Introduction to JDBC

Agenda

• Overview of JDBC technology
• JDBC drivers
• Seven basic steps in using JDBC
• Retrieving data from a ResultSet
• Using prepared and callable statements
• Handling SQL exceptions
• Submitting multiple statements as a transaction
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 not standardize SQL syntax
    - JDBC is not embedded SQL
    - JDBC class located in java.sql package
  - Note: JDBC is not officially an acronym; unofficially, “Java Database Connectivity” is commonly used.

On-line Resources

- Sun’s JDBC Site
- JDBC Tutorial
  - http://java.sun.com/docs/books/tutorial/jdbc/
- List of Available JDBC Drivers
- API for java.sql
Oracle On-line Resources

• Java Center
  – http://technet.oracle.com/tech/java/content.html

• SQLJ & JDBC Basic Samples

• JDBC Drivers
  – Requires free registration

• Certification
  – http://www.oracle.com/education/certification/

JDBC Drivers

• JDBC consists of two parts:
  – JDBC API, a purely Java-based API
  – JDBC Driver Manager, which communicates with vendor-specific drivers that perform the real communication with the database.
    • Point: translation to vendor format is performed on the client
      – No changes needed to server
      – Driver (translator) needed on client
JDBC Data Types

<table>
<thead>
<tr>
<th>JDBC Type</th>
<th>Java Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIT</td>
<td>boolean</td>
</tr>
<tr>
<td>TINYINT</td>
<td>byte</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>short</td>
</tr>
<tr>
<td>INTEGER</td>
<td>int</td>
</tr>
<tr>
<td>BIGINT</td>
<td>long</td>
</tr>
<tr>
<td>REAL</td>
<td>float</td>
</tr>
<tr>
<td>FLOAT</td>
<td>double</td>
</tr>
<tr>
<td>BINARY</td>
<td>byte[]</td>
</tr>
<tr>
<td>VARBINARY</td>
<td></td>
</tr>
<tr>
<td>LONGVARBINARY</td>
<td></td>
</tr>
<tr>
<td>CHAR</td>
<td>String</td>
</tr>
<tr>
<td>VARCHAR</td>
<td></td>
</tr>
<tr>
<td>LONGVARCHAR</td>
<td></td>
</tr>
<tr>
<td>NUMERIC</td>
<td>BigDecimal</td>
</tr>
<tr>
<td>DECIMAL</td>
<td></td>
</tr>
<tr>
<td>DATE</td>
<td>java.sql.Date</td>
</tr>
<tr>
<td>TIME</td>
<td>java.sql.Timestamp</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td></td>
</tr>
<tr>
<td>CLOB</td>
<td>Clob*</td>
</tr>
<tr>
<td>BLOB</td>
<td>Blob*</td>
</tr>
<tr>
<td>ARRAY</td>
<td>Array*</td>
</tr>
<tr>
<td>DISTINCT</td>
<td>mapping of underlying type</td>
</tr>
<tr>
<td>STRUCT</td>
<td>Struct*</td>
</tr>
<tr>
<td>REF</td>
<td>Ref</td>
</tr>
<tr>
<td>JAVA_OBJECT</td>
<td>underlying Java class</td>
</tr>
</tbody>
</table>

*SQL3 data type supported in JDBC 2.0

Seven Basic Steps in Using JDBC

1. Load the driver
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
JDBC: Details of Process

1. Load the driver

```java
try {
    Class.forName("oracle.jdbc.driver.OracleDriver");
    Class.forName("org.gjt.mm.mysql.Driver");
} catch (ClassNotFoundException cnfe) {
    System.out.println("Error loading driver: " + cnfe);
}
```

2. Define the Connection URL

```java
String host = "dbhost.yourcompany.com";
String dbName = "someName";
int port = 1234;
String oracleURL = "jdbc:oracle:thin:@" + host + ":" + port + ":" + dbName;
String mysqlURL = "jdbc:mysql://" + host + ":" + port + "/" + dbName;
```

3. Establish the Connection

```java
String username = "jay_debesee";
String password = "secret";
Connection connection =
    DriverManager.getConnection(oracleURL, username, password);
```

- Optionally, look up information about the database

```java
DatabaseMetaData dbMetaData = connection.getMetaData();
String productName =
    dbMetaData.getDatabaseProductName();
System.out.println("Database: " + productName);
String productVersion =
    dbMetaData.getDatabaseProductVersion();
System.out.println("Version: " + productVersion);
```
4. **Create a Statement**

```java
Statement statement = connection.createStatement();
```

5. **Execute a Query**

```java
String query = "SELECT col1, col2, col3 FROM sometable";
ResultSet resultSet = statement.executeQuery(query);
```

- To modify the database, use `executeUpdate`, supplying a string that uses `UPDATE`, `INSERT`, or `DELETE`
- Use `setQueryTimeout` to specify a maximum delay to wait for results

6. **Process the Result**

```java
while(resultSet.next()) {
    System.out.println(resultSet.getString(1) + " "+
                        resultSet.getString(2) + " "+
                        resultSet.getString(3));
}
```

- First column has index 1, not 0
- `ResultSet` provides various `getXxx` methods that take a column index or name and returns the data

7. **Close the Connection**

```java
connection.close();
```

- As opening a connection is expensive, postpone this step if additional database operations are expected
Basic JDBC Example

```java
import java.sql.*;

public class TestDB {
    public static void main(String[] args) {
        // Use driver from Connect SW.
        String driver = "connect.microsoft.MicrosoftDriver";
        try {
            Class.forName(driver);
            String url = "jdbc:ff-microsoft://dbtest.apl.jhu.edu:1433/" + // FastForward
                         "pubs"; // Database name
            String user = "sa", password="";
            Connection connection = DriverManager.getConnection(url, user, password);
            Statement statement = connection.createStatement();
            String query = "SELECT col1, col2, col3 FROM testDB";
            // Execute query and save results.
            ResultSet results = statement.executeQuery(query);
            // Print column names.
            String divider = "-----+------+-----");
            System.out.println("Col1 | Col2 | Col3\n" + divider);
            // Print results
            while(results.next()) {
                System.out.println(pad(results.getString(1), 4) + " | " +
                                  pad(results.getString(2), 4) + " | " +
                                  results.getString(3) + "\n" + divider);
            }
            connection.close();
        } catch(ClassNotFoundException cnfe) {
            System.out.println("No such class: " + driver);
        } catch(SQLException se) {
            System.out.println("SQLException: " + se);
        }
    }
}
```

Basic JDBC Example, cont.

```java
// Print column names.
String divider = "-----+------+-----";
System.out.println("Col1 | Col2 | Col3\n" + divider);
// Print results
while(results.next()) {
    System.out.println
    (pad(results.getString(1), 4) + " | " +
     pad(results.getString(2), 4) + " | " +
     results.getString(3) + "\n" + divider);
}
connection.close();
}
```
Microsoft Access Example

- Northwind sample database

- Northwind.mdb located in C:\Program Files\Microsoft Office\Office\Samples

MS Access Example: Setup

- Create System DSN through ODBC data source
import java.io.*;
import java.sql.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class NorthwindServlet extends HttpServlet {

    public static void main(String[] args) {
        System.out.println(doQuery());
    }

    public void doGet(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
        PrintWriter out = response.getWriter();
        out.println(doQuery());
    }

    public static String doQuery() {
        StringBuffer buffer = new StringBuffer();
        try {
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
            Connection connection =
                    DriverManager.getConnection("jdbc:odbc:Northwind","","");
            Statement statement = connection.createStatement();
            String query = "SELECT FirstName, LastName FROM Employees";
            ResultSet result = statement.executeQuery(query);
            buffer.append("Northwind Database\n\n");
            while (result.next()) {
                buffer.append(result.getString(1) + " "+result.getString(2) + "\n");
            }
            connection.close();
        } catch (ClassNotFoundException cnfe) {
            buffer.append("Couldn't find class file" + cnfe);
        } catch (SQLException sqle) {
            buffer.append("SQL Exception: " + sqle);
        }
        return buffer.toString();
    }
}
Overview
- A ResultSet contains the results of the SQL query
  - Represented by a table with rows and columns
  - In JDBC 1.0 you can only proceed forward through the rows using `next`

Useful Methods
- All methods can throw a SQLException
  - close
    - Releases the JDBC and database resources
    - The result set is automatically closed when the associated Statement object executes a new query
  - getMetaDataObject
    - Returns a ResultSetMetaData object containing information about the columns in the ResultSet
ResultSet (Continued)

• Useful Methods
  – next
    • Attempts to move to the next row in the ResultSet
      – If successful true is returned; otherwise, false
      – The first call to next positions the cursor at the first row
      – Calling next clears the SQLWarning chain
  – getWarnings
    • Returns the first SQLWarning or null if no warnings occurred

ResultSet (Continued)

• Useful Methods
  – findColumn
    • Returns the corresponding integer value corresponding to the specified column name
    • Column numbers in the result set do not necessarily map to the same column numbers in the database
  – getXxx
    • Returns the value from the column specified by column name or column index as an Xxx Java type
    • Returns 0 or null, if the value is a SQL NULL
    • Legal getXxx types:
      double  byte   int   Date    String
      float   short  long  Time   Object
  – wasNull
    • Used to check if the last getXxx read was a SQL NULL
Using MetaData

• **Idea**
  – From a `ResultSet` (the return type of `executeQuery`), derive a `ResultSetMetaData` object
  – Use that object to look up the number, names, and types of columns

• **ResultSetMetaData answers the following questions:**
  – How many columns are in the result set?
  – What is the name of a given column?
  – Are the column names case sensitive?
  – What is the data type of a specific column?
  – What is the maximum character size of a column?
  – Can you search on a given column?

Useful MetaData Methods

• **getColumnCount**
  – Returns the number of columns in the result set

• **getColumnDisplaySize**
  – Returns the maximum width of the specified column in characters

• **getColumnName/getColumnLabel**
  – The `getColumnName` method returns the database name of the column
  – The `getColumnLabel` method returns the suggested column label for printouts

• **getColumnType**
  – Returns the SQL type for the column to compare against types in `java.sql.Types`
Useful MetaData Methods (Continued)

• **isNullable**
  - Indicates whether storing a NULL in the column is legal
  - Compare the return value against `ResultSet` constants: `columnNoNulls`, `columnNullable`, `columnNullableUnknown`

• **isSearchable**
  - Returns `true` or `false` if the column can be used in a `WHERE` clause

• **isReadOnly/isWritable**
  - The `isReadOnly` method indicates if the column is definitely not writable
  - The `isWritable` method indicates whether it is possible for a write to succeed

Using MetaData: Example

```java
Connection connection = DriverManager.getConnection(url, username, password);

// Look up info about the database as a whole.
DatabaseMetaData dbMetaData = connection.getMetaData();
String productName =
  dbMetaData.getDatabaseProductName();
System.out.println("Database: " + productName);
String productVersion =
  dbMetaData.getDatabaseProductVersion();
...
Statement statement = connection.createStatement();
String query = "SELECT * FROM fruits";
ResultSet resultSet = statement.executeQuery(query);
```
Using MetaData: Example

// Look up information about a particular table.
ResultSetMetaData resultsMetaData = 
resultSet.getMetaData();
int columnCount = resultsMetaData.getColumnCount();
// Column index starts at 1 (a la SQL) not 0 (a la Java).
for(int i=1; i<columnCount+1; i++) {
    System.out.print(resultsMetaData.getColumnName(i) + " ");
}
System.out.println();

// Print results.
while(resultSet.next()) {
    // Quarter
    System.out.print(" "+ resultSet.getInt(1));
    // Number of Apples
    ...
}

Using MetaData, Result

Prompt> java cwp.FruitTest dbhost1.apl.jhu.edu PTE hall xxxx oracle

Database: Oracle
Version: Oracle7 Server Release 7.2.3.0.0 - Production Release
PL/SQL Release 2.2.3.0.0 - Production

Comparing Apples and Oranges
=================================
<table>
<thead>
<tr>
<th>QUARTER</th>
<th>APPLES</th>
<th>APPLESALES</th>
<th>ORANGES</th>
<th>ORANGESALES</th>
<th>TOPSELLER</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>32248</td>
<td>$3547.28</td>
<td>18459</td>
<td>$3138.03</td>
<td>Maria</td>
</tr>
<tr>
<td>2</td>
<td>35009</td>
<td>$3850.99</td>
<td>18722</td>
<td>$3182.74</td>
<td>Bob</td>
</tr>
<tr>
<td>3</td>
<td>39393</td>
<td>$4333.23</td>
<td>18999</td>
<td>$3229.83</td>
<td>Joe</td>
</tr>
<tr>
<td>4</td>
<td>42001</td>
<td>$4620.11</td>
<td>19333</td>
<td>$3286.61</td>
<td>Maria</td>
</tr>
</tbody>
</table>
Using Statement

• Overview
  – Through the Statement object, SQL statements are sent to the database.
  – Three types of statement objects are available:
    • Statement
      – for executing a simple SQL statements
    • PreparedStatement
      – for executing a precompiled SQL statement passing in parameters
    • CallableStatement
      – for executing a database stored procedure

Useful Statement Methods

• executeQuery
  – Executes the SQL query and returns the data in a table (ResultSet)
  – The resulting table may be empty but never null
    
    ```java
    ResultSet results = 
    statement.executeQuery("SELECT a, b FROM table");
    ```

• executeUpdate
  – Used to execute for INSERT, UPDATE, or DELETE SQL statements
  – The return is the number of rows that were affected in the database
    – Supports Data Definition Language (DDL) statements CREATE TABLE, DROP TABLE and ALTER TABLE
    
    ```java
    int rows = 
    statement.executeUpdate("DELETE FROM EMPLOYEES" +  
    "WHERE STATUS=0");
    ```
Useful Statement Methods (Continued)

- **execute**
  - Generic method for executing stored procedures and prepared statements
  - Rarely used (for multiple return result sets)
  - The statement execution may or may not return a ResultSet (use statement.getResultSet). If the return value is true, two or more result sets were produced

- **getMaxRows/setMaxRows**
  - Determines the number of rows a ResultSet may contain
  - Unless explicitly set, the number of rows are unlimited (return value of 0)

- **getQueryTimeout/setQueryTimeout**
  - Specifies the amount of time a driver will wait for a STATEMENT to complete before throwing a SQLException

Prepared Statements (Precompiled Queries)

- **Idea**
  - If you are going to execute similar SQL statements multiple times, using “prepared” (parameterized) statements can be more efficient
  - Create a statement in standard form that is sent to the database for compilation before actually being used
  - Each time you use it, you simply replace some of the marked parameters using the setXxx methods

- **As PreparedStatement inherits from Statement the corresponding execute methods have no parameters**
  - execute()
  - executeQuery()
  - executeUpdate()
Prepared Statement, Example

```java
Connection connection =
    DriverManager.getConnection(url, user, password);
PreparedStatement statement =
    connection.prepareStatement("UPDATE employees " +
    "SET salary = ? " +
    "WHERE id = ?");

int[] newSalaries = getSalaries();
int[] employeeIDs = getIDs();
for(int i=0; i<employeeIDs.length; i++) {
    statement.setInt(1, newSalaries[i]);
    statement.setInt(2, employeeIDs[i]);
    statement.executeUpdate();
}
```

Useful Prepared Statement Methods

- **setXxx**
  - Sets the indicated parameter (?) in the SQL statement to the value

- **clearParameters**
  - Clears all set parameter values in the statement

Handling Servlet Data

- Query data obtained from a user through an HTML form may have SQL or special characters that may require escape sequences
- To handle the special characters, pass the string to the `PreparedStatement` `setString` method which will automatically escape the string as necessary
Callable Statements

• **Idea**
  – Permit calls to a stored procedures in a database

• **Advantage**
  – Syntax errors are caught a compile time and not a runtime
  – Stored procedures execute much faster than dynamic SQL
  – The programmer need to know only about the input and output parameters for the stored procedure, not the table structure or internal details of the stored procedure

Callable Statements, cont.

• **Stored Procedure Syntax**
  – Procedure with no parameters
    
    \{ call procedure_name \}

  – Procedure with input parameters
    
    \{ call procedure_name(?, ?, ...) \}

  – Procedure with output parameters
    
    \{ ? = call procedure_name(?, ?, ...) \}

CallableStatement statement =
connection.prepareCall("{ call procedure(?, ?) }");
Callable Statements, cont.

- **Output Parameters**
  - Register the JDBC type of each output parameter through `registerOutParameter` before calling `execute`
    
    ```java
    statement.registerOutParameter(n, Types.FLOAT);
    ```
  - Use `getXxx` to access stored procedure return values

Callable Statements: Example

```java
String procedure = "( ? = call isValidUser(?, ?) )";
CallableStatement statement =
    connection.prepareCall(procedure);
statement.setString(2, username);
statement.setString(3, password);
statement.registerOutParameter(1, Types.BIT);
statement.execute();

if (statement.getBoolean(1)) {
    // Valid Username, password.
    ...}
else {
    // Invalid username, password.
    ...}
```
Useful CallableStatement Methods

• CallableStatement inherits from PreparedStatement

• `getXxx(int parameterIndex)`
  – Retrieves the JDBC output parameter at the specified index as the xxx Java type

• `registerOutputParameter`
  – Binds indexed output parameter to a JDBC type
  – Can also provide a scale parameter to specify the number of digits to the right of the decimal point for NUMERIC or DECIMAL JDBC types

```java
statement.registerOutParameter(2, Types.DECIMAL, 3);
```

Exception Handling

• SQL Exceptions
  – Nearly every JDBC method can throw a SQLException in response to a data access error
  – If more than one error occurs, they are chained together
  – SQL exceptions contain:
    • Description of the error, `getMessage`
    • The SQLState (Open Group SQL specification) identifying the exception, `getSQLState`
    • A vendor-specific integer, error code, `getErrorCode`
    • A chain to the next SQLException, `getNextException`
SQL Exception Example

try {
    ... // JDBC statement.
} catch (SQLException sqle) {
    while (sqle != null) {
        System.out.println("Message: " + sqle.getMessage());
        System.out.println("SQLState: " + sqle.getSQLState());
        System.out.println("Vendor Error: " +
                           sqle.getErrorCode());
        sqle.printStackTrace(System.out);
        sqle = sqle.getNextException();
    }
}

– Don’t make assumptions about the state of a transaction after an exception occurs
– The safest best is to attempt a rollback to return to the initial state

Transactions

• Idea
  – By default, after each SQL statement is executed the changes are automatically committed to the database
  – Turn auto-commit off to group two or more statements together into a transaction
    
    connection.setAutoCommit(false)
  – Call commit to permanently record the changes to the database after executing a group of statements
  – Call rollback if an error occurs
Transactions: Example

Connection connection =
    DriverManager.getConnection(url, username, passwd);
connection.setAutoCommit(false);
try {
    statement.executeUpdate(...);
    statement.executeUpdate(...);
    ...
} catch (SQLException e) {
    try {
        connection.rollback();
    } catch (SQLException sqle) {
        // report problem
    }
} finally {
    try {
        connection.commit();
        connection.close();
    } catch (SQLException sqle) { }
}

Useful Connection Methods
(for Transactions)

- **getAutoCommit/setAutoCommit**
  - By default, a connection is set to auto-commit
  - Retrieves or sets the auto-commit mode

- **commit**
  - Force all changes since the last call to commit to become permanent
  - Any database locks currently held by this `Connection` object are released

- **rollback**
  - Drops all changes since the previous call to commit
  - Releases any database locks held by this `Connection` object
Some JDBC Utilities

• Idea
  – Performing JDBC queries and formatting output are common tasks, so create helper classes to perform this function: DatabaseUtilities and DBResults

• Class methods
  – getQueryResults
    • Connects to a database, executes a query, retrieves all the rows as arrays of strings, and puts them inside a DBResults object
  – createTable
    • Given a table name, a string denoting the column formats, and an array of strings denoting row values, this method issues a CREATE TABLE command and then sends a series of INSERT INTO commands for each row
  – printTable
    • Given a table name, this method connects to the database, retrieves all the rows, and prints them on the standard output
  – printTableData
    • Given a DBResults object from a previous query, prints the results to standard output. Useful for debugging

Using JDBC Utilities

• Usage Example

```java
DBResults results = 
    DatabaseUtilities.getQueryResults(driver, url, 
    username, password, 
    query, true); 
out.println(results.toHTMLTable("CYAN"));
```
Summary

• In JDBC 1.0, can only step forward (next) through the ResultSet
• MetaDataResultSet provides details about returned ResultSet
• Improve performance through prepared statements
• Be sure to handle the situation where getXxx returns a NULL
• Be default, a connection is auto-commit
• SQL Exceptions are chained together

Questions?