





# JDBC



1



**Edward Haynes**  
[edwardahaynes@yahoo.com](mailto:edwardahaynes@yahoo.com)

**Sang Shin**  
[sang.shin@sun.com](mailto:sang.shin@sun.com)




**J2EE™ Programming with Passion!**  
[www.plurb.com/j2ee](http://www.plurb.com/j2ee)

1



## Disclaimer & Acknowledgments


- ✦ Even though Sang Shin is a full-time employee of Sun Microsystems, the contents here are created as his own personal endeavor and thus does not reflect any official stance of Sun Microsystems.
- ✦ Sun Microsystems is not responsible for any inaccuracies in the contents.



## Revision History

- ? 01/24/1998: version 1, created with speaker notes (Edward Haynes)
- ? 02/02/2003: version 2, cosmetic changed applied (Sang)

4



## Session Objectives

- ✦ Working knowledge of most commonly used classes of JDBC API
  - ✦ Connection
  - ✦ Statement
  - ✦ ResultSet
- ✦ Discuss practical usage of JDBC

1

Tonight we are going to cover some of the basics of SQL. We will talk about what DML and DDL are. We will learn how to retrieve data, how to add new data and how to modify existing data with in a Database. We are also going to learn some SQL statements that are used to define a database.

Once we have covered SQL, we are then going to discuss the JDBC API from Sun and then put to use our understanding of SQL with some code that interacts with a database.



## Agenda

- ✦ JDBC Overview
- ✦ Basic DB Manipulation
- ✦ Advanced DB Manipulation
- ✦ Connection Pooling and DataSource



The header of the slide features a horizontal banner. On the left, there is a photograph of the Great Pyramids of Giza at sunset, with the sun low on the horizon between two pyramids. To the right of the photo is the Sun Microsystems logo, which consists of a stylized 'Sun' symbol (a circle with eight rays) and the text 'Sun' in a script font, with 'microsystems' in a smaller sans-serif font below it. The entire banner has a yellow and red gradient background.

# JDBC API



A small cartoon character, resembling a white dog or cat with a red collar, is standing on a yellow oval. It is holding a stack of computer hardware, including a monitor, a tower unit, and a keyboard.

1



## JDBC API

- ? **Java Database Connectivity API**
- ? **API to generalize connections to any SQL compliant database.**
  - Developer no longer has to know db specific commands
- ? **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





# Database URL



## Database URL

- ? Used to make a connection to the database
  - Can contain server, port, protocol etc...
- ? jdbc:subprotocol\_name:driver\_dependant\_info
  - odbc bridge
    - jdbc:odbc:COREJAVA
  - oracle thin driver
    - 1.jdbc:oracle:thin:@machinename:1521:dbname
  - Pointbase
    - jdbc:pointbase:server://localhost/sample

1

The Database URL is database vendor specific. So when developing for a specific database, such as Oracle or Pointbase, you should review their documentation for the exact usage.



# Making a Connection



## Making a Connection

- ? To manually load the driver, load its class file
  - `Class.forName` which throws `ClassNotFoundException`
  - This must be done in a try/catch block

```
try {  
    //this loads an instance of the Pointbase DB Driver  
    Class.forName("com.pointbase.jdbc.jdbcUniversalDriver");  
  
} catch (ClassNotFoundException cnfe){  
    System.out.println("" + cnfe);  
}
```

1

- To manually load the drivers, its an easy 1 liner. This loads the driver with the current JVM




## Making a Connection cont.

```
? 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:pointbase:server://localhost/s
    ample", "pbpublic", "pbpublic ");
    } catch(SQLException sqle) {
        System.out.println("" + sqle);
    }
}
```

1

- This is just a quick example. In real world usage, you obviously declare the connection variable outside of the try.



## Making a Connection cont.

- ? **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

1

This is a quick view of the DriverManager and Connection API's. These are just a few of the methods for each class.

If you want to see them all, just review the javadoc



# Executing SQL



## Executing SQL

- ? **Create a Statement Object which allows you to execute a SQL query**
  - `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**

1

Once you have the Connection object, you can then execute SQL statements against the database. The Statement Object is responsible for sending the SQL commands to the database and for retrieving the results of that command.

The same Statement object can be used in multiple queries. You just have to call the necessary method with the appropriate SQL string





## Executing SQL cont.

- ? 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");`

1

? Here are the 2 most used SQL Commands. A Select and an Update/Insert

? The main different between these two commands is that the QUERY command returns a ResultSet object while the ACTION COMMAND returns an int value. The ResultSet contains the values of your query while the int is the total number of records affected by your statement.



# Displaying Results



## Displaying Results

### ? Loop through ResultSet retrieving information

- java.sql.ResultSet
  - boolean next()
  - xxx getXxx(int columnNumber)
  - xxx getXxx(String columnName)
  - void close()

### ? Note: the iterator is initialized to a position before the first row

- You must use next() once to move it to the first row

1

? The executeQuery() object returns a ResultSet object which is a 2 dimensional array that contains the results of the executed SQL statement. The developer has to loop through this object and retrieve each column and row. Unless the ResultSet is a scrollable ResultSet (advanced topics), once you read through the ResultSet once, you can no longer make use of it. In most cases, the developer loops through the ResultSet and stores the values in Collection so that the data can be used at a later date.

? A scrollable ResultSet is a ResultSet in which you can loop forward and backwards until the ResultSet object is closed. I haven't had much experience with Scrollable ResultSets because they are Vendor specific as to who supports them and the implementation is not all that reliable in my experience. If you are interested in loop through the data more then once, or going to any specify row/column at any point in time, you might want to consider converting the resultset to a Collection such as ArrayList or Vector. Once you have created the Collection Object, you are free to manipulate the data in any manner availble through those objects



## Displaying Results cont.

? In the case of the QUERY statement, once you have the ResultSet you can easily retrieve the data by looping through it

- Example

```
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");  
}
```

1

? Unlike Array's, the index for a ResultSet starts at 1, not 0.

? You can specify either the column name or the column number



## Displaying Results cont.

- ? 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

1

? These are just a few of the many choices. Check the `java.sql.ResultSet` API for a full listing of the accessor methods

? Note: you can use `getString()` on any column type and it will make the conversion for you.



## ResultSet MetaData and DatabaseMetaData

- ? 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.

- Both the ResultSetMetaData and the DatabaseMetaData provide extremely useful information to a developer about



## ResultSetMetaData Example

```
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.getScale(i) );  
}
```

1

- The getColumnCount() method of the ResultSetMetaData Object is very useful for looping through a resultSet. B.

The header of the slide features a horizontal banner. On the left, there is a photograph of the Great Pyramids of Giza at sunset, with the sun low on the horizon between two pyramids. To the right of the photo is the Sun Microsystems logo, which includes a stylized 'Sun' icon and the text 'Sun microsystems'.

# Advanced DB Topics

A small cartoon character, resembling a white stick figure with a red face, is standing on a yellow oval. The character is holding a computer monitor in its right hand and a small object in its left hand.

1





## Advanced DB Topics

- ? **PreparedStatement**
  - SQL is sent to the database and compiled or prepared beforehand.
- ? **CallableStatement**
  - Executes SQL Stored Procedures.
- ? **Transactions**
- ? **Connection Pooling**

1

- Once you get comfortable using the basic Statement object, try using the PreparedStatement Object and the Call



# PreparedStatement



## 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.

1

- A PreparedStatement is very similar in concept to a regular Statement object.
- In a practical application, using a PreparedStatement is the more preferred method to executing SQL queries the
- You can also change the input parameters of the statement without have to resend the PreparedStatement to the c



## 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

1

- As I stated before, a PreparedStatement takes in parameters, much like arguments to a method.
- Another added bonus to using PreparedStatement, is that the underlying database handles the data conversions. S



## 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

```
PreparedStatement updateSales =  
    con.prepareStatement("UPDATE OFFER_TBL SET  
        QUANTITY = ? WHERE ORDER_NUM = ? ");
```

- “?” are referred to as Parameter Markers

1

•The question marks are stand-ins for values to be set before statement execution and are called parameter markers



## PreparedStatement Steps cont.

- 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.

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

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

### **4. Once all the vairables have been bound, then you execute the prepared statement**

```
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.

```
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:

```
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 rsItemsSold = itemsSold.executeQuery();  
while (rsItemsSold.next())  
{  
    System.out.println( rsItemsSold.getString("NAME") + " sold  
    " + rsItemsSold.getString("QUANTITY") + " unit(s)");  
}
```

1

- For an update or insert, once you have the PreparedStatement object, you execute it as often as you like. For example, your query.





# CallableStatement



## 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.

1

- A CallableStatement extends PreparedStatement. So in essence, it works the same as the PreparedStatement obj



## 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();
```

1

•In this example, I am calling a stored procedure that returns a simple ResultSet object. The executeQuery method



## 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 );
```

1

- Once very big difference between `PreparedStatement`'s and `CallableStatement`'s is that with `PreparedStatement`'s you
- With `CallableStatement`'s, you have what is known as OUT and INOUT parameters.
- You bind in your IN variables just as you would with a `PreparedStatement`. You can have a stored procedure that return all different
- OUT parameters are just like return variables. The value returned by the Stored procedure is considered an OUT parameter, but ;



## 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;
```

1

- Here is an example of a stored procedure that has both IN, OUT and INOUT parameters and it returns a Cursor of data (this is an oracle spec that just defines the layout of the stored procedure)
- In this case, you would bind in your input parameters (appl\_id, dow\_in) and your INOUT parameter( status\_in), you would also need to register your OUT parameters and the INOUT (event\_type\_in and status\_in)
- Once you execute your statement, you retrieve your OUT values with the appropriate getXXX method



## Oracle Example

- ? This is an Oracle Specific example of a CallableStatement

```
try {
    Connection connection = DriverManager.getConnection("");
    CallableStatement queryreport = connection.prepareCall("{ ? = call
SRO21208_PKG.QUEUE_REPORT ( ? , ? , ? , ? , ? , ? , ? ) }");

    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);
```

1

- This statement takes a couple of IN parameters and has 1 OUT parameter. The OUT parameter is just an Oracle



## Oracle Example cont.

```
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);
}
```

1

- Once you have the ResultSet Object, you just loop through it in the normal manner



# Transaction





## Transaction

- ? One of the main benefits to using a PreparedStatement is executing the statements in a transactional manner
- ? 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.

1

- A transaction is when you have multiple statements that you want to have occur as a block that in case of failure a
- All or none
- By programming multiple update/inserts statements in a transactional manner, you will get faster results because tl



## Transactions Example

```
Connection connection = null;
try {
    connection =
        DriverManager.getConnection("jdbc:oracle:thin:@machinename:15
21:dbname","username","password");
    connection.setAutoCommit(false);

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

1

? To start a transaction, you create your PreparedStatement object as you normally would.

? The main difference here is that once you have the connection object, you turn off the Auto Committing feature, setAutoCommit(false)

1. Turned on by default



## 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];
```

1

? Here, we create an array of integers (for a simple example). Then loop through the array binding in each set of values.



## Transaction Example cont.

```
        updateQty.setInt(1,qty);
        updateQty.setInt(2,itemCode);
        iRecordsUpdate += updateQty.executeUpdate();
    }
    connection.commit();
    System.out.println(iRecordsUpdate + " record(s) have been
updated");
} catch(SQLException sqle) {
    System.out.println("" + sqle);
}
```

1

- ? Once each set is bound to the PreparedStatement object, then issue an executeUpdate() command on the statement.
- ? If you don't call the commit() explicitly, then none of you changes will be saved



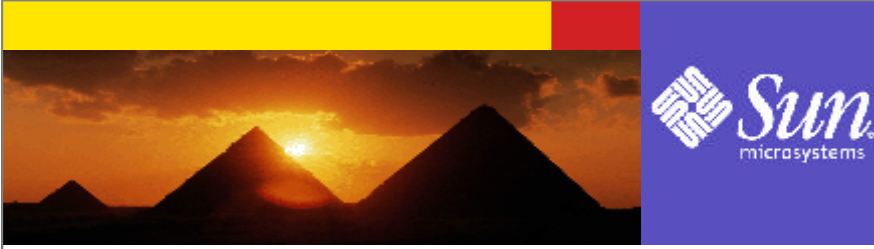
## Transaction Example cont.

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

1


? The entire process is wrapped in 1 try/catch block. The point behind this is that if there is a `SQLException` at any point, you can catch this error and then rollback the entire batch as a whole.

? Also, by closing the db connection in the finally statement, this guarantees that in the event of a `SQLException`, that the db connection will not be left open.



The banner image at the top of the slide features a landscape with three pyramids under a sunset sky. The Sun Microsystems logo is positioned on the right side of the banner, which has a yellow and red header bar.

# JNDI DataSource



A small cartoon character, resembling a penguin or a stick figure, is standing on a yellow patch of ground. It is holding a briefcase and looking towards the left.

1



## JNDI DataSource

- The Java™ Naming and Directory Interface™ (JNDI) makes it possible to connect to a database using a logical name instead of having to hard code a particular database and driver.
- 2. In a server such as Tomcat, you define a DataSource inside of the server.xml where you specify the username, password, connect string and other parameters
- 3. The DataSource also needs to be defined inside of your applications web.xml

1

? DataSource's are primarily used with application that run on a J2EE compliant server, such as Tomcat, Jrun etc.

? The main benefit for using a DataSource is that your code can be independent of any database specific commands. This makes your application very portable. As long as the database vendor has a set of JDBC drivers, then your application can be run on that database with little or no changes



## JNDI DataSource Example

? In your server.xml, you have the following:

```
<Resource name="jdbc/rtl-shaws"
    auth="Container"
    type="javax.sql.DataSource"
    scope="Shareable"
    description="DBCP Connection Pool DataSource"/>
<ResourceParams name="jdbc/rtl-shaws">
    <parameter>
        <name>factory</name>
        <value>org.apache.commons.dbcp.BasicDataSourceFactory</value>
    </parameter>
```

1

- This is a Tomcat specific example using the DBCP (Database Connection Pooling) package from Jakarta connection
- This entry can go into a couple of different place in your server.xml
  - <Context>
    - Accessible to only your application
  - <GlobalNamingResources>
    - Accessible by all applications
  - <DefaultContext>





## DataSource Example cont.

```
<parameter>
  <name>driverClassName</name>
  <value>oracle.jdbc.driver.OracleDriver</value>
</parameter>
<parameter>
  <name>username</name>
  <value>uuuuu</value>
</parameter>
<parameter>
  <name>password</name>
  <value>ppppp</value>
</parameter>
```



## DataSource Example cont.

```
<parameter>
  <name>url</name>
  <value>jdbc:oracle:thin:@dellisp:1521:rtl</value>
</parameter>
<parameter>
  <name>maxActive</name>
  <value>10</value>
</parameter>
<parameter>
  <name>maxIdle</name>
  <value>30000</value>
</parameter>
```



## DataSource Example cont.

```
<parameter>
  <name>maxWait</name>
  <value>100</value>
</parameter>
<parameter>
  <name>removeAbandoned</name>
  <value>true</value>
</parameter>
<parameter>
  <name>removeAbandonedTimeout</name>
  <value>60</value>
</parameter>
```



## DataSource Example cont.

```
<parameter>
  <name>logAbandoned</name>
  <value>true</value>
</parameter>

</ResourceParams>
```



## DataSource Example cont.

? In your web.xml you have the following

```
<resource-ref>
  <description>DB Connection</description>
  <res-ref-name>jdbc/rtl-shaws</res-ref-name>
  <res-type>javax.sql.DataSource</res-type>
  <res-auth>Container</res-auth>
</resource-ref>
```

1

- Once you have the jndi entry in the server.xml setup, you then have to declare a *resource-ref* entry in your applic



## DataSource Example cont.

### ? To make a connection using the DataSource

```
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
try{
    Context ctx = new InitialContext();
    String strContext = "java:comp/env/" + datasource;
    ds = (DataSource)ctx.lookup(strContext);
    conn = ds.getConnection();
    //NOW that you have your connection.....
}catch(NamingException ne){
    System.out.println("Datasource Error : " + ne);
    ne.printStackTrace();
}
```



The header of the slide features a horizontal banner. On the left, there is a photograph of the Great Pyramids of Giza at sunset, with the sun low on the horizon between two pyramids. To the right of the photo is the Sun Microsystems logo, which consists of a stylized 'Sun' symbol (a circle with eight rays) and the text 'Sun' in a large, italicized font, with 'microsystems' in a smaller font below it. The entire banner has a yellow and red gradient background.

# Connection Pooling



A small cartoon character, resembling a white stick figure with a red face and a black hat, is standing on a yellow oval. The character is holding a briefcase in its right hand and has a small blue building or structure behind it.

1



## Connection Pooling

- Another benefit to using a DataSource, is that you can have Connection Pooling setup without interfering with the developer
- A connection pool is a cache of open connections that can be used and reused, thus cutting down on the overhead of creating and destroying database connections
- The previous example uses the Jakarta DBCP (Database Connection Pooling) implementation





The header of the slide features a horizontal banner. On the left, a photograph shows a sunset over the Great Pyramids of Giza. To the right of the photo is the Sun Microsystems logo, which includes a stylized 'Sun' icon and the text 'Sun microsystems'.

**Live your life  
with Passion!**



A small cartoon character is positioned below the text. The character is white with a red shirt and black pants, holding a computer monitor in its right hand. It is standing on a small yellow patch of ground.

57

S