Course Name:

Advanced Java

Lecture 13
Topics to be covered

- The Design of JDBC
- The Structured Query Language
- Basic JDBC Programming Concepts
- Query Execution
- Scrollable and Updatable Result Sets

Introducing JDBC

- According to Sun, JDBC is not an acronym, but is commonly misinterpreted to mean Java DataBase Connectivity
- Supports ANSI SQL 92 Entry Level

The Standard Query Language (SQL)

- Composed of two categories:
  - Data Manipulation Language (DML)
    - used to manipulate the data
      - select
      - delete
      - update
  - Data Definition Language (DDL)
    - create database
    - create table
    - drop database

Data Manipulation Language

- SELECT - query the database
  - select * from customer where id > 1001
- INSERT - adds new rows to a table.
  - Insert into customer values (1009, ‘John Doe’)
- DELTE - removes a specified row
  - delete
- UPDATE - modifies an existing row
  - update customers set amount = 10 where id > 1003

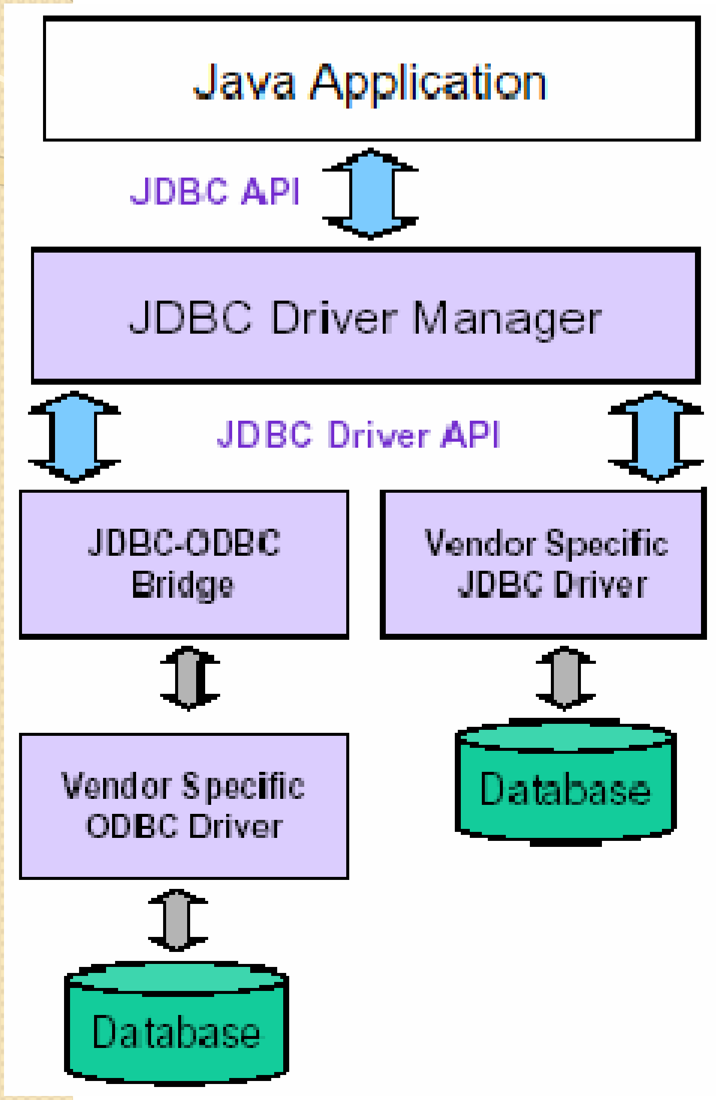
Data Definition Language

- CREATE DATABASE - allows you to create a database
- CREATE TABLE - allows you to create a table definition in a database
- DROP TABLE - removes a table from a database
- ALTER TABLE - modifies the definition of a table in a database

JDBC Framework

- The JDBC driver manager
- The JDBC driver

General Architecture



- What design pattern is implied in this architecture?
- What does it buy for us?
- Why is this architecture also multi-tiered?

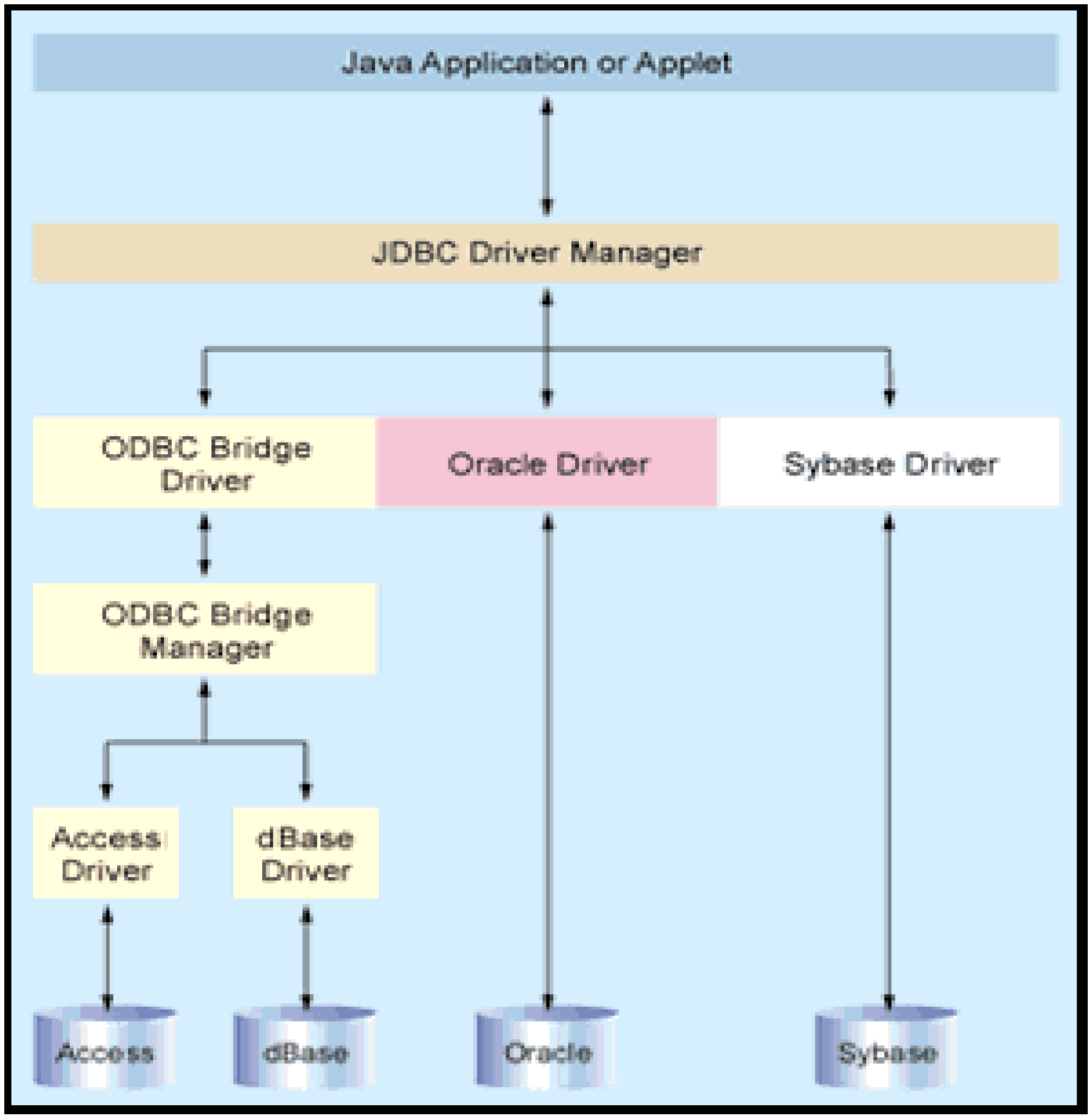


Figure 1. Anatomy of Data Access. The Driver Manager provides a consistent layer between your Java app and back-end database. JDBC works natively (such as with the Oracle driver in this example) or with any ODBC datasource.

The JDBC Driver Manager

- Management layer of JDBC, interfaces between the client and the driver.
- Keeps a hash list of available drivers
- Manages driver login time limits and printing of log and tracing messages
- Secure because manager will only allow drivers that come from local file system or the same initial class loader requesting a connection
- Most popular function:
  - Connection getConnection(url, id, passwd);

JDBC Driver Types

- Type 1 (JDBC-ODBC Bridge Technology)
- Type 2 (JNI drivers for C/C++ connection libraries)
- Type 3 (Socket-level Middleware Translator)
- Type 4 (Pure Java-DBMS driver)

Type 1 Drivers
JDBC-ODBC Bridges

- JDBC driver translates call into ODBC and redirects ODBC call to an ODBC driver on the DBMS
- ODBC binary code must exist on every client
- Translation layer compromises execution speed to small degree

Type 2 Drivers
Native-API + Java Driver

- Java driver makes JNI calls on the client API (usually written in C or C++)
  - eg: Sybase dblib or ctlib
  - eg: Oracle Call Interface libs (OCI)
- Requires client-side code to be installed
- Often the fastest solution available
- Native drivers are usually delivered by DBMS vendor
- bug in driver can crash JVMs
- Example: JDBC=>Sybase dblib or ctlib

Type 3 Drivers
JDBC-Middleware Pure Java Driver

- JDBC driver translates JDBC calls into a DBMS-independent protocol
- Then, communicates over a socket with a middleware server that translates Java code into native API DBMS calls
- No client code need be installed
- Single driver provides access to multiple DBMSs, eg. WebLogic Tengah drivers
- Type 3 drivers auto-download for applets.

Type 4 Drivers
Pure Java Drivers

- Java drivers talk directory to the DBMS using Java sockets
- No Middleware layer needed, access is direct.
- Simplest solution available.
- No client code need be installed.
- Example: JConnect for Sybase
- Type 4 drivers auto-download for applets

Result Sets and Cursors

- Result Sets are returned from queries.
- Number of rows in a RS can be zero, one, or more
- Cursors are iterators that iterate through a result set
- JDBC 2.0 allows for backward as well as forward cursors, including the ability to go to a specific row or a relative row

A JDBC Primer

- First, load the JDBC Driver:
  - call new to load the driver’s implementation of Driver class (redundant--Class.forName does this for you automatically) and call DriverManager.RegisterDriver()
  - add driver to the jdbc.drivers property - DriverManager will load these automatically
    - eg: ~/.hotjava/properties:
      - jdbc.drivers=com.oracle.jdbc.OracleDriver:*etc*;
    - or programatically:
      - String old = sysProps.getProperty(“jdbc.drivers”);
      - drivers.append(“:” + oldDrivers);
      - sysProps.put(“jdbc.drivers”, drivers.toString());
  - call Class.forName and pass it the classname for the driver implementation

Create a Connection to the database vi the driver

- Call the getConnection method on the DriverManager object.
- Connection conn = DriverManager.**getConnection**(url, login, password)
- url: jdbc:*subprotocol*:host:port[/database]
  - registered subprotocol: sybase, odbc, msql, etc.
  - eg: jdbc:sybase:Tds:limousin:4100/myDB
- Only requirement: The relevant Drivers must be able to recognize their own URL

SQL Statements

- Create some form of Statement
  - Statement
    - Represents a basic SQL statement
    - Statement stmt = conn.createStatement();
  - PreparedStatement
    - A *precompiled* SQL statement, which can offer improved performance, especially for large/complex SQL statements
  - Callable Statement
    - Allows JDBC programs access to stored procedures

Execute the Statement

- executeQuery(): execute a query and get a ResultSet back
- executeUpdate(): execute an update and get back an int specifying number of rows acted on
  - UPDATE
  - DELETE
- execute(): execute unknown SQL and returns true if a resultSet is available:
  - Statement genericStmt = conn.createStatement();
  - if(genericStmt.execute(SQLString)) {
    - ResultSet rs = genericStmt.getResultSet(); process(); }
  - else {
    - int updated = genericStmt.getUpdateCount(); processCount();
  - }
    - etc.

Result Sets

- ResultSet rs = stmt.executeQuery(“select id, price from inventory”);
  - rs.next() - go to next row in ResultSet
    - call once to access first row: while(rs.next()) {}
  - getXXX(*columnName*/*indexVal*)
    - getFloat()
    - getInt()
    - getDouble()
    - getString() (highly versatile, inclusive of others; automatic conversion to String for most types)
    - getObject() (returns a generic Java Object)
  - rs.wasNull() - returns true if last get was Null

JDBC 2 – Scrollable Result Set

```
...
Statement stmt =
con.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
                              ResultSet.CONCUR_READ_ONLY);

String query = “select students from class where type=‘not
   sleeping’ “;
ResultSet rs = stmt.executeQuery( query );

rs.previous();  / / go back in the RS (not possible in JDBC 1...)
rs.relative(-5); / / go 5 records back
rs.relative(7); / / go 7 records forward
rs.absolute(100); / / go to 100th record
...
```

JDBC 2 – Updateable ResultSet

```
...
Statement stmt =
con.createStatement(ResultSet.TYPE_FORWARD_ONLY,
                              ResultSet.CONCUR_UPDATABLE);
String query = " select students, grade from class
                    where type='really listening this presentation☺' ";
ResultSet rs = stmt.executeQuery( query );
...
while ( rs.next() )
{
   int grade = rs.getInt("grade");
   rs.updateInt("grade", grade+10);
   rs.updateRow();
}
```

Prepared Statements

- Use for complex queries or repeated queries
- Features:
  - precompiled at database (statement usually sent to database immediately on creation for compilation)
  - supply with new variables each time you call it (repeatedly eg.)
- eg:
  - PreparedStatement ps = conn.prepareStatement(“update table set sales = ? Where custName = ?”);
- Set with values (use setXXX() methods on PreparedStatement:
  - ps.setInt(1, 400000);
  - ps.setString(2, “United Airlines”);
- Then execute:
  - int count = ps.executeUpdate();