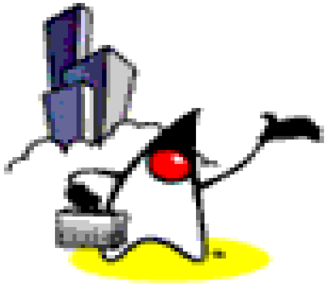
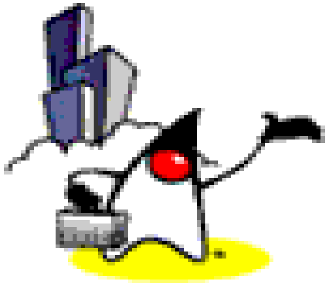
JDBC Basics



Agenda

- What is JDBC?
- Step By Step Usage of JDBC API
- DataSource & Connection Pooling
- Transaction
- Prepared and Callable Statements

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 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 implemention of JDBC interfaces
  - Every database server has corresponding JDBC driver(s)
- You can see the list of available drivers from
  - http://industry.java.sun.com/products/jdbc/drivers

Database URL

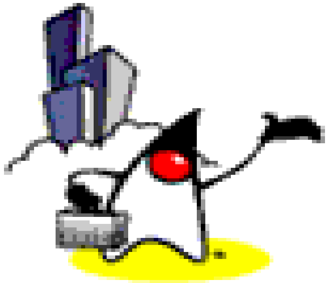
- Used to make a connection to the database
  - Can contain server, port, protocol etc…
- jdbc:subprotocol_name:driver_dependant_databasename
  - Oracle thin driver

    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 and creating the database connection
  - Using DataSource is a prefered means of getting a conection 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 columnNumber)
    - 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

```
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)

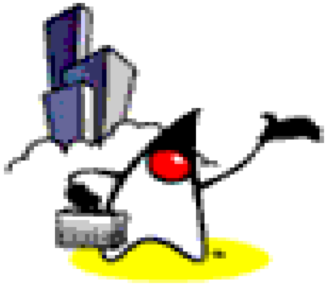
- Once you have the ResultSet or Connection objects, you can obtain the Meta Data about the database or the query
- This gives valuable information about the data that you are retrieving or the database that you are using
  - ResultSetMetaData rsMeta = rs.getMetaData();
  - DatabaseMetaData dbmetadata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

ResultSetMetaData Example

```
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) );
}
```

DataSource & Connection Pooling



Sub-Topics

- DataSource interface and DataSource object
- Properties of a DataSource object
- JNDI registration of a DataSource object
- DataSource object that implements Connection pooling
- Retrieval of DataSource object (within your application)

javax.sql.DataSource Interface and DataSource Object

- Driver vendor implements the interface
- DataSource object is the factory for creating database connections

javax.sql.DataSource Interface and DataSource Object

- Three types of possible implementations
  - Basic implementation: produces standard Connection object
  - Connection pooling implementation: produces a Connection object that will automatically participate in connection pooling
  - Distributed transaction implementation: produces a Connection object that may be used for distributed transactions and almost always participates in connection pooling

Properties of DataSource Object

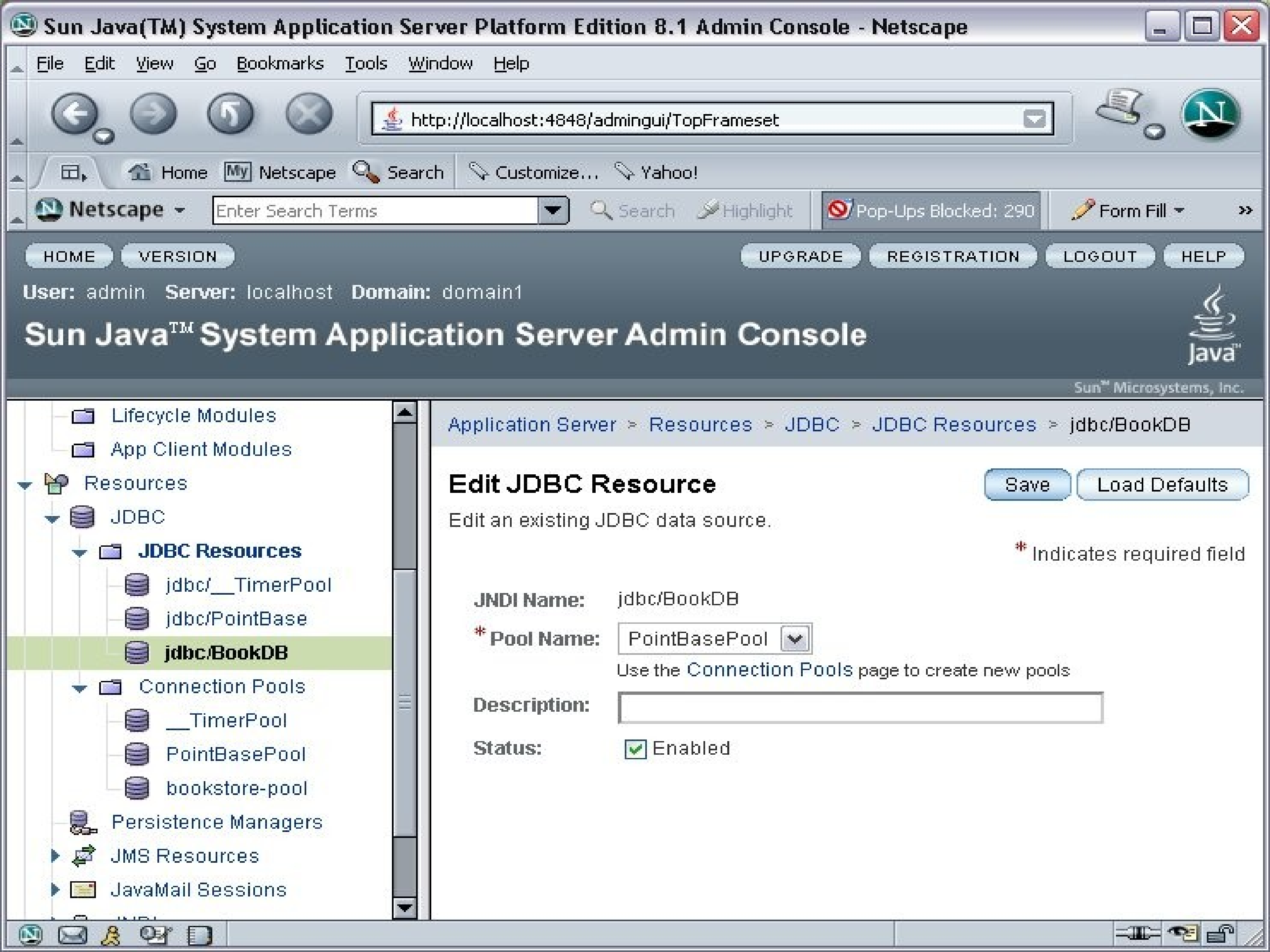
- A DataSource object has properties that can be modified when necessary – these are defined in a container's configuration file
  - location of the database server
  - name of the database
  - network protocol to use to communicate with the server
- The benefit is that because the data source's properties can be changed, any code accessing that data source does not need to be changed
- In the Sun Java System Application Server, a data source is called a JDBC resource

Where Are Properties of a DataSource Defined?

- In container's configuration file
- In Sun Java System App Server, they are defined in
  - <J2EE_HOME>/domains/domain1/config/domain.xml
- In Tomcat, they are defined in server.xml
  - <TOMCAT_HOME>/conf/server.xml

DataSource (JDBC Resource) Definition in Sun Java System App Server's domain.xml

```
<resources>
   <jdbc-resource enabled="true" jndi-name="jdbc/BookDB" object-
    type="user" pool-name="PointBasePool"/>
   <jdbc-connection-pool connection-validation-method="auto-commit"
    datasource-classname="com.pointbase.xa.xaDataSource" fail-all-
    connections="false" idle-timeout-in-seconds="300" is-connection-
    validation-required="false" is-isolation-level-guaranteed="true" max-
    pool-size="32" max-wait-time-in-millis="60000" name="PointBasePool"
    pool-resize-quantity="2" res-type="javax.sql.XADataSource" steady-
    pool-size="8">
     <property name="DatabaseName"
    value="jdbc:pointbase:server://localhost:9092/sun-appserv-samples"/>
     <property name="Password" value="pbPublic"/>
     <property name="User" value="pbPublic"/>
   </jdbc-connection-pool>
</resources>
```



Sun Java(TM) System Application Server Platform Edition 8.1 Admin Console - Netscape
File Edit View Go Bookmarks Tools Window Help
http://localhost:4848/admingui/TopFrameset
Home Netscape Search Customize... Yahoo!
Netscape Enter Search Terms Search Highlight Pop-Ups Blocked: 290 Form Fill
HOME VERSION UPGRADE REGISTRATION LOGOUT HELP
Java
Sun™ Microsystems, Inc.
Lifecycle Modules
App Client Modules
Resources
JDBC
JDBC Resources
jdbc/__TimerPool
jdbc/PointBase
jdbc/BookDB
Connection Pools
__TimerPool
PointBasePool
bookstore-pool
Persistence Managers
JMS Resources
JavaMail Sessions
Application Server > Resources > JDBC > JDBC Resources > jdbc/BookDB
Save
Load Defaults

User: admin Server: localhost Domain: domain1

Sun Java™ System Application Server Admin Console

Edit JDBC Resource

Edit an existing JDBC data source.

* Indicates required field

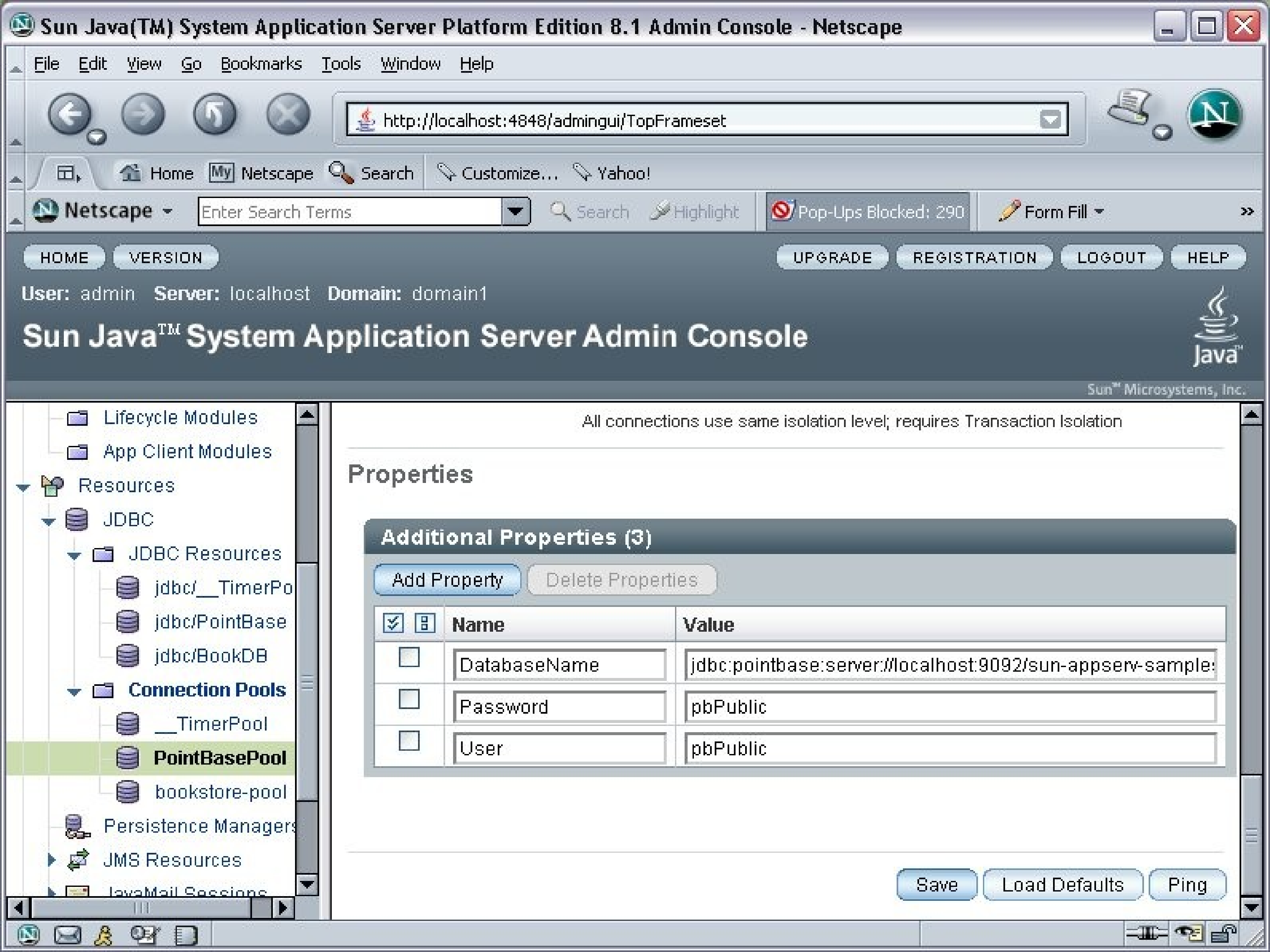
JNDI Name: jdbc/BookDB

* Pool Name: PointBasePool

Use the Connection Pools page to create new pools

Description:

Status: ☑ Enabled



User: admin **Server:** localhost **Domain:** domain1

Sun Java™ System Application Server Admin Console

All connections use same isolation level; requires Transaction Isolation

Properties

Additional Properties (3)

☑	Name	Value
□	DatabaseName	jdbc:pointbase:server://localhost:9092/sun-appserv-sample
□	Password	pbPublic
□	User	pbPublic

JNDI Registration of a DataSource Object

- A driver that is accessed via a DataSource object does not register itself with the DriverManager
- Rather, a DataSource object is registered to JNDI naming service by the container and then retrieved by a client though a lookup operation
- With a basic implementation, the connection obtained through a DataSource object is identical to a connection obtained through the DriverManager facility

JNDI Registration of a DataSource (JDBC Resource) Object

- The JNDI name of a JDBC resource is expected in the java:comp/env/jdbc subcontext
  - For example, the JNDI name for the resource of a BookDB database could be java:comp/env/jdbc/BookDB
- Because all resource JNDI names are in the java:comp/env subcontext, when you specify the JNDI name of a JDBC resource enter only jdbc/name. For example, for a payroll database, specify jdbc/BookDB

Why Connection Pooling?

- Database connection is an expensive and limited resource
  - Using connection pooling, a smaller number of connections are shared by a larger number of clients
- Creating and destroying database connections are expensive operations
  - Using connection pooling, a set of connections are pre-created and are available as needed basis cutting down on the overhead of creating and destroying database connections

Connection Pooling & DataSource

- DataSource objects that implement connection pooling also produce a connection to the particular data source that the DataSource class represents
- The connection object that the getConnection method returns is a handle to a PooledConnection object rather than being a physical connection
  - The application code works the same way

Example: PointBasePool

- The Sun Java Application Server 8 is distributed with a connection pool named PointBasePool, which handles connections to the PointBase database server
- Under Sun Java Application Server, each DataSource object is associated with a connection pool

Retrieval and Usage of a DataSource Object

- Application perform JNDI lookup operation to retrieve DataSource object
- DataSource object is then used to retrieve a Connection object
- In the application's web.xml, information on external resource, DataSource object in this case, is provided
- For Sun Java System App server, the mapping of external resource and JNDI name is provided
  - This provides further flexibility

Example: Retrieval of DataSource Object via JNDI

- BookDBAO.java in bookstore1 application

```
public class BookDBAO {
    private ArrayList books;
    Connection con;
    private boolean conFree = true;

    public BookDBAO() throws Exception {
        try {
            Context initCtx = new InitialContext();
            Context envCtx = (Context) initCtx.lookup("java:comp/env");
            DataSource ds = (DataSource)
    envCtx.lookup("jdbc/BookDB");
           con = ds.getConnection();
        } catch (Exception ex) {
            throw new Exception("Couldn't open connection to database: " +
                ex.getMessage());
        }
    }
```

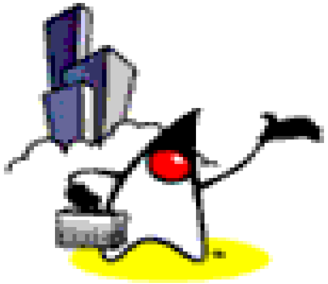
JNDI Resource Information in bookstore1's web.xml

```
<resource-ref>
  <res-ref-name>jdbc/BookDB</res-ref-name>
  <res-type>javax.sql.DataSource</res-type>
  <res-auth>Container</res-auth>
  <res-sharing-scope>Shareable</res-sharing-scope>
</resource-ref>
```

JNDI and Resource Mapping in bookstore1's sun-web.xml

```
 <sun-web-app>
  <context-root>/bookstore1</context-root>
  <resource-ref>
    <res-ref-name>jdbc/BookDB</res-ref-name>
    <jndi-name>jdbc/BookDB</jndi-name>
  </resource-ref>
</sun-web-app>
```

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 then 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

```
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];
```

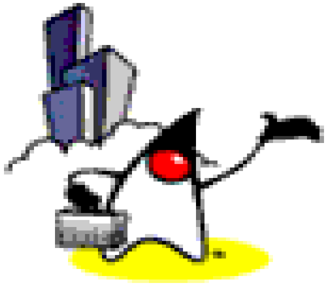
Transaction Example cont.

```
          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);
```

Transaction Example cont.

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

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

```
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

```
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)");
}
```

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.

```
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

```
  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);
```

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);

  }
```

Passion!