Session: 2

Features of JDBC 4.0 and JDBC 4.1

Objectives

- Explain Auto loading of JDBC driver and enhancement in connection management
- Describe the ResultSet interface and SQL Exception Handling enhancements
- Explain ROWID Support and enhancements in interfaces and Large Data Objects
- Explain the use of Scalar functions and try-with resources
- Explain the RowSetFactory interface and RowSetProvider class
- Describe JoinRowSet, WebRowSet, and FilteredRowSet

Auto Loading of JDBC Driver

- In earlier versions of JDBC, `Class.forName()` method was used to explicitly load the database driver.
- The following Code Snippet shows an example where an “sqlserver” driver is automatically loaded:

Code Snippet:

```
...
String connectionUrl =
“jdbc:sqlserver://10.2.10.72\\SQLExpress;” +
“databaseName=master;user=sa; password=playware_123;”;
Connection con =
DriverManager.getConnection(connectionUrl);
...
```

- `DriverManager` class is enhanced to automatically locate suitable driver.

Connection Management Enhancements 1-2

Methods are added to `Connection` and `Statement` interfaces for more efficient connection management.

Following are the methods added in the `Connection` interface:

- Methods to establish the validity of a connection – `isValid()`
- Methods to set and retrieve client information - `getClientInfo()` and `setClientInfo()`

The following Code Snippet shows the usage of `isvalid()` method:

Code Snippet:

```
...
int connectionTime = 20;
Connection.isValid(connectionTime);
...
```

Connection Management Enhancements 2-2

Following table shows the methods added in the Statement interface.

Method	Description
`public boolean isClosed()`	Returns true, if the `Statement` object is closed.
`public void setPoolable(true)`	Requests to pool `PreparedStatement` and `CallableStatement` objects for better performance.
`public boolean isPoolable()`	Returns true, if the `Statement` object is poolable.

ResultSet Interface 1-4

ResultSet interface is typically a table of data that represents the output of a SQL query. Following enhancements are made to it in JDBC 4.0/4.1:

- The `ResultSet` can be edited offline.
- It supports `SQLXML` datatype.
- Serialize and deserialize `ResultSet` to XML type and vice versa.
- The following code creates the structure of the XML data:

Code Snippet:

```
...
public class XMLStoreRetrieveData {
static final String Employee1 =”<Details>” +
“<EmployeeId>E001</EmployeeId>” + “<Name>Richard
Henry</Name>” + “</Details>”;
static final String Employee2 =”<Details>” +
“<EmployeeId>E002</EmployeeId>” +”<Name>William
Paul</Name>” + “</Details>”;
static final String[] Employees = {Employee1,
```

ResultSet Interface 2-4

```
Employee2};
...
}
...
```

- The following Code Snippet shows how to insert XML data into a database table using `SQLXML` object:

Code Snippet:

```
...
public void storeXMLData(Connection cn) {
try {
stmtDetails = cn.createStatement();
stmtDetails.execute("DROP Table EmployeeDetails");
stmtDetails.execute("CREATE TABLE EmployeeDetails(Id
INTEGER, Name XML)");
System.out.println("Created table: EmployeeDetails");
stmtDetails.close();
int id = 1;
```

ResultSet Interface 3-4

```
for (String store : Employees) {
pstDetails = cn.prepareStatement("INSERT INTO
EmployeeDetails(Id, Name) VALUES (?, ?)");
SQLXML sqlXml = cn.createSQLXML();
sqlXml.setString(store);
pstDetails.setInt(1, id++);
pstDetails.setSQLXML(2, sqlXml);
pstDetails.executeUpdate();
sqlXml.free();
}
System.out.println("Inserted data into EmployeeDetails
table.");
} catch (SQLException e) {
e.printStackTrace();
}
}
...
```

ResultSet Interface 4-4

- The following Code Snippet shows how to retrieve data with an `SQLXML` object from a `ResultSet` object:

```
...
public void retrieveXmlData(Connection cn) {
try {
pstDetails = cn.prepareStatement("SELECT * FROM
EmployeeDetails");
rsDetails = pstDetails.executeQuery();
while (rsDetails.next()) {
SQLXML sqlXml = rsDetails.getSQLXML("Name");
System.out.println(sqlXml.getString());
}
} catch (SQLException ex) {
ex.printStackTrace();
}
}
...
```

SQL Exception Handling 1-4

`SQLException` class is defined to handle database related exceptions

Iterable SQL exceptions or chained exceptions

- `getNextException()` method helps retrieving all the chained exceptions, which are can then be handled.
- Enhanced for-each loop for chained exception handling.
- `getSQLState()`, `getErrorCode()`, `getCause()`, and `getMessage()` methods are used to retrieve the information pertaining to various exceptions.

SQL Exception Handling 2-4

Following code shows usage of `getNextException()` method.

Code Snippet:

```
...
try {
// Database Access Code
throw new SQLException ("Cannot access database file", new
java.
io.IOException("File I/O Access Error"));
} catch(SQLException se) {
System.out.println(se.getMessage());
SQLException nextException = se.getNextException();
while (nextException != null) {
System.out.println(nextException.getMessage());
nextException = nextException.getNextException();
}
}
...
```

SQL Exception Handling 3-4

- Following code shows the use of enhanced `for` loop feature.

Code Snippet:

```
...
try {
// Database Access Code
throw new SQLException ("Cannot access database file",
new java.
io.IOException("File I/O Access Error"));
} catch(SQLException se) {
for(Throwable e : se ) {
System.out.println("Exception type occurred: " + e);
}
}
...
```

SQL Exception Handling 4-4

- Following code shows the usage of `getSQLState()`, `getErrorCode()`, and `getMessage()` methods.

Code Snippet:

```
...
try {
// Database Access Code
throw new SQLException ("Cannot access database file", new
java.io.IOException ("File I/O Access Error"));
} catch(SQLException se) {
while(se != null) {
System.out.println("SQL State:" + se.getSQLState());
System.out.println("Error Code:" + se.getErrorCode());
System.out.println("Message:" + se.getMessage());
Throwable t = se.getCause();
while(t != null){
System.out.println("Cause:" + t);
t = t.getCause();
} }
se = se.getNextException();
}}
…
```

Concrete Sub Classes for Specific SQL Exception

Transient Exception

- `SQLTransientException` class

Recoverable Exception

- `SQLRecoverableException` class

Non-Transient Exception

- `SQLNonTransientException` class

Transient Exception

- Following are the new exception subclasses of `SQLTransientException`:

Exception Class	Description
`SQLTransientConnectionException`	Thrown when a previously failed connection operation can be retried without any changes in code of an application.
`SQLTransactionRollbackException`	Thrown when the current operation is automatically rolled back because of deadlock or transaction serialization failure.
`SQLTimeoutException`	Thrown when a timeout specified by a Statement interface expires.

Non-Transient Exception

- Following are the new exception classes that have been added in JDBC 4.0 as subclasses of `SQLNonTransientException` class.

Exception Class	Description
`SQLFeatureNotSupportedException`	Thrown when JDBC optional features, such as overloaded methods are not supported by JDBC driver.
`SQLNonTransientConnectionException`	Thrown when a connection operation fails.
`SQLDataException`	Thrown due to data errors, such as divide by 0, invalid arguments to functions.
`SQLIntegrityConstraintViolationException`	Thrown due to violation of integrity constraint, such as primary key.
`SQLSyntaxErrorException`	Thrown when an SQL query has violated an SQL syntax.

RowId Interface 1-3

`RowId` data type defined which uniquely identifies each row in a database table.

It is an unique identifier for row of a table.

`getRowId()` and `setRowId()` methods.

`RowId` can be used in the absence of a primary key in the database design to uniquely identify certain row.

`RowId` objects are specific to data sources and are not portable.

- Following table displays the values that a `RowId` object can have:

RowId	Description
ROWID_UNSUPPORTED	It does not support ROWID data type.
ROWID_VALID_OTHER	Lifetime of the ROWID depends on database vendor implementation.
ROWID_VALID_TRANSACTION	Thrown due to data errors, such as divide by 0, invalid arguments to functions.
ROWID_VALID_SESSION	Thrown due to violation of integrity constraint, such as primary key.
ROWID_VALID_FOREVER	Thrown when an SQL query has violated an SQL syntax.

RowId Interface 2-3

The following Code Snippet shows the usage of `RowId` object.

Code Snippet:

```
...
ResultSet rs = stmt.executeQuery("Select name, designation,
RowId from EmployeeDetails");
while(rs.next()) {
String name = getString(1);
String designation = getString(2);
RowId rowid = getRowId(3);
...
}
...
```

RowId Interface 3-3

- The following Code Snippet shows the usage of `getRowIdLifetime()` method.

Code Snippet:

```
...
RowIdLifeTime lifeTime =
DatabaseMetaData.getRowIdLifetime();
if (lifeTime != ROWID_UNSUPPORTED){
// Row id is supported by the database
}
...
```

Enhancements in Interfaces and Large Objects

JDBC supports following three types of large data objects:

Binary Large Object (BLOB)

- Stores a large amount of byte-oriented data such as images, music, and videos as a column value in a row of a database table.

Character Large Object (CLOB)

- Stores a large amount of character-oriented data.

National Character Large Object (NCLOB)

- Stores a large amount of character-oriented data using the national character set as a column value in a row.

Handling Large Objects

The interfaces which are responsible to support large objects are as follows:

- `PreparedStatement`
- `Connection`
- `ResultSet`
- `Blob`, `Clob`, and `Nclob`

In version 4.0 there are new methods added to these interfaces to handle the large objects more efficiently.

Methods in Connection Interface

- Following table lists methods added to the `Connection` interface in JDBC 4.0:

Method	Description
`Blob createBlob() throws SQLException`	Creates and returns an object with no data present in it and whose class implements the `java.sql.Blob` interface.
`Clob createClob() throws SQLException`	Creates and returns an object with no data present in it and whose class implements the `java.sql.Clob` interface.
`NClob createNClob() throws SQLException`	Creates and returns an object with no data present in it and whose class implements the `java.sql.NClob` interface.

- Following methods are added to the `PreparedStatement` interface.

Method	Description
`void setBlob(int parameterIndex,InputStream inputStream) throws SQLException`	Sets the index of parameter to an `InputStream` object and informs the driver that this parameter value should be sent to the server as an SQL BLOB.
`void setBlob(int parameterIndex, InputStream inputStream, long length) throws SQLException`	Sets the index of parameter to an `InputStream` object containing the specified number of characters in the input stream and informs the driver that this parameter value should be sent to the server as an SQL BLOB.
`void setClob(int parameterIndex, Reader reader) throws SQLException`	Sets the index of the parameter to a `Reader` object and informs the driver that this parameter value should be sent to the server as an SQL BLOB.

Methods in PreparedStatement Interface

- Following methods are added to the `PreparedStatement` interface.

Method	Description
`void setBlob(int parameterIndex,InputStream inputStream) throws SQLException`	Sets the index of parameter to an `InputStream` object and informs the driver that this parameter value should be sent to the server as an SQL BLOB.
`void setBlob(int parameterIndex, InputStream inputStream, long length) throws SQLException`	Sets the index of parameter to an `InputStream` object containing the specified number of characters in the input stream and informs the driver that this parameter value should be sent to the server as an SQL BLOB.
`void setClob(int parameterIndex, Reader reader) throws SQLException`	Sets the index of the parameter to a `Reader` object and informs the driver that this parameter value should be sent to the server as an SQL BLOB.
`void setClob(int parameterIndex, Reader reader, long length) throws SQLException`	Sets the index of the parameter to a `Reader` object containing specified number of characters in the input stream and informs the driver that this parameter value should be sent to the server as an SQL CLOB.
`void setNClob(int parameterIndex, Reader reader) throws SQLException`	Sets the parameter to a `Reader` object and informs the driver that this parameter value should be sent to the server as a SQL NCLOB.
`void setNClob(int parameterIndex, Reader reader, long length) throws SQLException`	Sets the parameter to a `Reader` object with specified number of bytes in the input stream and informs the driver that this parameter value should be sent to the server as an SQL NCLOB.

Methods Added to ResultSet Interface

Update to the large objects is done through `ResultSet` interface objects, following is a set of overloaded methods for Blob objects.

Method	Description
`void updateBlob(int columnIndex, InputStream inputStream) throws SQLException`	Updates the column of BLOB data type based on the column index by reading the data from the input stream as needed.
`void updateBlob(int columnIndex, InputStream inputStream, long length) throws SQLException`	Updates the column of BLOB data type based on the column index by reading the data from the input stream containing the specified number of bytes.
`void updateBlob(String columnLabel, InputStream inputStream) throws SQLException`	Updates the column of BLOB data type based on the column name by reading the data from the input stream as needed.
`void updateBlob(String columnLabel, InputStream inputstream, long length) throws SQLException`	Updates the column of BLOB data type based on the column name by reading the data from the input stream containing the specified number of bytes.

Similarly, methods are added for updation of `Clob` and `Nclob`.

Methods in Blob, Clob, and Nclob Interfaces 1-2

Methods to retrieve partial large objects into a stream are added to `Blob`, `Clob`, and `NClob` interfaces.

`Blob`

- `InputStream getBinaryStream(long pos,long length)`

`Clob` and `NClob`

- `Reader getCharacterStream(long pos, long length)`

Methods in Blob, Clob, and Nclob Interfaces 2-2

- The following Code Snippet shows how to use the `createBlob()` and `free()` methods:

Code Snippet:

```
...
Connection con = getConnection(); // User-defined method
PreparedStatement ps = con.prepareStatement("INSERT INTO
Student(NAME,photograph) VALUES (?, ?)");
ps.setString(1, "Martin");
Blob blob = con.createBlob();
// Serialize an ImageIcon with martin.png image to blob
...
ps.setBlob(2, blob);
ps.execute();
blob.free();
ps.close();
...
```

Enhancements to Other Interfaces 1-3

Apart from those mentioned earlier, methods were added to the following interfaces:

Array

- Provides methods to store `SQL ARRAY` values data to the application in either an array or a `ResultSet` object.
- The `free()` method has been `added`, `releases` an array object and the resources it occupies from the memory.

DataSet

- Several new methods in the `Connection` and `ResultSet` interfaces such as `createSQLXML()`, `isvalid()`, and so on.

DatabaseMetadata

- Methods such as `getSchemas()` for querying the database metadata information.

Enhancements to Other Interfaces 2-3

- The following Code Snippet shows the usage of `getSchemas()` method.

Code Snippet:

```
...
Connection cn = getConnection();
DatabaseMetaData dmd = cn.getMetaData();
ResultSet rsDetails = dmd.getSchemas("PolarBank", null);
while(rsDetails.next()) {
System.out.print("" + rsDetails.getString(1) + "\t");
System.out.println("" + rsDetails.getString(2));
}
...
```

Enhancements to Other Interfaces 3-3

- Following figure shows the output:

```
Output - DBS (run)
run:
db_accessadmin          null
db_backupoperator          null
db_datareader          null
db_datawriter          null
db_ddladmin          null
db_denydatareader          null
db_denydatawriter          null
db_owner          null
db_securityadmin          null
public          null
dbo          dbo
guest          guest
BUILD SUCCESSFUL (total time: 4 seconds)
```

Scalar Functions

- A scalar function operates on input values and returns the result.
- Following table lists the scalar functions are added in JDBC 4.0/4.1:

Method	Description
CHAR_LENGTH(string)	Returns the length in characters of the string expression, if it is of character data type. If the expression is not a character data type, the function returns its length in bytes such that the length is the smallest integer not less than the number of bits divided by 8. This function is synonym for CHARACTER_LENGTH(string).
CURRENT_DATE()	Returns the current date. This function is synonym for CURDATE().
CURRENT_TIME()	Returns the current time. This function is a synonym for CURTIME().
CURRENT_TIMESTAMP()	Returns the current date and time. This function is synonym for NOW().
EXTRACT(field FROM source)	Returns the year, month, day, hour, minute, or second field from the date-time source.
OCTET_LENGTH(string)	Returns the length in bytes of the string expression such that the length is the smallest integer not less than the number of bits divided by 8.
POSITION (substring IN string)	Returns the position of first substring occurrence in string as a NUMERIC. The precision is implementation-defined, and the scale is zero.

Using Statement try-with Resources 1-3

Automatic Resource Management (ARM) introduced in JDBC 4.1, to ensure that all the resources are closed after the purpose is served.

Resource management was done through `finally` block in earlier versions of JDBC.

`try` statement allows declaration of more than one resources.

Suppressed exceptions are retrieved through method `Throwable.getSuppressed();`

Using Statement try-with Resources 2-3

- The following Code Snippet shows the usage of a try-with statement.

Code Snippet:

```
static String readFile(String path) throws IOException {
try (BufferedReader b = new BufferedReader(new
FileReader(path))) {
return b.readLine();
}
}
```

- The following Code Snippet shows a code which has two resources declared with the `try` statement.

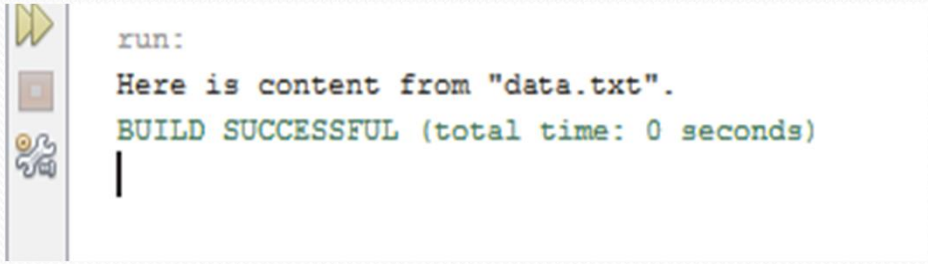
Code Snippet:

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
public class trywith {
public static void main(String[] args) {
```

Using Statement try-with Resources 3-3

```
try (BufferedReader br = new BufferedReader(new
FileReader("C:\\Users\\ Documents\\Database programming with
Java\\data.txt")))
{
String line;
while ((line = br.readLine()) != null) {
System.out.println(line);
}
} catch (IOException e) {
e.printStackTrace();
}
}
}
```

- Following figure shows the output of the program:



RowSet Interface and RowSetProvider Class

All the SQL queries return a set of rows as a result of SQL query. RowSet interfaces are created to store these results.

Following classes are defined to handle multiple queries and their results:

- RowSetProvider
- RowSetFactory

- The following Code Snippet demonstrates how a `RowSetProvider` API can be used to implement different `RowSets`:.

Code Snippet:

```
RowSetFactory rsf = RowSetProvider.newFactory();
CachedRowSet crs = a.createCachedRowSet();
...
RowSetFactory rsf1 =
RowSetProvider.newFactory("com.sun.rowset.
RowSetFactoryImpl", null);
WebRowSet w = r.createWebRowSet();
...
```

RowSet Implementations

Following are the various `RowSet` implementations to suit the requirements of different applications.

- `JdbcRowSet`
- `CachedRowSet`
- `WebRowSet`
- `JoinRowSet`
- `FilteredRowSet`

JoinRowSet 1-5

- Used to perform join of multiple RowSets in the database. Following syntax shows how a `JoinRowSet` object can be created:

Syntax:

```
JoinRowSet jrs = new JoinRowSetImpl();
```

- The `addRowSet(rowset, join column)` method used to specify the RowSet on which the join operation has to be performed and based on which column.

Syntax:

```
jrs.addRowSet(tablename, column name);
```

- Consider the following query:

```
SELECT EMPLOYEE.FIRST_NAME " + "FROM EMPLOYEE, DEPARTMENT " +
"WHERE DEPARTMENT.DNAME = RESEARCH " + "and " +
"DEPARTMENT.DNUMBER = EMPLOYEE.DID";
```

JoinRowSet 2-5

- The following Code Snippet implements this SQL query by using `JoinRowSet` objects, given an **Employee** database with **Employee** and **Department** tables.

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;
import javax.sql.RowSet.*;
import javax.sql.rowset.JoinRowSet;
import javax.sql.rowset.CachedRowSet;
import com.sun.rowset.CachedRowSetImpl;
import com.sun.rowset.JoinRowSetImpl;
import java.util.Hashtable;
public class Databasetrials {
public static void main(String[] args) {
try{
String driver = "com.microsoft.sqlserver.jdbc.SQLServerDriver";
```

JoinRowSet 3-5

```
String url = "jdbc:sqlserver://localhost:1433;
  databaseName=company";

String username = "root";
String password = "*****";
Class.forName(driver).newInstance();
Connection conn = DriverManager.getConnection(url,
  username,password);
System.out.println("Connection Done");
Statement statement = conn.createStatement();
CachedRowSet employee = new CachedRowSetImpl();
String query = "select * from Employee";
employee.setCommand(query);
employee.execute(conn);
while (employee.next()) {
// Generating cursor Moved event
System.out.println("Emp_id- " + employee.getString(1)+" name- " +
  employee.getString(2)+" Did- " + employee.getString(6));
}
```

JoinRowSet 4-5

```
Statement statement2 = conn.createStatement();
CachedRowSet department= new CachedRowSetImpl();
String quer = "select * from Department";
department.setCommand(quer);
department.execute(conn);
while (department.next()) {
// Generating cursor Moved event
System.out.println("Dep_id- " + department.getString(1)+" name- "
  + department.getString(2)+" Manager- " +
  department.getString(3));
}
JoinRowSet jrs = new JoinRowSetImpl();
jrs.addRowSet(employee, "Did");
jrs.addRowSet(department, "Dnumber");
String depname = "Research";
System.out.println("Employees working in " + depname + ": ");
while (jrs.next()) {
if (jrs.getString("Dname").equals(depname)) {
String Emp_name = jrs.getString(2);
System.out.println(" " + Emp_name);
```

JoinRowSet 5-5

```
   }
  }
}catch(SQLException |ClassNotFoundException
   |InstantiationException|Ille galAccessException i){
System.out.println(i+" Exception reported");
  }
}  }
```

- Following figure shows the output of the program:

```
run:
Connection Done
Emp_id- 123456 name- John Did- 5
Emp_id- 333445 name- Franklin Did- 5
Emp_id- 345678 name- Ramesh Did- 5
Emp_id- 357444 name- Joyce Did- 5
Emp_id- 454555 name- Ahmad Did- 4
Emp_id- 777889 name- James Did- 1
Emp_id- 987654 name- Jennifer Did- 4
Emp_id- 999888 name- Alicia Did- 4
Dep_id- 1 name- Headquarters Manager- 777889
Dep_id- 4 name- Administration Manager- 454555
Dep_id- 5 name- Research Manager- 333445
Employees working in Research:
     Joyce
     Ramesh
     Franklin
     John
BUILD SUCCESSFUL (total time: 0 seconds)
```

WebRowSet 1-5

- `WebRowSet` enables the conversion of data in tabular form in the database to xml.

Syntax:

```
WebRowSet priceList = new WebRowSetImpl();
```

- `readXml()` and `writeXml()` methods are used to read XML documents into database and write the output of a query into XML form respectively.
- The following Code Snippet shows the usage of `writeXml()` method:

```
import com.sun.rowset.WebRowSetImpl;
import javax.sql.rowset.WebRowSet;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;
import java.io.FileWriter;
import java.io.IOException;
```

WebRowSet 2-5

```
public class WebSet {
public static void main(String[] args) throws IOException {
try{
String driver = "
com.microsoft.sqlserver.jdbc.SQLServerDriver";
String url = "
jdbc:sqlserver://localhost:1433;databaseName=comp any";
String username = "root";
String password = "*****";
Class.forName(driver).newInstance();
Connection conn = DriverManager.getConnection(url, username,
password);
System.out.println("Connection Done");
WebRowSet employee = new WebRowSetImpl();
String query = "select * from Employee";
employee.setCommand(query);
employee.execute(conn);
employee.first();
FileWriter f = new FileWriter("emp.xml");
employee.writeXml(f);
employee.first();
```

WebRowSet 3-5

```
while (employee.next()) {
// Generating cursor Moved event
System.out.println("Emp_id- " + employee.getString(1)+" name- " +
  employee.getString(2)+" Did- " + employee.getString(6));
employee.getString(2)+" Did- " + employee.getString(6));
}
}
catch(SQLException |ClassNotFoundException
  |InstantiationException|Ill egalAccessException|IOException i)
{
System.out.println(i+" Exception reported");
}
}
}
```

WebRowSet 4-5

- Following figure shows the output of the given program:

```
run:
Connection Done
Emp_id- 123456 name- John Did- 5
Emp_id- 333445 name- Franklin Did- 5
Emp_id- 345678 name- Ramesh Did- 5
Emp_id- 357444 name- Joyce Did- 5
Emp_id- 454555 name- Ahmad Did- 4
Emp_id- 777889 name- James Did- 1
Emp_id- 987654 name- Jennifer Did- 4
Emp_id- 999888 name- Alicia Did- 4
BUILD SUCCESSFUL (total time: 2 seconds)
```

WebRowSet 5-5

- Following figure shows the **emp.xml** file to which the `WebRowSet` is written:

File Edit Format View Help


```
<?xml version="1.0"?><webRowSet xmlns="http://java.sun.com/xml/ns/jdbc"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"xsi:schemaLocation="http://java.sun.com/xml/ns/jdbc
http://java.sun.com/xml/ns/jdbc/webrowset.xsd">  <properties>    <command>select * from Employee</command>
<concurrency>1008</concurrency>    <datasource><null/></datasource>    <escape-processing>true</escape-
processing>    <fetch-direction>1000</fetch-direction>    <fetch-size>0</fetch-size>    <isolation-
level>2</isolation-level>    <key-columns>    </key-columns>    <map>    </map>    <max-field-size>0</max-
field-size>    <max-rows>0</max-rows>    <query-timeout>0</query-timeout>    <read-only>true</read-only>
<rowset-type>ResultSet.TYPE_SCROLL_INSENSITIVE</rowset-type>    <show-deleted>false</show-deleted>
<table-name>Employee</table-name>    <url><null/></url>    <sync-provider>      <sync-provider-
name>com.sun.rowset.providers.RIOptimisticProvider</sync-provider-name>      <sync-provider-vendor>Oracle
Corporation</sync-provider-vendor>      <sync-provider-version>1.0</sync-provider-version>      <sync-
provider-grade>2</sync-provider-grade>      <data-source-lock>1</data-source-lock>    </sync-provider>
</properties>  <metadata>    <column-count>6</column-count>    <column-definition>      <column-
index>1</column-index>      <auto-increment>false</auto-increment>      <case-sensitive>false</case-
sensitive>      <currency>false</currency>      <nullable>0</nullable>      <signed>true</signed>
<searchable>true</searchable>      <column-display-size>10</column-display-size>      <column-
label>Emp_id</column-label>      <column-name>Emp_id</column-name>      <schema-name></schema-name>
<column-precision>10</column-precision>      <column-scale>0</column-scale>      <table-
name>Employee</table-name>      <catalog-name>company</catalog-name>      <column-type>4</column-type>
<column-type-name>INT</column-type-name>    </column-definition>    <column-definition>      <column-
index>2</column-index>      <auto-increment>false</auto-increment>      <case-sensitive>false</case-
sensitive>      <currency>false</currency>      <nullable>1</nullable>      <signed>false</signed>
<searchable>true</searchable>      <column-display-size>40</column-display-size>      <column-
label>First_name</column-label>      <column-name>First_name</column-name>      <schema-name></schema-name>
```

FilteredRowSet 1-6

- `FilteredRowSet` object retrieves a subset of rows from the input RowSet based on a filtering condition.

Syntax:

```
FilteredRowSet frs = new FilteredRowSetImpl();
```

- The filtering condition is defined through `Predicate` interface.
- The following Code Snippet shows a `Predicate` class defined for the `FilteredRowSet` object:

Code Snippet:

```
import java.sql.SQLException;
import javax.sql.rowset.CachedRowSet;
import javax.sql.RowSet;
import javax.sql.rowset.Predicate;
public class Filter implements Predicate {
private int lo;
private int hi;
```

FilteredRowSet 2-6

```
private String colName = null;
private int colNumber = -1;
public Filter(int lo, int hi, int colNumber) {
this.lo = lo;
this.hi = hi;
this.colNumber = colNumber;
}
public Filter(int lo, int hi, String colName) {
this.lo = lo;
this.hi = hi;
this.colName = colName;
}
public boolean evaluate(Object value, String columnName) {
boolean evaluation = true;
if (columnName.equalsIgnoreCase(this.colName)) {
int columnValue = ((Integer)value).intValue();
if ((columnValue >= this.lo) &&
(columnValue <= this.hi)) {
evaluation = true;
} else {
evaluation = false;
}
```

FilteredRowSet 3-6

```
}
return evaluation;
}
public Filter(int lo, int hi, String colName) {
this.lo = lo;
this.hi = hi;
this.colName = colName;
}
public boolean evaluate(Object value, String columnName) {
boolean evaluation = true;
if (columnName.equalsIgnoreCase(this.colName)) {
int columnValue = ((Integer)value).intValue();
if ((columnValue >= this.lo)&& (columnValue <= this.hi)) {
evaluation = true;
} else {
evaluation = false;
}
 }
return evaluation;
}
```

FilteredRowSet 4-6

- The following Code Snippet demonstrates the usage of `FilteredRowSet` object:

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import javax.sql.rowset.FilteredRowSet;
import com.sun.rowset.FilteredRowSetImpl;
import java.sql.Driver;
public class TestFilteredRowSet {
public static void main(String[] args) {
try {
String driver = "com.microsoft.sqlserver.jdbc.SQLServerDriver";
String url = "
jdbc:sqlserver://localhost:1433;databaseName=company ";
String username = "root";
String password = "*****";
Class.forName(driver).newInstance();
Connection conn = DriverManager.getConnection(url,
username,password);
System.out.println("Connection Done");
FilteredRowSet frs = new FilteredRowSetImpl();
```

FilteredRowSet 5-6

```
String query = "select * from Employee";
frs.setCommand(query);
frs.execute(conn);
Filter myFilter = new Filter(10000, 50000, 4);
System.out.println("\nBefore filter:");
while (frs.next()) {
System.out.println("Emp_id- " + frs.getString(1)+" name- " + frs.
  getString(2)+" Salary- " + frs.getString(4));
}
System.out.println("\nAfter filter:");
frs.beforeFirst();
frs.setFilter(myFilter);

while (frs.next()) {
System.out.println("Emp_id- " + frs.getString(1)+" name- " + frs.
  getString(2)+" Salary- " + frs.getString(4));
}
} catch (SQLException e) {
e.getMessage();
}
}
}
```

FilteredRowSet 6-6

- Following figure shows the `FilteredRowSet` program output:

```
Output - Databasetrials (run)
run:
Connection Done
Before Filter
Emp_id- 123456 name- John salary- 30000
Emp_id- 333445 name- Franklin salary- 40000
Emp_id- 345678 name- Ramesh salary- 38000
Emp_id- 357444 name- Joyce salary- 25000
Emp_id- 454555 name- Ahmad salary- 25000
Emp_id- 777889 name- James salary- 55000
Emp_id- 987654 name- Jennifer salary- 43000
Emp_id- 999888 name- Alicia salary- 25000
After Filter
Emp_id- 123456 name- John salary- 30000
Emp_id- 333445 name- Franklin salary- 40000
Emp_id- 345678 name- Ramesh salary- 38000
Emp_id- 357444 name- Joyce salary- 25000
Emp_id- 454555 name- Ahmad salary- 25000
Emp_id- 987654 name- Jennifer salary- 43000
Emp_id- 999888 name- Alicia salary- 25000
BUILD SUCCESSFUL (total time: 1 second)
```

Summary

- A Java program in JDBC version 4.0 need not load a database driver explicitly using the Class. forName() method.
- The RowId interface has been introduced in JDBC 4.0 for supporting the ROWID data type. However, this interface is not supported across all databases.
- NCLOB is a new data type introduced in JDBC 4.0 that stores a large amount of character-oriented data using the National Character Set (NCS) as a column value in a row of a database table.
- JDBC 4.0 supports numeric, string, date/time, conversion, and system functions that operate on scalar values.
- try-with resources statement has been introduced in this module which is important for Automatic Resource Management (ARM).
- JDBC 4.1 has RowSetFactory and RowSetProvider interfaces through which various RowSets can be created.
- JDBC 4.1 has introduced various interfaces to efficiently handle disconnected RowSets. These interfaces extend all the features of CachedRowSet and also add additional methods for handling WebRowSet, FilteredRowSet, and JoinRowSet.