UTD Programming Workshop: Introduction to JDBC

With MySQl

JDBC Introduction

- What is JDBC?

JDBC stands for **J**ava **D**ata**b**ase **C**onnectivity, which is a standard Java API for database-independent connectivity between the Java programming language and a wide range of databases.

The JDBC library includes APIs for each of the tasks commonly associated with database usage:

- Making a connection to a database
- Creating SQL or MySQL statements
- Executing that SQL or MySQL queries in the database
- Viewing & Modifying the resulting records

JDBC Introduction

- JDBC provides a standard library for
- accessing relational databases
- – API standardizes
- • Way to establish connection to database
- • Approach to initiating queries
- • Method to create stored (parameterized) queries
- • The data structure of query result (table)
- – Determining the number of columns
- – Looking up metadata, etc.
- – API does notstandardize SQL syntax
- • You send strings; JDBC is not embedded SQL
- – JDBC classes are in the java.sql package

Getting Ready - Environment

For Windows

1. Download the latest MySQL JDBC driver from http://dev.mysql.com/downloads ⇒ "MySQL Connectors" ⇒ "Connector/J" ⇒ Connector/J 5.1.{xx} ⇒ select "Platform Independent" ⇒ ZIP Archive(e.g., "mysql-connector-java-5.1.{xx}.zip", where {xx} is the latest release number).
2. UNZIP the download file into any temporary folder.
3. Copy "mysql-connector-java-5.1.{xx}-bin.jar" to your JDK's Extension Directory at "<JAVA_HOME>\jre\lib\ext" (e.g., "c:\program files\java\jdk1.7.0_{xx}\jre\lib\ext").

For Mac

1.Download the latest MySQL JDBC driver from http://www.mysql.com/downloads ⇒ MySQL Connectors ⇒ Connector/J ⇒ Connector/J 5.1.{xx} ⇒ select "Platform Independent" ⇒ Compressed TAR Archive (e.g., mysql-connector-java-5.1.{xx}.tar.gz, where {xx} is the latest release number).
2.Double-click on the downloaded TAR file to expand into folder "mysql-connector-java-5.1.{xx}".
3.Open the expanded folder. Copy the JAR file "mysql-connector-java-5.1.{xx}-bin.jar" to JDK's extension directory at "/Library/Java/Extension".

Setting up a Database

- We have to set up a database before embarking on our database programming. We shall call our database "ebookshop" which contains only one table called "books", with 5 columns,.
- To setup the database, start the MySQL server and verify the server's TCP port number from the console messages.

```
// For Windows > cd {path-to-mysql-bin}
// Check your MySQL installed directory
➢  mysqld --console
// For Mac OS X
$ cd /usr/local/mysql/bin
$ sudo ./mysqld_safe --console
```

What is JDBC Driver ? And Types

- JDBC drivers implement the defined interfaces in the JDBC API for interacting with your database server.

Types:

Type 1: JDBC-ODBC Bridge Driver:

Type 2: JDBC-Native API:

Type 3: JDBC-Net pure Java:

Type 4: 100% pure Java:

Seven Basic Steps in Using JDBC

1. Load the driver
   – Not required in Java >6, so Java >6 needs only 6 steps
2. Define the Connection URL
3. Establish the Connection
4. Create a Statement object
5. Execute a query
6. Process the results
7. Close the connection

Connecting The DB

- **Import JDBC Packages:** Add **import** statements to your Java program to import required classes in your Java code

```
import java.sql.* ; // for standard JDBC programs
import java.math.* ; // for BigDecimal and BigInteger support
```

- **Register JDBC Driver:** This step causes the JVM to load the desired driver implementation into memory so it can fulfill your JDBC requests.

  i. The most common approach to register a driver is to use Java's **Class.forName()** method to dynamically load the driver's class file into memory, which automatically registers it. This method is preferable because it allows you to make the driver registration configurable and portable.

```
try {
        Class.forName("oracle.jdbc.driver.OracleDriver");
}
catch(ClassNotFoundException ex) {
        System.out.println("Error: unable to load driver class!");
        System.exit(1);
}
```

  ii. The second approach you can use to register a driver is to use the staticDriverManager.registerDriver() method.You should use the registerDriver() method if you are using a non-JDK compliant JVM, such as the one provided by Microsoft.

Connecting The DB

- After you've loaded the driver, you can establish a connection using the**DriverManager.getConnection()** method. For easy reference, let me list the three overloaded DriverManager.getConnection() methods:
  - getConnection(String url)
  - getConnection(String url, Properties prop)
  - getConnection(String url, String user, String password)

Here each form requires a database URL. A database URL is an address that points to your database.

Formulating a database URL is where most of the problems associated with establishing a connection occur.

RDBMS	JDBC driver name	URL format
MySQL	com.mysql.jdbc.Driver	**jdbc:mysql://**hostname/ databaseName

If you have a host at TCP/IP address 192.0.0.1 with a host name of amrood, and your Oracle listener is configured to listen on port 1521, and your database name is EMP, then complete database URL would then be:jdbc:oracle:thin:@amrood:1521:EMP

Now you have to call getConnection() method with appropriate username and password to get a**Connection** object as follows:

```
String URL = "jdbc:oracle:thin:@amrood:1521:EMP";
String USER = "username";
String PASS = "password" Connection conn = DriverManager.getConnection(URL,
USER, PASS);
```

SQL: The Queries

Create Database:

```
SQL> CREATE DATABASE DATABASE_NAME;
```

Drop/Del Database:

```
SQL> DROP DATABASE DATABASE_NAME;
```

Create Table:

```
SQL> CREATE TABLE Employees
(
        id INT NOT NULL,
        age INT NOT NULL,
        first VARCHAR(255),
        last VARCHAR(255),
        PRIMARY KEY ( id )
);
```

Drop Table:

```
SQL> DROP TABLE table_name;
```

You can get a more comprehensive list from here:
http://www.w3schools.com/sql/sql_quickref.asp

```
 try {
Class.forName("com.mysql.jdbc.Driver");
} catch (ClassNotFoundException e) {
// TODO Auto-generated catch block
e.printStackTrace();
}
     Connection con;
try {
con =
DriverManager.getConnection("jdbc:mysql://localhost:3306/firstdb","root",“password");

     Statement stmt=con.createStatement();
     String query="SELECT * FROM employee_rec";
     ResultSet rs=stmt.executeQuery(query);
     while(rs.next())
     {
          System.out.println(rs.getString("fname”));
     }
     rs.close();
     stmt.close();
     con.close();
} catch (SQLException e) {
// TODO Auto-generated catch block
e.printStackTrace();
}
```

Statements

- Once a connection is obtained we can interact with the database. The JDBC *Statement, CallableStatement,* and *PreparedStatement* interfaces define the methods and properties that enable you to send SQL or PL/SQL commands and receive data from your database.
- They also define methods that help bridge data type differences between Java and SQL data types used in a database.

Interfaces	Recommended Use
Statement	Use for general-purpose access to your database. Useful when you are using static SQL statements at runtime. The Statement interface cannot accept parameters.
PreparedStatement	Use when you plan to use the SQL statements many times. The PreparedStatement interface accepts input parameters at runtime.
CallableStatement	Use when you want to access database stored procedures. The CallableStatement interface can also accept runtime input parameters.

Statement Objects

Creating Statement Object:

- Before you can use a Statement object to execute a SQL statement, you need to create one using the Connection object's createStatement( ) method, as in the following example:

```
Statement stmt = null;
try {
                stmt = conn.createStatement( ); . . .
}
catch (SQLException e) {
 . . .
}
finally {
 . . .
}
```

Closing Statement Obeject:

- Just as you close a Connection object to save database resources, for the same reason you should also close the Statement object.

```
Statement stmt = null;
try { stmt = conn.createStatement( ); . . . }
catch (SQLException e) { . . . }
finally { stmt.close();
}
```

PreparedStatement Objects

Creating PreparedStatement Object:

```
PreparedStatement pstmt = null;
try {
        String SQL = "Update Employees SET age = ? WHERE id = ?";
        pstmt = conn.prepareStatement(SQL); . . . }
catch (SQLException e) { . . . }
finally { . . . }
```

Closing PreparedStatement Object

```
PreparedStatement pstmt = null;
try {
        String SQL = "Update Employees SET age = ? WHERE id = ?";
        pstmt = conn.prepareStatement(SQL); . . . }
catch (SQLException e) { . . . }
finally { pstmt.close(); }
```

Callable Statement

- Just as a Connection object creates the Statement and PreparedStatement objects, it also creates the CallableStatement object which would be used to execute a call to a database stored procedure.

- **Creating**

```
CallableStatement cstmt = null;
 try {
                String SQL = "{call getEmpName (?, ?)}";
                cstmt = conn.prepareCall (SQL); . . . }
catch (SQLException e) { . . .
}
finally { . . .
}
```

- **Closing**

```
CallableStatement cstmt = null; try { String SQL = "{call getEmpName (?, ?)}"; cstmt =
conn.prepareCall (SQL); . . . } catch (SQLException e) { . . . } finally { cstmt.close(); }
```

Executing SQL - Statements

- A SQL statement can be executed with any one of its three execute methods:
- **boolean execute(String SQL)** : Returns a boolean value of true if a ResultSet object can be retrieved; otherwise, it returns false. Use this method to execute SQL DDL statements or when you need to use truly dynamic SQL.
- **int executeUpdate(String SQL)** : Returns the numbers of rows affected by the execution of the SQL statement. Use this method to execute SQL statements for which you expect to get a number of rows affected - for example, an INSERT, UPDATE, or DELETE statement.
- **ResultSet executeQuery(String SQL)** : Returns a ResultSet object. Use this method when you expect to get a result set, as you would with a SELECT statement.

Process the Result: Resultset

- The SQL statements that read data from a database query return the data in a result set. The SELECT statement is the standard way to select rows from a database and view them in a result set. The *java.sql.ResultSet* interface represents the result set of a database query.

Important ResultSet methods

- – resultSet.next()
- • Goes to the next row. Returns false if no next row.
- – resultSet.getString("columnName")
- • Returns value of column with designated name in current
- row, as a String. Also getInt, getDouble, getBlob, etc.
- – resultSet.getString(columnIndex)
- • Returns value of designated column. First index is 1 (ala
- SQL), not 0 (ala Java).
- – resultSet.beforeFirst()
- • Moves cursor before first row, as it was initially. Also first
- – resultSet.absolute(rowNum)
- • Moves cursor to given row (starting with 1). Also last and
- afterLast.

Commit That Code!: JDBC Transactions

- If your JDBC Connection is in *auto-commit* mode, which it is by default, then every SQL statement is committed to the database upon its completion.
- if you have a Connection object named conn, code the following to turn off auto-commit:

```
conn.setAutoCommit(false);
```

Commit! → `conn.commit( );`

Rollback (I messed up) → `conn.rollback( );`

Example:

```
try{
//Assume a valid connection object conn
                    conn.setAutoCommit(false);
                    Statement stmt = conn.createStatement();
                    String SQL = "INSERT INTO Employees " +
                    "VALUES (106, 20, 'Rita', 'Tez')"; stmt.executeUpdate(SQL);
//Submit a malformed SQL statement that breaks
String SQL = "INSERTED IN Employees " +
                    "VALUES (107, 22, 'Sita', 'Singh')";
                    stmt.executeUpdate(SQL);
                    // If there is no error.
                    conn.commit();
}catch(SQLException se){
// If there is any error.
conn.rollback();
}
```

Stored Procedures

- Just as a Connection object creates the Statement and PreparedStatement objects, it also creates the CallableStatement object which would be used to execute a call to a database stored procedure.

Creating CallableStatement Object:

```
DELIMITER $$
DROP PROCEDURE IF EXISTS `EMP`.`getEmpName` $$
CREATE PROCEDURE `EMP`.`getEmpName` (IN EMP_ID INT, OUT EMP_FIRST VARCHAR(255))
BEGIN
        SELECT first INTO EMP_FIRST
        FROM Employees
        WHERE ID = EMP_ID;
END $$
DELIMITER ;
```

Closing CallableStatement Obeject:

```
CallableStatement cstmt = null;
try {
String SQL = "{call getEmpName (?, ?)}";
cstmt = conn.prepareCall (SQL); . . .
}
catch (SQLException e) {
 . . . }
finally { cstmt.close();
}
```

Stored Procedure Types

- Three types of parameters exist: IN, OUT, and INOUT. The PreparedStatement object only uses the IN parameter. The CallableStatement object can use all three.

Parameter	Description
IN	A parameter whose value is unknown when the SQL statement is created. You bind values to IN parameters with the setXXX() methods.
OUT	A parameter whose value is supplied by the SQL statement it returns. You retrieve values from theOUT parameters with the getXXX() methods.
INOUT	A parameter that provides both input and output values. You bind variables with the setXXX() methods and retrieve values with the getXXX() methods.