JDBC

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

Relational Databases

- Persistent objects map to relation tables
Properties of a Relational Database

- Fundamental Properties
  - Tables, Indexes, Triggers, Stored Procedures, Transactions
  - ACID Properties – Atomicity, Consistency, Isolation, Durability
- Data Definition (DDL)
  - Manage Database Objects
  - Create and delete tables, indexes, stored procs
- Queries (DML)
  - Data Manipulation Language
  - Read, write, or delete information between one or more related tables
  - SQL

What is JDBC?

- A pure Java API for database communication similar to ODBC
  - JDBC and ODBC are based on the same standard: X/Open’s SQL Call-Level Interface
- Created in 1995
- A set of classes that perform database transactions
  - Connect to relational databases
  - Send SQL commands
  - Process results
  - Additional features: scrollable cursors, batch updates
JDBC Goals

- To provide Java programmers with a uniform, simple interface to a wide range of relational databases DB independence.
- It can replace underlying database with minimal code impact.
- It defines a common base on which higher level tools and interfaces can be built.

JDBC Architecture

![Diagram showing JDBC Architecture]
Type 1 Driver

- JDBC – ODBC Bridge
  - Translates all JDBC calls into ODBC (Open Database Connectivity)
- Need to have ODBC client installed on the machine

Type 2 Driver

- Native-API/Partly Java driver
  - Converts JDBC calls into db-specific calls
  - Communicates directly with the db server
  - Requires some binary code be present on the client machine.
  - Better performance than type 1 driver
Type 3 Driver

- Net-protocol – 100% Java driver
- Follows a three-tiered approach
  - JDBC database requests passed to the middle-tier server
  - Middle-tier server (middleware server, JEE application server) translates the request to the database-specific native-connectivity interface
    - May use a type 1 or type 2 JDBC driver
  - Request forwarded to the database server

Type 4 Driver

- Native protocol - 100% Java
- Converts JDBC calls into the vendor-specific DBMS protocol
- Client applications communicate directly with the database server
- Best performance
- Need a different driver for each database
**JDBC Classes**

**JDBC API**

- Defines a set of Java Interfaces, which are implemented by vendor-specific JDBC Drivers
  - Applications use this set of Java interfaces for performing database operations - portability
- 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
JDBC Driver

- Database specific implementation of JDBC interfaces
  - Every database server has corresponding JDBC driver(s)

- You can see the list of available drivers from

Database URL

- Used to make a connection to the database
  - Can contain server, port, protocol etc…

- jdbc:subprotocol_name:driver_depdendant_databasename
  - Oracle thin driver
    jdbc:oracle:thin:@machinename:1521:dbname
  - Derby
    jdbc:derby://localhost:1527/sample
  - Pointbase
    jdbc:pointbase:server://localhost/sample
Step By Step Usage of JDBC API

Steps of Using JDBC

1. Load DB-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 Pointbase DB Driver.
      // The driver has to be in the classpath.
      Class.forName("org.apache.derby.jdbc.ClientDriver");
  }

  }catch (ClassNotFoundException cnfe){
      System.out.println("" + cnfe);
  }
```

2. Get a Connection Object

• **DriverManager** class is responsible for selecting the database and creating the database connection

  - Using **DataSource** is a preferred means of getting a connection object (we will talk about this later)

• Create the database connection as follows:

  ```
  try {
      Connection connection =
      DriverManager.getConnection("jdbc:derby://localhost:1527/sample", "app"," app ");
  } catch(SQLException sqle) {
      System.out.println("" + sqle);
  }
```
DriverManager & Connection

- 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
  - java.sql.Statement
    - ResultSet executeQuery(string sql)
    - int executeUpdate(String sql)
- Example:
  Statement statement = connection.createStatement();

- The same **Statement** object can be used for many, unrelated queries
4. Executing Query or Update

• From the Statement object, the 2 most used commands are
  
  – (a) QUERY (SELECT)
    • ResultSet rs = statement.executeQuery("select * from customer_tbl");

  – (b) ACTION COMMAND (UPDATE/DELETE)
    • int iReturnValue = statement.executeUpdate("update manufacture_tbl set name = 'IBM' where mfr_num = 19985678");

5. Reading Results

• Loop through ResultSet retrieving information
  
  – java.sql.ResultSet
    • boolean next()
    • xxx getXxx(int columnName)
    • xxx getXxx(String columnName)
    • void close()

• The iterator is initialized to a position before the first row
  – You must call next() once to move it to the first row
5. Reading Results (Continued)

- Once you have the ResultSet, you can easily retrieve the data by looping through it

```java
while (rs.next()){
    // Wrong this will generate an error
    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
  - getString()
  - getInt()
  - getDouble()
  - getobject()

- There is an appropriate getXXX method of each java.sql.Types datatype
6. Read ResultSet MetaData and DatabaseMetaData (Optional)

- Once you have the `ResultSet` or `Connection` objects, you can obtain the Meta Data about the database or the query
- This gives valuable information about the data that you are retrieving or the database that you are using
  - `ResultSetMetaData rsMeta = rs.getMetaData();`
  - `DatabaseMetaData dbmetadata = connection.getMetaData();`
    - There are approximately 150 methods in the `DatabaseMetaData` class.

---

**ResultSetMetaData Example**

```java
ResultSetMetaData meta = rs.getMetaData();
//Return 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.setScale(i));
}
```
Prepared & Callable Statements

What Are They?

• PreparedStatement
  • SQL is sent to the database and compiled or prepared beforehand

• CallableStatement
  • Executes SQL Stored Procedures
PreparedStatement

- The contained SQL is sent to the database and compiled or prepared beforehand.
- From this point on, the prepared SQL is sent and this step is bypassed. The more dynamic Statement requires this step on every execution.
- Depending on the DB engine, the SQL may be cached and reused even for a different PreparedStatement and most of the work is done by the DB engine rather than the driver.

PreparedStatement cont.

- A PreparedStatement can take IN parameters, which act much like arguments to a method, for column values.
- PreparedStatements deal with data conversions that can be error prone in straight ahead, built on the fly SQL:
  - handling quotes and dates in a manner transparent to the developer.
**PreparedStatement Steps**

1. You register the drive and create the db connection in the usual manner

2. Once you have a db connection, create the prepared statement object

   ```java
   PreparedStatement updateSales = 
   con.prepareStatement("UPDATE OFFER_TBL SET 
   QUANTITY = ? WHERE ORDER_NUM = ? ");
   // “?” are referred to as Parameter Markers
   // Parameter Markers are referred to by number,
   // starting from 1, in left to right order.
   // PreparedStatement's setXXX() methods are used to set
   // the IN parameters, which remain set until changed.
   ```

**PreparedStatement Steps cont.**

3. Bind in your variables. The binding in of variables is positional based

   ```java
   updateSales.setInt(1, 75);
   updateSales.setInt(2, 10398001);
   ```

4. Once all the variables have been bound, then you execute the prepared statement

   ```java
   int iUpdatedRecords = updateSales.executeUpdate();
   ```
PreparedStatement Steps

- If AutoCommit is set to true, once the statement is executed, the changes are committed. From this point forth, you can just re-use the Prepared Statement object.

  ```java
  updateSales.setInt(1, 150);
  updateSales.setInt(2, 10398002);
  ```

PreparedStatement cont.

- If the prepared statement object is a select statement, then you execute it, and loop through the result set object the same as in the Basic JDBC example:

  ```java
  PreparedStatement itemsSold =
      con.prepareStatement("select o.order_num, o.customer_num, c.name, o.quantity from order_tbl o, customer_tbl c where o.customer_num = c.customer_num and o.customer_num = ?;";");
  itemsSold.setInt(1, 10398001);
  ResultSet rslItemsSold = itemsSold.executeQuery();
  while (rslItemsSold.next()){
      System.out.println( rslItemsSold.getString("NAME") + " sold " + rslItemsSold.getString("QUANTITY") + " unit(s)");
  }
  ```
CallableStatement

- The interface used to execute SQL stored procedures
- A stored procedure is a group of SQL statements that form a logical unit and perform a particular task
- Stored procedures are used to encapsulate a set of operations or queries to execute on a database server.

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.

```java
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;
Oracle Example

- This is an Oracle Specific example of a CallableStatement

```java
try {
    Connection connection = DriverManager.getConnection(""");

    queryreport.registerOutParameter(1, OracleTypes.CURSOR);
    queryreport.setInt(2, 10);
    queryreport.setString(3, "000004357");
    queryreport.setString(4, "01/07/2003");
    queryreport.setString(5, "N");
    queryreport.setString(6, "N");
    queryreport.setString(7, "N");
    queryreport.setInt(8, 2);

    queryreport.execute();
    ResultSet resultset = (ResultSet)queryreport.getObject(1);

    while (resultset.next())
    {
        System.out.println("" + resultset.getString(1) + " " + resultset.getString(2));
    }
} catch( SQLException sqle)
{
    System.out.println("" + sqle);
}
```
Transaction

- The committing of each statement when it is first executed is very time consuming.
- By setting AutoCommit to false, the developer can update the database more than once 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

```java
Connection connection = null;
try {
    connection = DriverManager.getConnection("jdbc:oracle:thin:@machinename:1521:dbname","username","password");
    connection.setAutoCommit(false);

    PreparedStatement updateQty = connection.prepareStatement("UPDATE STORE_SALES SET QTY = ? WHERE ITEM_CODE = ?");
```
Transaction Example cont.

    int [][] arrValueToUpdate =
    { {123, 500},
    {124, 250},
    {125, 10},
    {126, 350} };  

    int iRecordsUpdate = 0;
    for ( int items=0; items < arrValueToUpdate.length; items++) {
        int itemCode = arrValueToUpdate[items][0];
        int qty = arrValueToUpdate[items][1];
        updateQty.setInt(1, qty);
        updateQty.setInt(2, itemCode);
        iRecordsUpdate += updateQty.executeUpdate();
    }
    connection.commit();
    System.out.println(iRecordsUpdate + " record(s) have been updated");
}  
catch(SQLException sqlException) { 
    System.out.println("" + sqlException);
Transaction Example cont.

```
try {
    connection.rollback();
} catch (SQLException sqleRollback) {
    System.out.println("" + sqleRollback);
}
finally {
    try {
        connection.close();
    }
    catch (SQLException sqleClose) {
        System.out.println("" + sqleClose);
    }
}

SQL Exception

try {
    ...
} catch (SQLException ex) {
    System.out.println("\n-- SQLException caught --\n");
    while (ex != null) {
        System.out.println("Message:" + ex.getMessage());
        System.out.println("State: " + ex.getSQLState());
        System.out.println("Code:" + ex.getErrorCode());
        ex = ex.getNextException();
        System.out.println("");
    }
}