JDBC Overview

"Code with Passion!"

Topics

- What is JDBC?
- Database URL
- Step By Step Usage of JDBC API
- PreparedStatement
- CallableStatement
- Transaction
- Transaction Isolation levels
- DataSource

What is JDBC?

What is JDBC?

- Standard Java API for accessing relational database
  - Hides database specific details from application
- Part of Java SE (J2SE)
  - Java SE 6+ has JDBC 4

JDBC API

- Defines a set of Java Interfaces, which are implemented by vendor-specific JDBC client drivers
  - Applications use this set of Java interfaces for performing database operations – portability is guaranteed
- Majority of JDBC API is located in java.sql package
  - DriverManager, Connection, ResultSet, DatabaseMetaData, ResultSetMetaData, PreparedStatement, CallableStatement and Types
- Other advanced functionality exists in the javax.sql package
  - DataSource

What is JDBC Driver?

- Database specific implemention of JDBC interfaces
  - Each database has its own JDBC driver implementing the same JDBC interfaces
- Example JDBC drivers
  - mysql-connector-java-5.1.34.jar (for MySQL)
  - derbyclient-10.8.1.2.jar (for JavaDB)

Database URL

Database URL

- Used to make a connection to the database
- Can contain server, port, protocol etc
- Format
  - jdbc:[subprotocol]:[server-location]/[database-name]
- Examples
  - jdbc:mysql://localhost:3306/mydatabase
  - jdbc:derby://localhost:1527/sample

Step by Step Usage of JDBC API

Steps of Using JDBC

1. Load database-specific JDBC driver
2. Get a Connection object
3. Get a Statement object
4. Execute queries and/or updates
5. Read results
6. Read Meta-data (optional step)
7. Close Statement and Connection objects

1. Load DB-Specific Database Driver

- To manually load the database driver and register it with the DriverManager, load its class file: Class.forName(<database-driver>)

```
try {
    // This loads an instance of the MySQL Driver.
    // The driver has to be in the classpath.
    Class.forName("com.mysql.jdbc.Driver");
 } catch (ClassNotFoundException cnfe){
     System.out.println("" + cnfe);
}
```

2. Get a Connection Object

- DriverManager class is responsible for selecting the database and and creating the database connection
- Create the database connection as follows:

```
try {
   Connection connection =
       DriverManager.getConnection("jdbc:mysql://localhost:3306/mydb",
                                   “username", "password");
 } catch(SQLException sqle) {
   System.out.println("" + sqle);
 }
```

DriverManager & Connection classes

- java.sql.DriverManager
  - getConnection(String url, String user, String password) throws SQLException

- java.sql.Connection
  - Statement createStatement() throws SQLException
  - void close() throws SQLException
  - void setAutoCommit(boolean b) throws SQLException
  - void commit() throws SQLException
  - void rollback() throws SQLException

3. Get a Statement Object

- Create a Statement Object from Connection object
  - *Statement statement = connection.createStatement();*
- The Statement object is used for executing a query statement

4. Executing Query or Update

- Using the Statement object, you can do SELECT/UPDATE/DELETE

- SELECT
  - *ResultSet rs = statement.executeQuery("select * from customer_tbl");*

- UPDATE
  - *int iReturnValue = statement.executeUpdate("update manufacture_tbl set name = ABC where mfr_num = 19985678");*

5. Reading Results

- Once you have *ResultSet*, you can easily access the data by looping through it
- The iterator is initialized to a position before the first row so you must call *next()* once to move it to the first row

```
while (rs.next()){
  // Wrong this will generate an error since column index starts from 1
  String value0 = rs.getString(0);

  // Correct!
  String value1 = rs.getString(1);
  int    value2 = rs.getInt(2);
  int    value3 = rs.getInt("ADDR_LN1");
}
```

5. Reading Results (Continued)

- When retrieving data from the *ResultSet*, use the appropriate getXXX() method
- There is an appropriate getXXX() method of each java.sql.Types datatype
  - getString()
  - getInt()
  - getDouble()
  - getObject()
  - ...

6.a Read ResultSet MetaData (Optional)

- Once you have the *ResultSet*, you can obtain the meta data about the query
  - *ResultSetMetaData rsMeta = rs.getMetaData();*
- From the ResultSetMetaData, you get meta data such as
  - Column count
  - Column name and type

ResultSetMetaData Example

```
ResultSetMetaData meta = rs.getMetaData();
// Get the column count
int iColumnCount = meta.getColumnCount();

for (int i =1 ; i <= iColumnCount ; i++){
  System.out.println(“Column Name: " + meta.getColumnName(i));
  System.out.println(“Column Type" + meta.getColumnType(i));
  System.out.println("Display Size: " + meta.getColumnDisplaySize(i) );
  System.out.println("Precision: " + meta.getPrecision(i));
  System.out.println(“Scale: " +  meta.getScale(i) );
}
```

6.b Read DataBaseMetaData (Optional)

- Once you have the *Connection* object, you can obtain the Meta Data about the database
  - *DatabaseMetaData dbmetadata = connection.getMetaData();*
- There are approximately 150 methods in the *DatabaseMetaData* class.

PreparedStatement

What is PreparedStatement?

- Represents a precompiled SQL statement.
- A SQL statement is precompiled and stored in a *PreparedStatement* object
- This object can then be used to efficiently execute this statement multiple times.

Example: PreparedStatement

```
PreparedStatement pstmt =
        con.prepareStatement("UPDATE EMPLOYEES
                              SET SALARY = ? WHERE ID = ?");
pstmt.setBigDecimal(1, 100000.00)
pstmt.setInt(2, 110592)
pstmt.executeUpdate();

pstmt.setBigDecimal(1, 200000.00)
pstmt.setInt(2, 222222)
pstmt.executeUpdate();
```

CallableStatment

CallableStatement

- Used to execute SQL stored procedures

CallableStatement cont.

- A *CallableStatement* object contains a call to a stored procedure; it does not contain the stored procedure itself.
- The first line of code below creates a call to the stored procedure SHOW_SUPPLIERS using the connection con .
- The part that is enclosed in curly braces is the escape syntax for stored procedures.

```
CallableStatement cs = con.prepareCall("{call SHOW_SUPPLIERS}");
ResultSet rs = cs.executeQuery();
```

CallableStatement Example

Here is an example using IN, OUT and INOUT parameters

```
// set int IN parameter
cstmt.setInt( 1, 333 );
// register int OUT parameter
cstmt.registerOutParameter( 2, Types.INTEGER );
// set int INOUT parameter
cstmt.setInt( 3, 666 );
// register int INOUT parameter
cstmt.registerOutParameter( 3, Types.INTEGER );
//You then execute the statement with no return value
cstmt.execute(); // could use executeUpdate()
// get int OUT and INOUT
int iOUT = cstmt.getInt( 2 );
int iINOUT = cstmt.getInt( 3 );
```

Stored Procedure example

```
FUNCTION event_list (appl_id_in    VARCHAR2,
                     dow_in        VARCHAR2,
                     event_type_in VARCHAR2 OUT,
                     status_in     VARCHAR2 INOUT)
 RETURN ref_cur;
```

Lab:

Exercise 1: Single Table JDBC
Exercise 2: Joined Table JDBC
Exercise 3: Advanced JDBC
1020_jdbc_overview.zip

Transaction

Why use Transaction?

- Committing of each statement, which is a default behavior, could be very time consuming
  - This is the default behavior – *AutoCommit* is set to true as a default
- By setting *AutoCommit* to *false*, the developer can update the database more then once without commiting and then commit the entire transaction as a whole
- Also, if each statement is dependant on the other, the entire transaction can be rolled back and the user notified

JDBC Transaction Methods

- setAutoCommit()
  - If set true, every executed statement is committed immediately
- commit()
  - Relevant only if setAutoCommit(false)
  - Commit operations performed since the opening of a Connection or last commit() or rollback() calls
- rollback()
  - Relevant only if setAutoCommit(false)
  - Cancels all operations performed

Transactions Example

```
String url = "jdbc:mysql://localhost/testdb";
String username = "root";
String password = "";
Class.forName("com.mysql.jdbc.Driver");
Connection conn = null;
try {
  conn = DriverManager.getConnection(url, username, password);
  conn.setAutoCommit(false);

  Statement st = conn.createStatement();
  st.execute("INSERT INTO orders (username, order_date) VALUES ('java', '2007-12-13')",
      Statement.RETURN_GENERATED_KEYS);

  ResultSet keys = st.getGeneratedKeys();
  int id = 1;
  while (keys.next()) {
    id = keys.getInt(1);
  }

 // To be continued
```

Transactions Example

```
  PreparedStatement pst = conn.prepareStatement("INSERT INTO order_details (order_id, product_id,
   quantity, price) VALUES (?, ?, ?, ?)");
  pst.setInt(1, id);
  pst.setString(2, "1");
  pst.setInt(3, 10);
  pst.setDouble(4, 100);
  pst.execute();

  conn.commit();
  System.out.println("Transaction commit...");
} catch (SQLException e) {
  if (conn != null) {
    conn.rollback();
    System.out.println("Connection rollback...");
  }
  e.printStackTrace();
} finally {
  if (conn != null && !conn.isClosed()) {
    conn.close();
  }
}
```

Lab:

Exercise 4: Transaction
1020_jdbc_overview.zip

Transaction Isolation Levels

4 Isolation Levels

- Level 1 - READ UNCOMMITTED
  - Provides lowest level of isolation among transactions but best performing
- Level 2 - READ COMMITTED
- Level 3 - REPEATABLE READ
- Level 4 - SERIALIZABLE
  - Provides the highest level of isolation among transactions but least performing

Level 1 - READ UNCOMMITTED Isolation level

- Causes 'dirty reads' symptom
  - > Uncommitted changes in one transaction (client #1 below) is visible in other transactions (client #2 below)

Client #1: Start Transaction----INSERT a record A----------ROLLBACK---------->

Client #2: Start Transaction----------(See Record A:Dirty read)----------(Does not see Record A)-->

Level 2 - READ COMMITTED Isolation level

- Committed updates in one transaction (client # 1 below) are visible within another transaction (client #2 below). This means identical queries within a transaction (in client #2 below) can return differing results

Client #1: Start Transaction----INSERT a record A----------COMMIT---------->

Client #2: Start Transaction----------(Not see Record A)---------(See Record A)------COMMIT------>

Level 3 - REPEATABLE READ Isolation level

- Committed changes in one transaction (client #1 below) is visible in another transaction (client #2 below) only after its own Commit.
  - > Within a transaction, all reads are consistent.
- The default isolation level for most databases

Client #1: Start Transaction----INSERT a record A----------COMMIT---------->

Client #2: Start Transaction----------(Not see Record A)----------COMMIT--(See Record A) ---->

Level 4 - SERIALIZABLE Isolation level

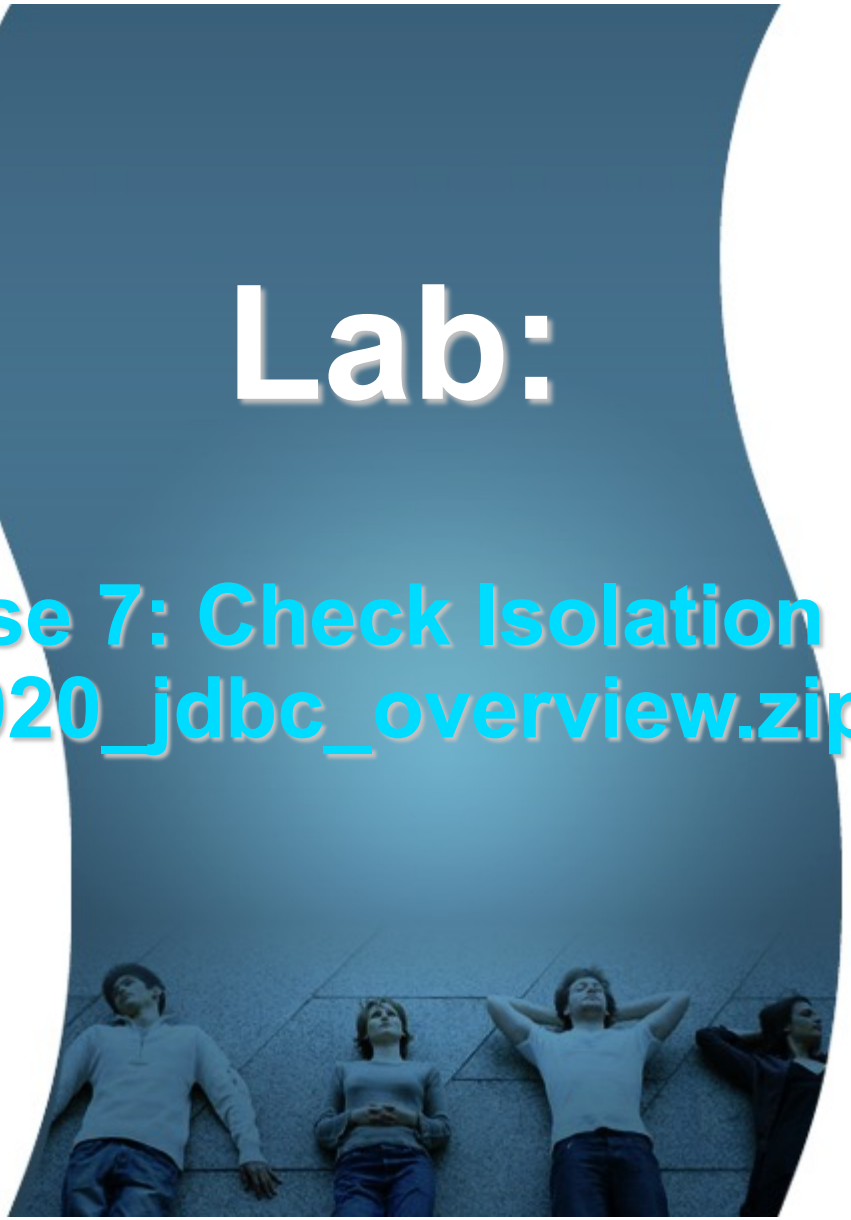
- Transactions are serialized
- Until the previous transaction ends via either COMMIT or ROLLBACK, the database operations in other transactions are blocked
- Highest isolation level but not practical in real-life environment

Client #1: Start Transaction----INSERT a record A----------COMMIT----------->

Client #2: Start Transaction----------(SELECT Blocked)---------- (SELECT returns)--COMMIT---->

Lab:

Exercise 7: Check Isolation Level 1020_jdbc_overview.zip



Code with Passion!