afterLast()` method. If row value is negative, the cursor is moved backward that many rows. The method returns `true` if the cursor in positioned on a valid row and `false` otherwise.

Row Positioning Methods [3-3]

Method	Description
`absolute (int row)`	The method moves the cursor to the row specified by row value. If row value is positive, the cursor is positioned that many rows from the beginning of the `ResultSet`. The first row is numbered 1, the second is 2, and so on. If row value is negative, the cursor is positioned that many rows from the end of the `ResultSet`. The last row is numbered -1, the second to last is -2, and so on. If row value is 0, this method operates similar to `beforeFirst()`. The method returns `true` if the cursor is positioned on a valid row and `false` otherwise.

Updatable ResultSet

- Updatable `ResultSet` is the ability to update rows in a result set using Java programming language methods rather than SQL commands.
- An updatable `ResultSet` will allow the programmer to change the data in the existing row, to insert a new row, or delete an existing row.
- The `newUpdateXXX()` methods of `ResultSet` interface can be used to change the data in an existing row.
- When using an updatable result set, it is recommended to make it scrollable.

Concurrency in ResultSet

- Concurrency is a process wherein two events take place in parallel.
- The concurrency type of a result set determines whether it is updatable or not.
- The constant values that can be assigned for specifying the concurrency types are as follows:
  - `CONCURRENCY.READ_ONLY`: The result set cannot be modified and hence, it is not updatable in any way.
  - `CONCURRENCY.UPDATABLE`: The update, insert, and delete operations can be performed on the result set and the changes are copied to the database.

Updating a Row

- There are two steps involved in this process.
  - **First step**: Change the values for a specific row using various `update<Type>` methods, where `<Type>` is a Java data type.
  - **Second step**: Apply the changes to the rows of the underlying database.
- The database itself is not updated until the second step.
- Updating columns in a `ResultSet` without calling the `updateRow()` method does not make any changes to the database.
- Once the `updateRow()` method is called, changes to the database are final and cannot be undone.

Steps for Inserting a Row [1-2]

Step 1: Positioning the Cursor

- Move to the insert row, using the `moveToInsertRow()` method.
- The following Code Snippet demonstrates the `moveToInsertRow()` method:

Code Snippet

```
// Create an updatable result set
   ResultSet rs = stmt.executeQuery("SELECT NAME,
EMPLOYEE_ID FROM EMPLOYEES");
// Move cursor to the "insert row"
   rs.moveToInsertRow();
```

Steps for Inserting a Row [2-2]

Step 2: Updating the Columns

- Use `updateXXX()` methods to load new data into the insert row.
- The following Code Snippet demonstrates the `updateXXX()` method:

Code Snippet

```
// Set values for the new row
  rs.updateString(1, "William Ferris");
  rs.updateInt(2, 35244);
```

Step 3: Inserting the Row

- The insertRow() method is called to append the new row to the ResultSet and the underlying database.
- The following Code Snippet demonstrates the `insertRow()` method:

Code Snippet

```
// Commit appending of new row to the result set
   rs.insertRow();
```

Steps for Deleting a Row

Position the cursor

Call the `deleteRow()` method to commit the deletion of the row

Stored Procedure

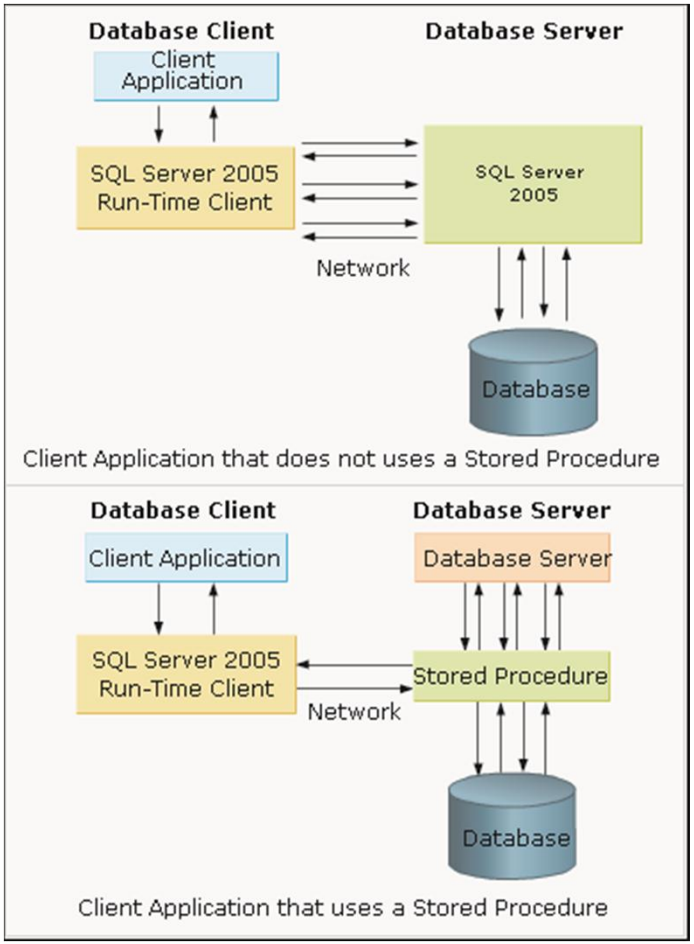
- A stored procedure can be defined as a group of SQL statements performing a particular task.
- Stored procedures having any combination of input, output, or input/output parameters can be compiled and executed.
- As stored procedures are pre-compiled, they are faster and more efficient than using individual SQL query statements.
- Following are the basic elements that a stored procedure consists of:
  - SQL statements
  - Variables
  - One or more parameters
- Statements form the main element of the stored procedure.

Characteristics of Stored Procedures

They contain SQL statements using constructs and control structures.

They can be invoked by name in an application that is using SQL.

They allow an application program to run in two parts such as the application on the client and the stored procedure on the server.



Creating a Stored Procedure Using Statement Object [1-2]

Step 1: Create stored procedure and store it in a String variable

The following Code Snippet shows the code to declare the string variable containing the definition of a stored procedure:

Code Snippet

```
//String createProcedure = "Create Procedure DISPLAY_PRODUCTS
" + "as " + "select PRODUCTS.PRD_NAME, COFFEES.COF_NAME " +
"from PRODUCTS, COFFEES " + "where PRODUCTS.PRD_ID =
COFFEES.PRD_ID " + "order by PRD_NAME";
```

Step 2: Use the Statement object

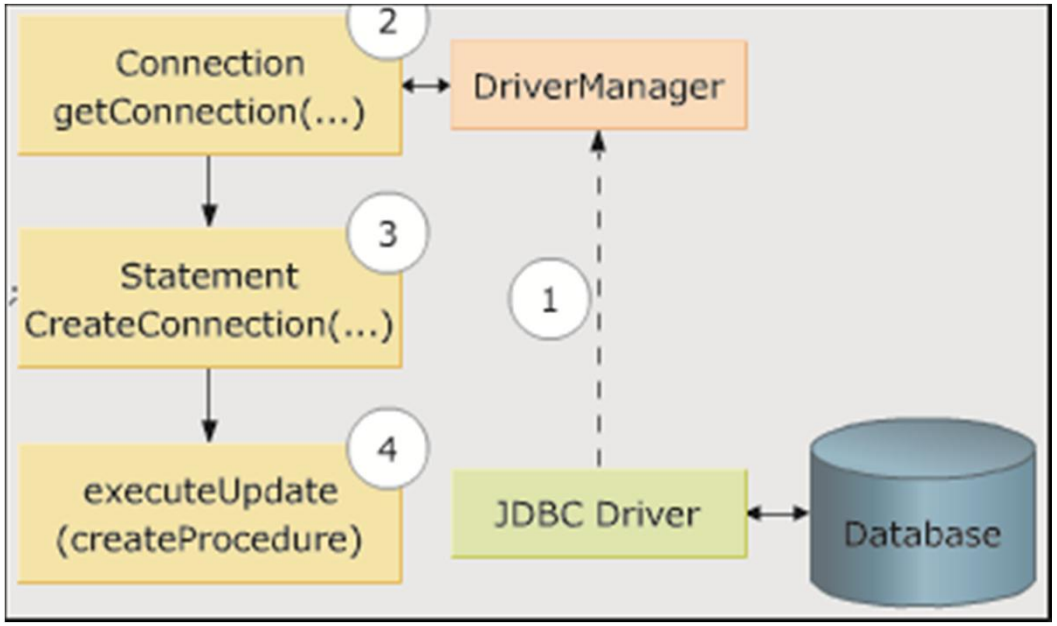
The following Code Snippet shows the use of the `Statement` object:

Code Snippet

```
// An active connection cn is used to create a Statement
object
   Statement st = cn.createStatement();
// Execute the stored procedure
   st.executeUpdate(createProcedure);
```

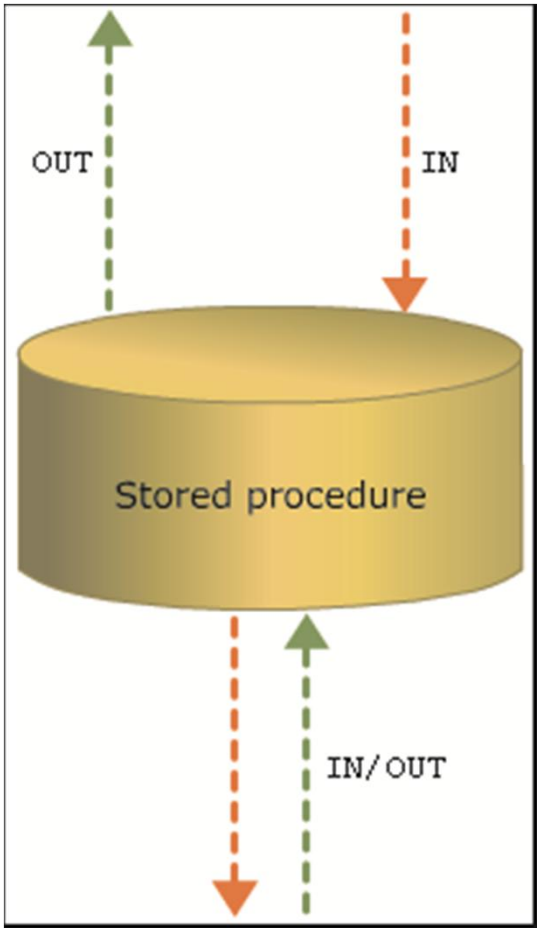
Creating a Stored Procedure Using `Statement` Object [2-2]

- Figure displays the stored procedure using `Statement` object.



Parameters of a Stored Procedure

- **IN**
  - An IN parameter is used to pass values into a stored procedure.
  - The value of an IN parameter cannot be changed or reassigned within the module and hence is constant.
- **OUT**
  - An OUT parameter's value is passed out of the stored procedure module, back to the calling block.
  - A value can be assigned to an OUT parameter in the body of a module.
  - The value stored in an OUT parameter is a variable and not a constant.
- **IN/OUT**
  - An IN/OUT parameter is a parameter that can act as an IN or an OUT parameter or both.
  - The value of the IN/OUT parameter is passed in the stored procedure and a new value can be assigned to the parameter and passed out of the module.
  - An IN/OUT parameter behaves same as an initialized variable.



Creating a CallableStatement Object [1-13]

- A stored procedure can be called from a Java application with the help of a `CallableStatement` object.
- A `CallableStatement` object does not contain the stored procedure itself but contains only a call to the stored procedure.
- The call to a stored procedure is written in an escape syntax.
- The call may take two forms, such as with a result parameter and without a result parameter.
- The result parameter is a value returned by a stored procedure, similar to an `OUT` parameter.
- Both the forms have a different number of parameters used as input (`IN` parameters), output (`OUT` parameters), or both (`INOUT` parameters). A question mark (`?`) is used to represent a placeholder for a parameter.

Creating a CallableStatement Object [2-13]

Syntax for calling a stored procedure without parameters is as follows:

Syntax

```
{call procedure_name}
```

Syntax for calling a stored procedure in JDBC is as follows:

Syntax

```
{call procedure_name[(?, ?, ...)]}
```

Placeholders enclosed in square brackets indicate that they are optional.

Syntax for a procedure that returns a result parameter is as follows:

Syntax

```
{? = call procedure_name[(?, ?, ...)]}
```

Creating a CallableStatement Object [3-13]

- The `CallableStatement` inherits methods from the `Statement` and `PreparedStatement` interfaces.
- All methods that are defined in the `CallableStatement` interface deal with `OUT` parameters.
- The `getXX()` methods such as `getInt()`, `getString()` in a `ResultSet` will retrieve values from a result set whereas in a `CallableStatement`, they will retrieve values from the `OUT` parameters or return values of a stored procedure.
- `CallableStatement` objects are created using the `prepareCall()` method of the `Connection` interface.
- The section enclosed within the curly braces is the escape syntax for stored procedures.
- The driver converts the escape syntax into native SQL used by the database.

Creating a CallableStatement Object [4-13]

Syntax

```
CallableStatement cst = cn.prepareCall("{call
functionname(?, ?)}");
```

where,

`cst` is the name of the `CallableStatement` object.

`functionname` is the name of function/procedure to be called.

Creating a CallableStatement Object [5-13]

IN Parameters

- The `set<Type>()` methods are used to pass any IN parameter values to a `CallableStatement` object.
- These `set<Type>()` methods are inherited from the `PreparedStatement` object.
- The type of the value being passed in, determines which `set<Type>()` method to use.

OUT Parameters

- In case the stored procedure returns some values (`OUT` parameters), the JDBC type of each `OUT` parameter must be registered before executing the `CallableStatement` object.
- The `registerOutParameter()` method registers the JDBC type.

Creating a CallableStatement Object [6-13]

- The `get<Type>()` methods of `CallableStatement` are used to retrieve the `OUT` parameter value.
- The `registerOutParameter()` method uses a JDBC type (so that it matches the JDBC type that the database will return), and `get<Type>()` casts this to a Java type.
- The following Code Snippet demonstrates how to use the `registerOutParameter()` method:

Code Snippet

```
. . .
CallableStatement cs = cn.prepareCall("{call getData(?,
?)}");
   cs.registerOutParameter(1, java.sql.Types.INTEGER);
   cs.registerOutParameter(2, java.sql.Types.DECIMAL, 3);
   cs.executeQuery();
   int x = cs.getInt(1);
   java.math.BigDecimal n = cs.getBigDecimal(2, 3);
. . .
```

Creating a CallableStatement Object [7-13]

The following Code Snippet demonstrates how to retrieve an OUT parameter returned by a stored procedure:

Code Snippet

```
import java.sql.*;
import java.util.*;
class CallOutProc {
   Connection con;
   String url;
   String serverName;
   String instanceName;
   String databaseName;
   String userName;
   String password;
   String sql;
```

Creating a CallableStatement Object [8-13]

```
  CallOutProc() {
      url = "jdbc:sqlserver://";
      serverName = "10.2.1.51";
      instanceName = "martin";
      databaseName = "DeveloperApps";
      userName = "sa";
      password = "playware";
  }
 private String getConnectionUrl() {
  // Constructing the connection string
  return url + serverName + ";instanceName = " +instanceName
+" ;DatabaseName = " +databaseName;
 }
```

Creating a CallableStatement Object [9-13]

```
private java.sql.Connection getConnection() {
 try {
  Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver");
  // Establishing the connection
     con = DriverManager.getConnection(getConnectionUrl(), userName, password);
    if(con != null)
      System.out.println("Connection Successful!");
 } catch(Exception e) {
        e.printStackTrace();
        System.out.println("Error Trace in getConnection() : "
        + e.getMessage());
 }
   return con;
}
```

Creating a CallableStatement Object [10-13]

```
public void display(){
 try {
   con = getConnection();
   CallableStatement cstmt = con.prepareCall("{call
recalculatetotal (?, ?)}");
   cstmt.setInt(1,2500);
   cstmt.registerOutParameter(2, java.sql.Types.INTEGER);
   cstmt.execute();
   int maxSalary = cstmt.getInt(2);
   System.out.println(maxSalary);
 } catch(SQLException ce) {
       System.out.println(ce);
 }
}
```

Creating a CallableStatement Object [11-13]

```
 public static void main(String args[]) {
    CallOutProc proObj = new CallOutProc();
     proObj.display();
 }
}
```

- The `CallableStatement` makes a call to the stored procedure called `recalculatetotal`.
- The integer variable '**a**' from the stored procedure is initialized to a value of 2500 by passing an argument through the `setInt()` method.
- The `OUT` parameter is then registered with JDBC as belonging to data type `java.sql.Types.Integer`.

Creating a `CallableStatement` Object [12-13]

- The `CallableStatement` is then executed and this, in turn, executes the stored procedure `recalculatetotal`.
- The value of the `OUT` parameter is retrieved by invoking the `getInt()` method.
- This value is the maximum salary from the **Employee** table and stored in an integer variable **`maxSalary`** and displayed.

Creating a CallableStatement Object [13-13]

A procedure for recalculating the salary of the highest salary earner is shown in the following Code Snippet:

Code Snippet

```
create procedure recalculatetotal
   @a int, @inc_a int OUT
as
select @inc_a = max(salary) from Employee
set @inc_a = @a * @inc_a;
```

The procedure will accept a value and will store it in an `OUT` parameter.

Batch Update

- Batch update can be defined as a set of multiple update statements that is submitted to the database for processing as a batch.
- In Java, the `Statement`, `PreparedStatement`, and `CallableStatement` objects can be used to submit batch updates.
- The benefits of batch updating is that it allows you to request records, bring them to the client, make changes to the records on the client side, and then send the updated record back to the data source at some other time.
- Submitting multiple updates together, instead of individually, can greatly improve performance.
- Also, batch updating is used when there is no need to maintain a constant connection to the database.

Batch Update Using Statement Interface

Batch Update Using PreparedStatement Interface [1-3]

- The batch update facility is used with a `PreparedStatement` to associate multiple sets of input parameter values with a single `PreparedStatement` object.
- The `addBatch()` method of the `Statement` interface is given an SQL update statement as a parameter, and the SQL statement is added to the `Statement` object's list of commands to be executed in the next batch.
- `PreparedStatement` interface allows creating parameterized batch update.
- The `setXXX()` methods of the `PreparedStatement` interface are used to create each parameter set, while the `addBatch()` method adds a set of parameters to the current batch.

Batch Update Using PreparedStatement Interface [2-3]

- Finally, the `executeBatch()` method of the `PreparedStatement` interface is called to submit the updates to the DBMS, which also clears the statement's associated list of batch elements.
- The following Code Snippet shows how to perform batch updates using `PreparedStatement`:

Code Snippet

```
// Turn off auto-commit
   cn.setAutoCommit(false);
// Creating an instance of Prepared Statement
   PreparedStatement pst = cn.prepareStatement("INSERT INTO
EMPLOYEES VALUES (?, ?)");
// Adding the calling statement batches
  pst.setInt(1, 5000);
  pst.setString(2, "Roger Hoody");
```

Batch Update Using PreparedStatement Interface [3-3]

Code Snippet

```
  pst.addBatch();
  pst.setInt(1, 6000);
  pst.setString(2, "Kelvin Keith");
  pst.addBatch();
// Submit the batch for execution
  int[] updateCounts = pst.executeBatch();
// Enable auto-commit mode
   cn.commit();
```

- The `pst.executeBatch()` method is called to submit the updates to the DBMS.
- Calling `pst. executeBatch()` clears the statement's associated list of batch elements.
- The array returned by `pst.executeBatch()` contains an element for each set of parameters in the batch, similar to the case for `Statement` interface.

Batch Update Using CallableStatement Interface [1-2]

- The functionality of a `CallableStatement` object and a `PreparedStatement` object is same.
- The batch update facility on a `CallableStatement` object can call only stored procedures that take input parameters or no parameters at all.
- Also, the stored procedure must return an update count.
- The `executeBatch()` method of the `CallableStatement` interface that is inherited from `PreparedStatement` interface will throw a `BatchUpdateException` if the return value of stored procedure is anything other than an update count or takes `OUT` or `IN/OUT` parameters.

Batch Update Using CallableStatement Interface [2-2]

The following Code Snippet shows batch update using `CallableStatement` interface.

Code Snippet

```
// Creating an instance of Callable Statement
CallableStatement cst = cn.prepareCall("{call
updateProductDetails(?, ?)}");
// Adding the calling statement batches
   cst.setString(1, "Cheese");
   cst.setFloat(2, 70.99f);
   cst.addBatch();
   cst.setString(1, "Almonds");
   cst.setFloat(2, 80.99f);
   cst.addBatch();
// Submitting the batch for execution
   int [] updateCounts = cst.executeBatch();
// Enabling auto-commit mode
   cn.commit();
```

Transactions

- A transaction is a set of one or more statements that are executed together as a unit.
- This ensures that either all the statements in the set are executed or none of them is executed.
- Transactions also help to preserve the integrity of the data in a table.
- To avoid conflicts during a transaction, a DBMS will use locks, which are mechanisms for blocking access by others to the data that is being accessed by the transaction.
- Once a lock is set, it will remain in force until the transaction is committed or rolled back.

Properties of Transaction

- Atomicity
- Consistency
- Isolation
- Durability

Implementing Transactions Using JDBC [1-5]

Step 1: Start the Transaction

- When a connection is created, by default, it is in the auto-commit mode.
- When you disable auto-commit, the start and end of a transaction is defined which lets you determine whether to commit or rollback the entire transaction.
- To disable a connection's auto-commit mode, you invoke the `setAutoCommit()` method, which accepts a single boolean parameter, as shown `cn.setAutoCommit(false);`
- To start the transaction, the active connection auto-commit mode is disabled.
- Once auto-commit mode is disabled, no SQL statements will be committed until the method commit is called explicitly.

Implementing Transactions Using JDBC [2-5]

- All statements executed after the previous call to the method `commit` are included in the current transaction and are committed together as a unit.

Step 2: Perform Transactions

The second step is to perform the transaction as shown in the following Code Snippet:

Code Snippet

```
// Creating an instance of Callable Statement
     CallableStatement cst = cn.prepareCall("{call
updateProductDetails(?, ?)}");
// Adding the calling statement batches
    cst.setString(1, "Cheese");
    cst.setFloat(2, 70.99f);
    cst.addBatch();
    cst.setString(1, "Almonds");
    updateTotal.setString(2, "Russian");
    updateTotal.executeUpdate();
```

Implementing Transactions Using JDBC [3-5]

Step 3: Use SavePoint

The third step is to use 'Savepoint' in the transaction as shown in the following Code Snippet:

Code Snippet

```
// Create an instance of Statement object
    Statement st = cn.createStatement();
    int rows = st.executeUpdate("INSERT INTO EMPLOYEE
(NAME) VALUES (?COMPANY?)");
// Set the Savepoint
    Savepoint svpt = cn.setSavepoint("SAVEPOINT_1");
    rows = st.executeUpdate("INSERT INTO EMPLOYEE (NAME)
VALUES (?FACTORY?)");
```

Implementing Transactions Using JDBC [4-5]

- The code inserts a row into a table, sets the `Savepoint svpt`, and then inserts a second row.
- When the transaction is later rolled back to `svpt`, the second insertion is undone, but the first insertion remains intact.
- Thus, when the transaction is committed, only the row containing `?COMPANY?` will be added to `EMPLOYEE`.
- The method `cn.releaseSavepoint` takes a `Savepoint` object as a parameter and removes it from the current transaction.

Implementing Transactions Using JDBC [5-5]

Step 4: Close the Transaction

- A transaction can end either with a commit or with a rollback.
- When a transaction commits, the data modifications made by its statements are saved.
- If a statement within a transaction fails, the transaction rolls back, undoing the effects of all statements in the transaction.
- The following Code Snippet demonstrates how to close the transaction:

Code Snippet

```
. . .
// End the transaction
   cn.rollback(svpt);
  OR
. . .
   cn.commit();
```

JDBC 4.0 Features [1-2]

- JDBC 4.0 has redefined subclasses of SQLExceptions.
- Application servers use these objects to look for vendor-specific extensions inside standard JDBC objects such as Statements and Connections.
- There are new methods included in connection pool such as `javax.sql. PooledConnection`, `addStatementEventListener`, and `removeStatementEventListener`.
- There are new implementations of `javax.sql.DataSource` that help easy development with JDBC 4.0.
- There are new overloads of the streaming methods in `CallableStatement`, `PreparedStatement`, and `ResultSet` to define long lengths or omit length arguments.

JDBC 4.0 Features [2-2]

- New methods are added to the following interfaces:
  - `javax.sql.DatabaseMetaData`
  - `javax.sql.Statement`
  - `javax.sql.Connection`
- With JDBC 4.0, the JDBC application does not have to register drivers programmatically.

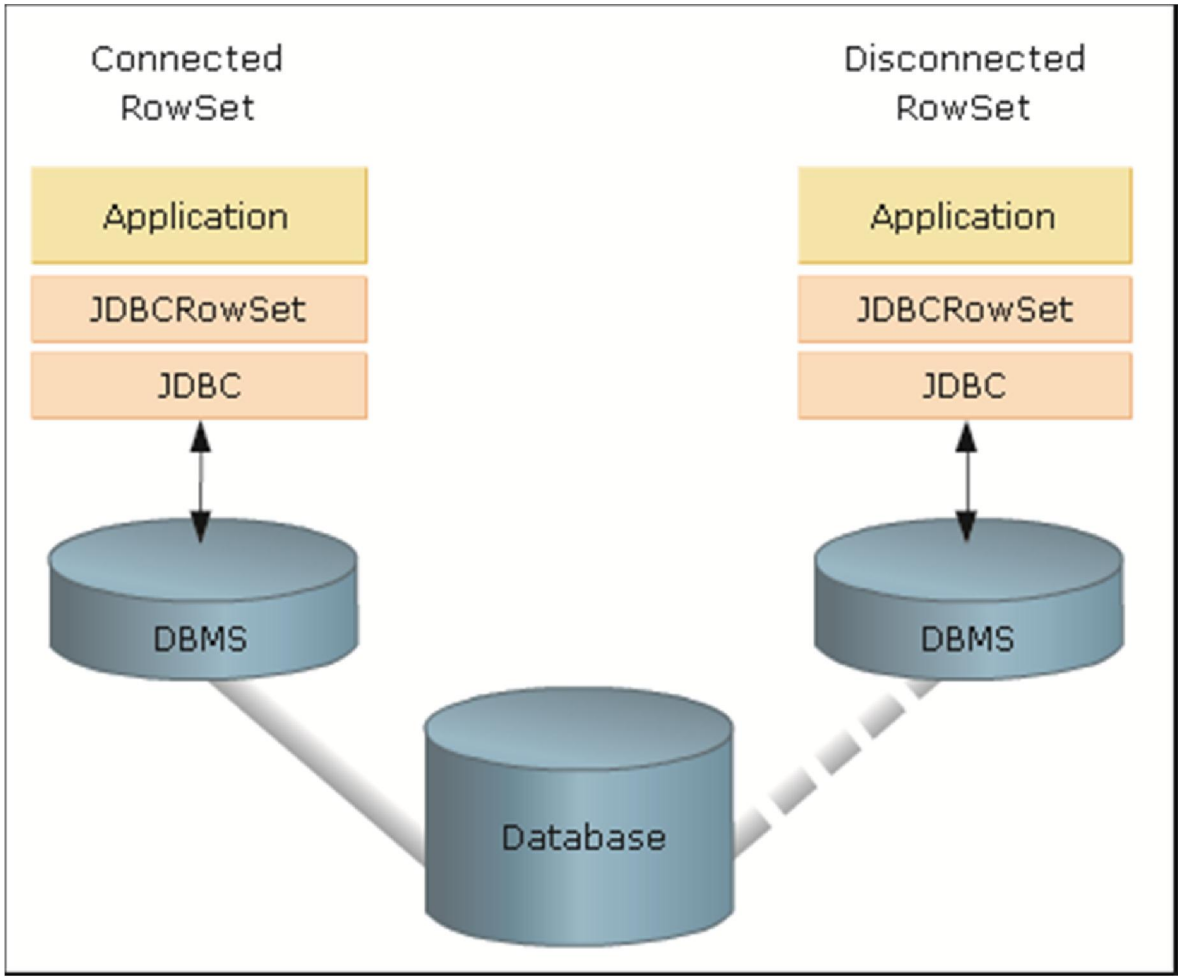
RowSet

- `RowSet` is an interface in the new standard extension package `javax.sql.rowset` and is derived from the `ResultSet` interface.
- It typically contains a set of rows from a source of tabular data similar to a result set.
- It can be configured to connect to and read/write data from a JDBC data source.
- A JDBC `RowSet` object is more easier to use than a result set.

Benefits of Using RowSet over ResultSet

- It provides scrollability and updatability for any kind of DBMS or driver.
- It is a JavaBeans component that can be used to notify other registered GUI components of a change.

Different Types of RowSets



Implementation of Connected RowSet [1-2]

JdbcRowSet Interface

- `JdbcRowSet` is an interface in the `javax.sql.rowset` package.
- The `JdbcRowSetImpl` is the reference implementation class of the `JdbcRowSet` interface.
- If a `JdbcRowSet` object has been constructed using the default constructor, then the new instance is not usable until its `execute()` method is invoked.
- The `execute()` method can only be invoked on such an instance if all the other properties such as command, username, password, and URL have been set.
- These properties can be set using the methods of the `RowSet` and the `JdbcRowSet` interface.

Implementation of Connected RowSet [2-2]

JdbcRowSetImpl Class

- The `JdbcRowSetImpl` class is the standard implementation of the `JdbcRowSet` interface.
- An instance of this class can be obtained by using any of the following constructors:
  - `JdbcRowSetImpl()`
  - `JdbcRowSetImpl(Connection con)`
  - `JdbcRowSetImpl(String url, String user, String password)`
  - `JdbcRowSetImpl(java.sql. ResultSet res)`

Using JDBCRowSet Object

Some of the most commonly used methods of the `JdbcRowSetImpl` class are as follows:

- `absolute(int row)`
- `afterLast()`
- `beforeFirst()`
- `first()`
- `deleteRow()`
- `insertRow()`
- `last()`
- `boolean isLast()`
- `boolean isAfterLast()`
- `isBeforeFirst()`
- `next()`
- `previous()`
- `moveToCurrentRow()`
- `moveToInsertRow()`
- `updateDate(int column, java.sql.Date date)`
- `updateInt(int column, int i)`
- `updateString(int column, String str)`
- `updateRow()`

Implementation of a Disconnected RowSet

- A disconnected `RowSet` is useful since it does not require a continuous connection with the database.
- A disconnected `RowSet` stores its data in memory and operates on that data rather than directly operating on the data in the database.
- A `CachedRowSet` object is an example of a disconnected `RowSet` object.
- A disconnected `RowSet` needs reader and writer objects to read and write data from its data source.
- A `CachedRowSet` object can be created in any of the following ways:
  - Using the default constructor
  - Using the `SyncProvider` implementation

Using CachedRowSet Object [1-4]

- A row of data can be updated, inserted, and deleted in a `CachedRowSet` object.
- Changes in data are reflected on the database by invoking the `acceptChanges()` method.

Update

- Updating a record from a `RowSet` object involves navigating to that row, updating data from that row and finally, updating the database.
- The following Code snippet illustrates the updation of row:

Code Snippet

```
if (crs.getInt("EMP_ID") == 1235) {
int currentQuantity = crs.getInt("BAL_LEAVE") + 1;
System.out.println("Updating balance leave to " +
currentQuantity);
crs.updateInt("BAL_LEAVE", currentQuantity + 1);
crs.updateRow();
// Synchronizing the row back to the DB
crs.acceptChanges(con);
```

Using CachedRowSet Object [2-4]

Insert

- To insert a record into the `CachedRowSet` object the `moveToInsertRow()` method is invoked.
- The current cursor position is remembered and the cursor is then positioned on an insert row.
- The insert row is a special buffer row provided by an updatable result set for constructing a new row.
- When the cursor is in this row only the `update`, `get`, and `insertRow()` methods can be called.
- All the columns must be given a value before the `insertRow()` method is invoked.
- The `insertRow()` method inserts the newly created row in the result set.

Using CachedRowSet Object [3-4]

- The `moveToCurrentRow()` method moves the cursor to the remembered position.

Delete

- Deleting a row from a `CachedRowSet` object is simple.
- The following Code Snippet illustrates this:

Code Snippet

```
while (crs.next()) {
    if (crs.getInt("EMP_ID") == 12345) {
        crs.deleteRow();
        break;
    }
}
```

Using CachedRowSet Object [4-4]

Retrieve

- A `CachedRowSet` object is scrollable, which means that the cursor can be moved forward and backward by using the `next()`, `previous()`, `last()`, `absolute()`, and `first()` methods.
- Once the cursor is on the desired row, the getter methods can be invoked on the `RowSet` to retrieve the desired values from the columns.

Event Notification Mechanism in RowSet

- A `RowSet` object is inherently a JavaBeans component.
- The fields of a `RowSet` are the JavaBean properties of the `RowSet`.
- `RowSet` objects follow the JavaBeans Event Notification Model for processing events.
- According to this model, all components that need to be notified of an event need to be registered as event listeners for the component generating the events.
- The following events trigger notifications in `RowSet` objects:
  - Movement of a cursor
  - Insertion, updation, or deletion of a row
  - Changing the entire `RowSet` contents

Summary

- A ResultSet object maintains a cursor pointing to its current row of data.
- Updatable ResultSet is the ability to update rows in a result set using methods in the Java programming language rather than SQL commands.
- A stored procedure is a group of SQL statements.
- A batch update is a set of multiple update statements that is submitted to the database for processing as a batch.
- 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 the statements is executed.
- RowSet is an interface that is derived from the ResultSet interface.
- A JdbcRowSet is the only implementation of a connected RowSet.