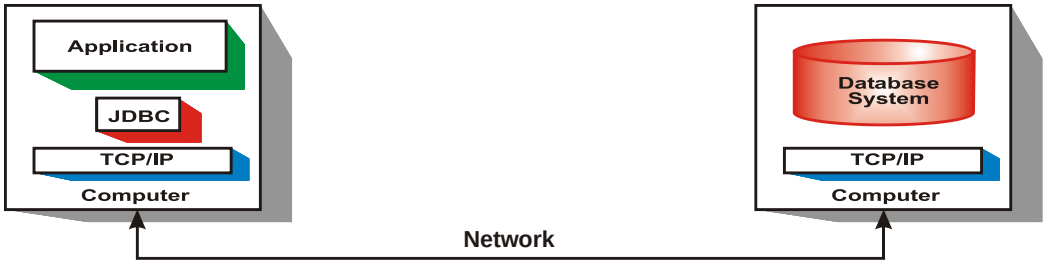
Network Computing Lecture 10

Professor Louise E. Moser

Spring 2012

More on Java Database Connectivity

- **Java Database Connectivity** (**JDBC**) sends the **Structured Query Language** (**SQL**) code that you created in your Java program to the **Database Management System** (**DBMS**) to
  - Establish a connection
  - Manipulate a table
    - Create the table
    - Insert values into the table and update the table
    - Query a table and retrieve results from the table
  - Commit and abort a transaction

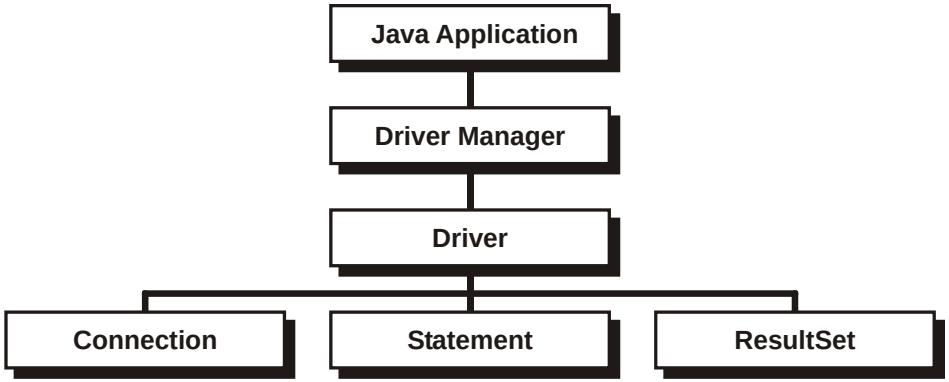


Establishing a Connection

- **Step 1. Loading the Driver**
  - For the JDBC/ODBC Bridge Driver, use

    Class.forName*("sun.jdbc.odbc.JdbcOdbcDriver");*
  - This code automatically creates an instance of the Driver and registers it with the Driver Manager
  - For a different driver, say *jdbc.DriverXYZ*, use

    Class.forName*("jdbc.DriverXYZ");*



Establishing a Connection (cont)

- **Step 2. Making the Connection**
  - Make the Driver connect to the DBMS,

    Connection conn = ***DriverManager.getConnection****(*
    url, "myLogin", "myPassword");
  - For the JDBC/ODBC Bridge Driver,
    - *url* is of the form *jdbc:odbc:xxxx*, where *xxxx* is your data source name or database system
    - *myLogin* and *myPassword* are the name and password that you use to login to the DBMS

Converting from SQL to JDBC

- To create, update, or query a table in your database
  - First, create a *Statement* object in Java for the connection that you already have gotten
  - Next, create a *String* in Java that corresponds to the SQL command that you want to execute
  - Then, invoke *execute()*, *executeUpdate()*, *executeQuery()*, etc. on the *Statement* object with the *String* as a parameter

Example: COFFEES Table

COF_NAME	SUP_ID	PRICE	SALES	TOTAL
Columbian	101	7.99	0	0
French_Roast	49	8.99	0	0
Expresso	150	9.99	0	0
Columbian_Decaf	101	8.99	0	0
French_Roast_Decaf	49	9.99	0	0

Example: SUPPLIERS Table

SUP_ID	SUP_NAME	STREET	CITY	STATE	ZIP
101	Acme, Inc.	99 Market Street	Groundsville	CA	95199
49	Superior Coffee	1 Party Place	Mendocino	CA	95460
150	The High Ground	100 Coffee Lane	Meadows	CA	93966

Creating a Database Table

- Example: COFFEES Table
- In **SQL**

  CREATE TABLE *COFFEES*
  (COF_NAME VARCHAR(32), SUP_ID INTEGER,
  PRICE FLOAT,
  SALES INTEGER, TOTAL INTEGER)

- This sets up a table with a particular structure
  - The table is empty; it does not contain any data yet

Creating a Database Table (cont)

- In **Java**
  - First, create a **Statement object**

    Statement ***stmt*** *=* ***conn.createStatement();***
  - Next, create a **String**

    String ***str*** *= "****CREATE TABLE*** *COFFEES " +*
    "(COF_NAME VARCHAR(32), SUP_ID INTEGER, " +
    "PRICE FLOAT, " +
    "SALES INTEGER, TOTAL INTEGER)";
  - Then, invoke the *execute()* method of the Statement object with the String as a parameter

    stmt.execute(str);
  - This sends the *CREATE TABLE* statement via JDBC over the connection to the DBMS

Entering Data into a Table

- In **SQL**

INSERT INTO *COFFEES*
VALUES *('Columbian', 101, 7.99, 0, 0)*

Entering Data into a Table (cont)

- In **Java**
  - First, create a statement object

    Statement ***stmt*** = ***conn.createStatement();***
  - Next, create a **String**

    String ***str*** = *"**INSERT INTO** COFFEES " +*

    *"**VALUES** ('Columbian', 101, 7.99, 0, 0)";*
  - Then, use the *executeUpdate()* method of the Statement object with the String as a parameter to send the ***INSERT INTO*** statement via JDBC over the connection to the DBMS

    stmt.executeUpdate(str);

Entering Data into a Table (cont)

- Alternatively, in **Java**
  - First, create a statement object

    Statement ***stmt*** = ***conn.createStatement();***
  - Skip the step of introducing the String variable *str*
  - Use the *executeUpdate()* method to send the

    INSERT INTO statement via JDBC over the connection to the DBMS

    stmt.executeUpdate(

    *"**INSERT INTO** COFFEES " +*

    *"**VALUES** ('Columbian', 101, 7.99, 0, 0)");*

Retrieving Data from a Table

- In **SQL**
  - To get the entire table *COFFEES*

    SELECT * FROM *COFFEES*
  - To get a list of coffees and their price / pound

    SELECT *COF_NAME, PRICE* ***FROM*** *COFFEES*
  - To get a list of coffees that cost less that $10.00 / pound

    SELECT *COF_NAME, PRICE* ***FROM*** *COFFEES*

    WHERE *PRICE < 10.00*

Retrieving Data from a Table (cont)

- In **Java**
  - First, create a statement object

    Statement ***stmt*** = ***conn.createStatement();***
  - Then, use the *executeQuery()* method to send the *SELECT FROM* statement via JDBC over the connection to the DBMS

    ResultSet ***rs*** = ***stmt.executeQuery***(

    *"**SELECT** COF_NAME, PRICE **FROM** COFFEES");*
  - The *executeQuery()* method returns a *ResultSet*, i.e., a table (sequence of rows, each with two entries, i.e., *COF_NAME*, *PRICE*), as the result

Retrieving Data from a Table (cont)

- To access the coffee names and prices, go to each row of *ResultSet* and retrieve the values according to their types using the ***getXXX()*** methods
- The method ***next()*** moves the cursor to the next row of *ResultSet*, starting with the first row
- Example:

```
Statement stmt = conn.createStatement();
String str = "SELECT COF_NAME, PRICE FROM COFFEES";
ResultSet rs = stmt.executeQuery(str);
while (rs.next()) {
    String n = rs.getString("COF_NAME");
    float p = rs.getFloat("PRICE");
    System.out.println(n + "   " + p);
}
```

- Note that, when a *ResultSet* is created, the cursor points above the first row so that, when you execute *rs.next()* the first time, the cursor points to the first row

Updating a Table

- In **SQL**
  - To update the *SALES* in the table *COFFEES*

    UPDATE *COFFEES* ***SET*** *SALES = 75*

    WHERE *COF_NAME* ***LIKE*** *'Columbian'*

Updating a Table

- In **Java**
  - To update the *SALES* in the table *COFFEES* and execute the SQL statement contained in *updateStr*

```
Statement stmt = conn.createStatement();
String updateStr = "UPDATE COFFEES " +
                   "SET SALES = 75 " +
                   "WHERE COF_NAME LIKE 'Columbian'";
stmt.executeUpdate(updateStr);
```

  - To retrieve the value that was just set, after executing a query to obtain the *ResultSet rs* as on Slide 14

```
rs.next();
String s = rs.getString(1);
int n = rs.getInt(2);
System.out.println(n + " pounds of " + s + "sold this week.");
```

  - Note that *next()* is used, even though there is only one row in *ResultSet*

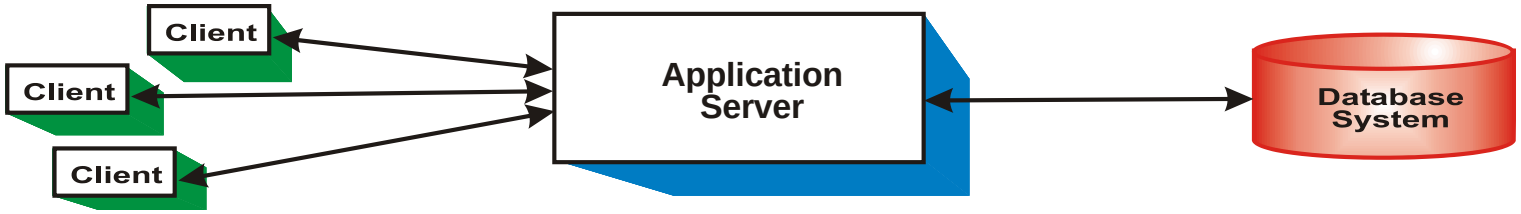
Updating a Table (cont)

- In **Java**
  - To update the *TOTAL* column by adding the weekly amount sold to the existing total

```
String updateStr = "UPDATE COFFEES " +
                   "SET TOTAL = TOTAL + 75 " +
                   "WHERE COF_NAME LIKE 'Columbian'";
stmt.executeUpdate(updateStr);
String queryStr = "SELECT COF_NAME, TOTAL FROM COFFEES " +
                  "WHERE COF_NAME LIKE 'Columbian'";
ResultSet rs = stmt.executeQuery(queryStr);
while (rs.next()) {
    String s = rs.getString(1);
    int n = rs.getInt(2);
    System.out.println(n + " pounds of " + s + " sold to date.")
 }
```

Transaction Processing

- In the three-tier architecture, a back-end database system is typically used with a middle-tier server that is programmed using the **transaction processing programming model**
  - When a server application receives a client's request, it initiates one or more transactions
  - When it finishes processing the request, the server application
    - Commits the transaction
    - Stores the resulting state in the back-end database
    - Returns the result to the client



What Is a Transaction?

- A **transaction** is a set of operations on the application state that exhibit the **ACID** properties (defined by Jim Gray)
  - **Atomicity** - Either ALL of the operations succeed in which case the transaction **commits**, or NONE of the operations is carried out in which case the transaction **aborts**
  - **Consistency** - If the application state is consistent at the start of a transaction, the application state remains consistent after the transaction commits
  - **Isolation** - One transaction does NOT read or overwrite intermediate results produced by another transaction, i.e., the transactions appear to execute serially (one after the other)
  - **Durability** - The update to the application state becomes permanent (persists) once the transaction is committed, even if a fault occurs

Transaction Processing System

- A **Transaction Processing (TP) System** typically consists of
  - TP monitor
  - Communication channels
  - Database servers
  - Operating systems
  - Applications
- A TP monitor provides tools, mechanisms, and APIs to ease or automate the application programming, execution, and administration of transactions

Committing a Transaction

- When a connection is created, it is in auto-commit mode
  - **Auto-commit mode** means that each individual SQL statement is treated like a transaction and is automatically committed immediately after it is executed
- However, for a transaction that consists of multiple SQL statements, you need to ensure that the atomicity property is satisfied
  - **Atomicity** means that ALL of the statements within the transaction are executed or that NONE of them is executed

Committing a Transaction (cont)

- What you need to do is simply *disable auto-commit* and explicitly commit the transaction, as in

```
conn.setAutoCommit(false); // at start of transaction
...
...
...
conn.commit();                    // at end of transaction
conn.setAutoCommit(true);
```

- Note that once you disable auto-commit, no SQL statements will be committed until you call the *commit()* method explicitly

Committing a Transaction (cont)

conn.setAutoCommit(false);

```
Statement updateSales = conn.createStatement();
String str = "UPDATE COFFEES SET SALES = 75 " +
    WHERE COF_NAME LIKE 'Columbian' ";
updateSales.executeUpdate(str);

Statement updateTotal = conn.createStatement();
String str = "UPDATE COFFEES SET TOTAL = TOTAL + 75 " +
    "WHERE COF_NAME LIKE 'Columbian' ";
updateTotal.executeUpdate(str);
```

conn.commit();
conn.setAutoCommit(true);

Aborting and Rolling Back a Transaction

- The ***rollback()*** method aborts a transaction and restores the data values to what they were before the update was attempted
- If you try to execute one or more statements in a transaction and you get an ***SQLException***, you should call the *rollback()* method to abort the transaction and then perhaps start the transaction again, after waiting a short amount of time

Aborting and Rolling Back a Transaction (cont)

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

   Statement updateSales = conn.createStatement();
   String str = "UPDATE COFFEES SET SALES = 75 " +
       WHERE COF_NAME LIKE 'Columbian' ";
   updateSales.executeUpdate(str);

   Statement updateTotal = conn.createStatement();
   String str = "UPDATE COFFEES SET TOTAL = TOTAL + 75 " +
       "WHERE COF_NAME LIKE 'Columbian' ";
   updateTotal.executeUpdate(str);

   conn.commit();
   conn.setAutoCommit(true);
}
catch (SQLException) {
        conn.rollback();
};
```