Database Management Systems - I, CS 157A

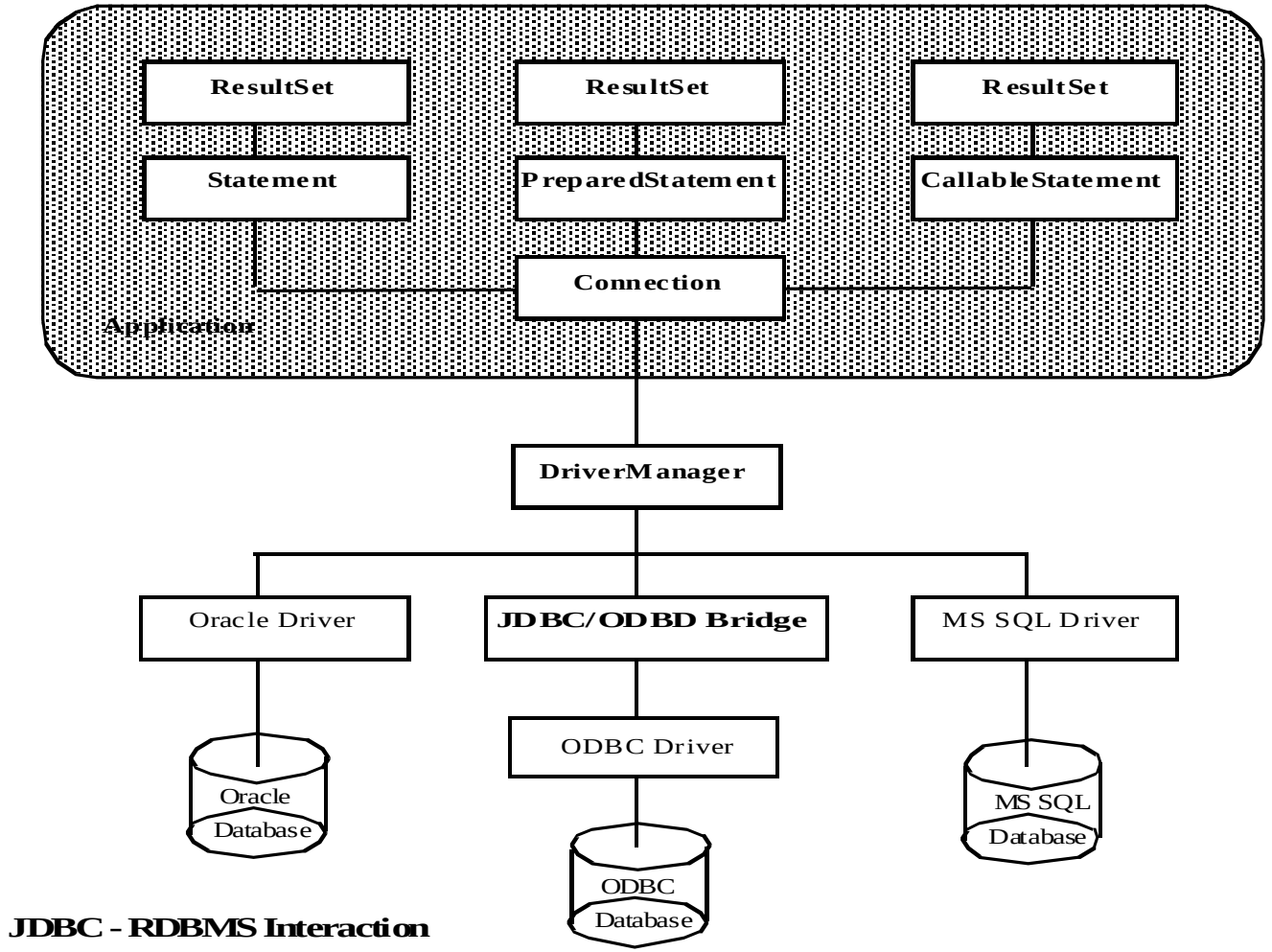
Java Database Connectivity (JDBC)

Agenda

- JDBC Architecture
- SQL Review
- Basic JDBC Database Access
- JDBC Support Classes
- Optimized Database Access
- JDBC 2.0
- JDBC 2.0 Optional Package (javax.sql)

1. JDBC Architecture

Java APIs to Access RDBMS



JDBC – RDBMS Interaction

2. SQL Review

SQL Review

SQL Keyword	Description
CREATE	Create a table
SELECT	Retrieve fields from one or more tables
FROM	Tables from which to get fields
WHERE	Criteria for selection that determine the rows to be retrieved
ORDER BY	Criteria for ordering records/tuples
JOIN	Merge data from multiple tables
INSERT INTO	Insert data into specified table
UPDATE	Update data in a specified table
DELETE FROM	Delete data from a specified table

3. Basic JDBC Database Access

JDBC Basics:

```
Package    class12;
Import     java.sql.*

Public     class Example1 {
            public  static  void  main (java.lang.String[]  args) {
                    try {
                        // This is where we load the JDBC driver (step-1)
                        Class.forName ("sun.jdbc.odbc.JdbcOdbcDriver");
                    }
                    catch (ClassNotFoundException  e) {
                         System.out.println("Unable to load the Driver class");
                          return;
                    }

                    try {
                          // All DB access is within try/catch block (Step 2)
                          // Connect to the database specifying user, password
```

JDBC Basics:

```
            Connection  conn = DriverManager.getConnection(
                                    "jdbc:odbc:comapnydb", "user", "passwd");
             // Create and execute SQL statement (Step 3)
             Statement   stmt = conn.createStatement();
             ResultSet   rs = stmt.executeQuery(
                                "SELECT   First_Name  FROM  EMPLOYEES" );
             // Display the SQL query results
             while ( rs.next() ) {
                   System.out.println(rs.getString("First_Name"));
             }
            // Make sure our DB resources are released (step 4)
            rs.close();
            stmt.close();
            conn.close();
          }
          catch ( SQLException  se) {
            System.out.println ("SQL Exception:" + se.getMessage() );
            se.printStackTrace ( System.out );
          }
      }
}
```

JDBC Drivers

- Before using JDBC driver, you need to register it with the JDBC Driver Manager:

  Class.forName (“sun.jdbc.odbc.JdbcOdbcDriver”);

- **There are 4 categories of JDBC drivers:**
  - **Type-1 JDBC-ODBC Bridge Driver:** uses bridge technology to connect java to an ODBC driver to access the DB. Sun provides a free implementation from Intersolve. Requires s/w installation at the client side.
  - **Type-2 Native-API Partial Java Driver (Client-Server):** java wrapper around the vendor proprietary driver – Oracle OCI.
  - **Type-3 Net-Protocol All Java Driver (3-tiers):** use generic network protocol with pure Java client that interfaces with custom middleware software.
  - **Type-4 native-Protocol All Java Driver (Client-Server thin driver)**: client uses native protocol for the DB written entirely in Java.

JDBC URL Format

- JDBC driver uses URL to identify and connect to a particular database:

  Sun recommendation: jdbc:driver:databaseName

  Oracle: jdbc:oracle:thin@site:port:databaseName

  The URL format is flexible; the main requirement is that a driver recognizes its own URL.

Connecting to the Database

- The primary usage of the DriverManager is to get connections; there are two flavors for getting connections:

```
public static Synchronized Connection  getConnection (
        String url, Properties  p) throws SQLException;

public static Synchronized Connection getConnection (
        String url, String user, String psswd) throws SQLException

Connection  conn= null;
conn = DriverManager.getConnection (
          “jdbc:oracle:thin://nest.us.oracle.com:1521:oradb7”, “Scott”,
          “tigger” );
```

Connecting to the Database

- The Driver interface implemented by the vendor also allows you to acquire a database connection. It is not considered as clean as it bypasses the DriverManager and as a result it may bypasses security checks:

```
public static  Connection  Connect (
                String url, Properties  p) throws SQLException;

public static  Connection Connect (
                String url, String user, String psswd) throws SQLException
```

When a connection is no longer needed, it should be explicitly closed (**expensive resource**). If you close() a connection before committing (auto-commit is off), any uncommitted statements will be lost.

Connecting to the Database

- JDBC 2.0 standard extensions (optional package) provides a facility for connection pooling where an application can maintain several open connections and spread the load among them.

Basic Database Access

Basic RDBMS operations include queries (SELECT) and DML (INSERT, UPDATE, DELETE, CREATE, DROP) operations. Once you created a *connection*, you can create a *statement* object to use for executing SQL statements and the basic operations.

JDBC uses different methods for sending queries and updates. Also, queries return an instance of *jave.sql.ResultSet* while non-queries return an integer. The *ResultSet* object provides you with methods to retrieve the query results.

```
Package    class12;
Import      java.sql.*

Public      class Example2 {
```

Basic Database Access

```
public  static  void  main (java.lang.String[]  args) {
        try {
            // This is where we load the JDBC driver (step-1)
            Class.forName ("sun.jdbc.odbc.JdbcOdbcDriver");
        }
        catch (ClassNotFoundException  e) {
             System.out.println("Unable to load the Driver class");
              return;
        }

        try {
            // All database access is within a try/catch block. Connect
            // to a DB specifying particular database, user and passwd
            Connection  conn = DriverManager.getConection (
                              "jdbc:odbc:comapnydb", "user", "password");
```

Basic Database Access

```
            // Create and execute SQL statement
            Statement  stmt = conn.createStatement( );

             String  test_id = args[0];
             String  test_value = args[1];
             int  update_cnt = stmt.executeUpdate(
                 "INSERT  INTO  t_test(test_id, test_value)" +
                 "VALUES ( 'test_id', 'test_value' )";
             System.out.println (update_cnt + "rows inserted.");
             stmt.cose( );
        }
        catch (Exception e)
             e.printStackTrace( );
        finally {
             try { conn.close( ); }
             catch (SQLException  se)
                  System.out.println(se.printStackTrace( ) );
        }
    }
}
```

3.1 Query Example

```
try {
    // All database access is within a try/catch block.
    Connection  conn = DriverManager.getConection (
                                "jdbc:odbc:comapnydb", "user", "password");
    // Create and execute SQL statement
    Statement  stmt = conn.createStatement( );
    ResultSet   rs = stmt.executeQuery(
      "SELECT  NAME, CUSTOMER_ID, PHONE FROM CUSTOMERS");
     // Display Query results
     while ( rs.next( ) ) {
            System.out.println("Customer# +
                                    rs.getString("CUSTOMER_ID") );
            System.out.println("," + rs.getString("NAME") );
            System.out.println(", is at " + rs.getString("PHONE") );
     }
     rs.close( );
     stmt.cose( );
}
    }
}
```

3.1 Query Example

ResultSet class provides a set of **getxxx**( ) methods including **getObject**( ) which returns any kind of data packaged in an object wrapper. For example, calling **getObject**() on an *integer* field returns an *Integer* object (i.e., wrapper around integer).

If you don't know whether a SQL statement is a Query or a DML statement, you should use the **execute**( ) method of the *statement* object. It returns TRUE if it was a query, otherwise it returns FALSE for DML statement.

3.1 Query Example

```
                    // Create and execute SQL statement
                    Statement  uSQL = conn.createStatement( );
                    if ( uSQL.execute (sqlString) ) {
                            // The stmt was a SELECT query
                            ResultSet  rs = uSQL.getResultSet( );
                            // Display results
                    }
                    else {
                            // The stmt was a DML stmt
                            System.out.println("Rows Updated:" +
                                                    uSQL.getUpdateCount( ) );
                    }
                    ……
                    ……
                    rs.close( );
                    stmt.cose( );
            }
    }
}
```

3.2 Multiple Result Sets

It is possible to write a SQL statement that returns more than one result set or update count (this is rare). For example, stored procedure or some vendors allow multiple statements to be executed in a batch:

```
// Create and execute SQL statement
Statement  uSQL = conn.createStatement( );
uSQL.execute (sqlString);
while (true ) {
        ResultSet  rs = uSQL.getResultSet( );
        int  i = uSQL.getUpdateCount( );
        if ( rs != null )
                // sqlString was a SELECT stmt - Display results
        else
                // Process the update count

        // Advance and quit if done!
        If ( (rs == false) && (i == -1) )
                break;
}
……
```

3.3 Handling NULLs:

Table columns may contain null (empty) value. This is a problem if you are not reading the column of the result set as an object:

```
// null example
Object  num = rs.getObject ("STOCK");
 if ( num == null)
        System.out.println( "Result was null" );
```

JDBC *ResultSet* object includes the **wasNull**( ) method which indicates whether or not last column read was null:

```
Integer  num = rs.getInt ("STOCK" );
// check if num is NULL
if ( rs.wasNull( ) )
        System.out.println( "Result was null" );
else
        System.out.println( "Number:" + num );
```

3.4 Large Data Types

You can retrieve large chunk of data (e.g., an image) from the result set as a stream using getAsciiStream( ), getBinaryStream( ), and getUnicodeStream( ). Each of these methods return *InputStream*.

```
ResultSet  rs  = stmt.executeQuery( "SELECT  IMAGE
                                    FROM    PICTURES
                                    WHERE PID = "+ req.getParameter ("PID") );
if ( rs.next( ) ) {
        BufferedInputStream  gifData  = new
            BufferedInputStream( rs.getBinaryStream( "IMAGE" ));
byte[]  buf = new byte[4096];
int       len;
while ( len = gifData.read ( buf, 0, buf.length)) != -1 )
        out.write ( buf, 0, len );
}
```

JDBC 2.0 API includes BLOB and CLOB objects to handle large data types.

4. JDBC Support Classes

4.1 Date and Time

JDBC supports 3 classes: *java.sql.Date*, *java.sql.Time* and *java.sql.TimeStamp*. These correspond to SQL *DATE*, *TIME* and *TIMESTAMP* data types. These classes are wrappers around the *java.util.Date* class to fit the JDBC model.

Since different RDBMS packages/vendors have different encoding to date and time information, JDBC supports ISO date escape sequence:

Date: { 'yyyy-mm-dd' }
time: { 'hh:mm:ss' }
time stamp: { 'yyyy-mm-dd hh:mm:ss:ms:microsecond:ns' }

Ex) **stmt**.execute("**INSERT INTO** FRIENDS(BIRTHDAY)
VALUES ({d '1978-12-14'}**)**") ;

4.2 Handling Errors

JDBC extends the *java.lang.Exception* class (*java.sql.SQLException*) provides extra information about the database errors:

- The SQLState string describing the error according to XOpen SQLState convention.
- The database specific vendor error code.
- A chain of exceptions leading up to this one.

```
try {
        Connection  conn = DriverManager.getConnection(
                        "jdbc:odbc:companydb", "user", "passwd" );
        …..
}
catch (SQLException  se) {
        se.printStackTrace( );
        while ( (se.getNextException( ) !=  NULL )
                se.printStackTrace( );
}
```

4.3 SQL Warning

Non fatal errors generate *SQLWarning* which is an extension to *SQLException*. When a JDBC object encounters a warning internally, it creates a *SQLWarning* object and adds it to a list of warnings it keeps. At any point, you can get these warnings by repeatedly calling getWarning( ) method until it returns null.

```
ResyltSet  rs = stmt.executeQuery( “SELECT  *  FROM  CUSTOMERS” );
SQLWarning  stmtW = stmt.getWarnings( );
while (stmtW  !=  null ) {
        System.out.println( “\nSQL Warning: “);
        System.out.println(stmtW.getMessage( ) );
        System.out.println( “ANSI-92 SQL State: “ +  stmtW.getSQLState( ) );
        System.out.println(“Vendor Error Code: “ + stmtW.getErrorCode( ) );
        statW = statW.getNextWarning( );
}
// Same for result set
SQLWarning   rsW  =  rs.getWarnings( );
```

A *DataTruncation* class is a special kind of warning. A *DataTruncation* object is chained as a warning on **Read** and is thrown an exception on **Write** operation.

5. Optimized Database Access

Optimized Database Access

Instead of writing the SQL statement and embed it in java code and send it to the database, it is better to store the SQL statements in the database (stored procedure) and call it by name when needed!

1. Prepared Statement
2. Stored Procedure – Callable Statement
3. Transactions
4. Dynamic Access

5.1 Prepared Statement

Every time JDBC sends a SQL statement to the RDBMS server, the database interpret it, builds a query plan (expensive operation), then executes the plan. By sending SQL as a prepared statement, you allow the database to build the plan once (avoid the overhead for subsequent calls). JDBC prepared statement does not guarantee that the underlying RDBMS engine provide the above optimization.

```
statement  stmt = conn.createStatement( );
int              i;


for ( i = 0 ; i < accounts.length ; i++ ) {
     stmt.executeUpdate( “UPDATE  accounts” +
                                   “SET  balance accounts[i].getBalance( ) +
                                   “WHERE  id  = “ + accounts[i].getId( ) );
}
conn.commit( );
stmt.close( );
```

5.1 Prepared Statement ☺

The above code sample creates the same query plan in every loop iteration ☻ :

```
Preparedstatement  pstmt = conn.prepareStatement( "UPDATE  accounts"
                                    + "SET balance = ?" + "WHERE  id  = ?" );
int              i;


for ( i = 0 ; I < accounts.length ; i++ ) {
     pstmt.setFLOAT( 1, accounts[i].getBalance( ) );
     pstmt.setInt( 2, accounts[i].getId( ) );
     pstmt.executeUpdate( );
}
conn.commit( );
pstmt.close( );
```

5.2 Stored Procedure (SP) – Callable Statement

The *callableStatement* interface in the JDBC object is the interface that supports stored procedures:

- **Stored procedures** are precompiled and as a result they run faster. Also, it eliminates many network round trips with potential unnecessary data traffic over the network.
- **Syntax errors** in the stored procedure can be caught at compile time rather than at run time.
- Developer needs to know only the name of the procedure and its In/Out parameters. Procedure implementation is invisible!

Different RDBMS use different syntax for stored procedure calls. JDBC provides database-independent SP escape syntax in the form { ? = call **proc-name** [(? [, ? …])] }.

5.2 Stored Procedure (SP) – Callable Statement

Each ? Represents a place holder for either input or output/return value. The JDBC driver translates this escape sequence into the database specific procedure syntax.

```
CallableStatement  cstmt = conn.prepareCall( " {call sp_interest(?, ?)}" );

cstmt.setInt( 1, accountId );
cstmt.setFloat( 2, 2345.23 );          ← input argument
cstmt.registerOutParameter( 2, Types.FLOAT );  → output argument
cstmt.execute( );

System.out.println( "New balance: "  cstmt.getFloat( 2) );
```

P.S. 2nd argument is an In/Out argument

5.3 Transactions

A transaction is a group of operations that must behave atomically. Transaction can operate at varying levels of isolation from the rest of the database. At the most isolated level, the result of the group of operations within the transaction become visible to the rest of the system only when the transaction is committed.

```
try {
        conn.setAutoCommit( false );
        stmt.executeUpdate( "……" );   ← Begin tx
        stmt.executeUpdate( "……." );
        conn.commit( );
}
catch ( SQLException  se )
        conn.rollback( );
```

5.3 Transactions

When auto-commit is set to false, you must call either commit() or rollback() at the end of the transaction or you changes will be lost. JDBC supports 5-levels of isolation, i.e., who see what and when! Some levels might not be supported by a specific RDBMS:

- TRANSACTION_NONE: tx is disabled
- TRANSACTION_READ_UNCOMITTED: allow dirty reads.
- TRANSACTION_READ_COMITTED: dirty read are not allowed.
- TRANSACTION_REPEATABLE_READ: protect against repeatable reads as well as dirty reads. T_1:READ → T_2:READ + WRITE + Commit → T_1:READ!
- TRANSACTION_SEARIALIZABLE: supports TRANSACTION_REPEATABLE_READ and guards against row insertion as well.

You can use the *DatabaseMetaData* class to find out the isolation levels supported by a specific database.

5.4 Dynamic Access

Some applications need to dynamically discover information about the underlying database schema or about the result set. This information is called metadata. JDBC supports both: *DatabaseMetaData* and *ResultSetMetaData*.

- ***Java.sql.DatabaseMetaData***: This class answers the following questions:
  - What tables exists in the database?
  - What user name is being used by this connection?
  - Is this database connection is RO?
  - Does the database support column aliasing?
  - Are multiple result sets from a single execute() is supported
  - What are the primary keys for a table?

5.4 Dynamic Access

- ***Java.sql.ResultSetMetaData***: This class provides answers to the following questions:
  - How many columns are in the result set?
  - Are column names case-sensitive?
  - Can you search on a given column?
    Is NULL a valid value for a given column?
  - How many characters is the max display size for a given column?
  - What is the name of a given column?
  - What table did a given column came from?
  - What is the data type of a given column?

5.4 Dynamic Access

```
Public  Vector executeSQL (String  sql) {
        Vector   v = new Vector( );
        try {
             Connection  conn = DriverManager.getConection (
                                        “jdbc:odbc:comapnydb”, “user”, “password”);
            // Create and execute SQL statement
            Statement  stmt = conn.createStatement( );
            if (stmt.execute( sql ) ) {
                // true means SELECT stmt
                ResultSet   rs = stmt.getResultSet( );
                ResultSetMetaData  rsmd = rs.getMetaData( );
                int   cols;
                cols = rsmd.getColumnCount( );
                while ( rs.next( ) ) {
                    Hashtable  h = new  Hashtable( cols );
                    int  i;
```

5.4 Dynamic Access

```
                    for ( i = 0 ;  i < cols ; i++ ) {
                        Object   obj = rs.getObject( i );
                        h.put( rsmd.getColumnLabel( i ), obj );    <name, value>
                    }
                    v.addElement( h );
                }
                return v;
            }
            // False means DML stmt
            return null;
        }
        catch ( SQLException  se ) {
            se.printStackTrace( );
            return null;
        }
}
```

6. JDBC 2.0

6. JDBC 2.0

- **Result Handling**:
  - With earlier JDK, the *resultSet* interface provides no support for updates, access to rows is limited to a single, sequential read, and no going back. JDBC 2.0 supports scrollable and updatable *ResultSet*, which allows advanced record navigation and in-place manipulation.
  - **Scrolling**: JDBC 2.0 supports 3 distinct types of *ResultSet* objects: forward-only, scroll-sensitive and scroll-insensitive. Scroll-insensitive *ResultSet* does not reflect changes to the underlying database, while scroll-sensitive one does.

6. JDBC 2.0

- **Updatable:** JDBC 2.0 supports 2-types of *ResultSet* objects: read-only and updatable result sets.

  Both scrollable and updatable capabilities are not mandatory in JDBC 2.0. You can find for a specific JDBC 2.0 driver if they are supported from the *DatabaseMetaData* object.

- **createStatement()** without argument will result in a forward-only and read-only *ResultSet* (i.e., like JDBC 1.0).

- **Example** of scroll-sensitive and updatable resultset:

```
Statement stmt = conn.createStatement(
                                ResultSet.TYPE_SCROLL_SENSITIVE,
                                ResultSet.CONCUR_UPDATABLE);
```

6. JDBC 2.0

JDBC 2.0 Record Scrolling Functions

Method	Description
first( )	Move to the first record
last( )	Move to the last record
next( )	Move to the next record
previous( )	Move to the previous record
beforefirst( )	Move to immediately before the first record
afterLast( )	Move to immediately after the last record
absolute(int)	Move to an absolute row number. Takes a positive or negative argument
relative(int)	Move backward or forward a specified number of rows. Takes a positive or negative argument.

6. JDBC 2.0

JDBC 2.0 *ResultSet* provides additional methods: isFirst(), isLast(), isBefore() and isAfterLast(). You can update/insert a row into an updatable *ResultSet*:

```
Statement stmt = conn.createStatement(
                              ResultSet.TYPE_SCROLL_SENSITIVE,
                              Resultset.CONCUR_UPDATABLE );
ResultSet  rs = stmt.executeQuery( "SELECT  NAME, CUSTOMER_ID
                                    FROM  CUSTOMERS" );
// Update
rs.first( );                          // goto the tuple to change
rs.updateInt( 2, 35243 );             // change value of CUSTMER_ID
rs.updateRow( );                      // Make the change in the DB
```

6. JDBC 2.0

```
// Insert…
rs.moveToInsertRow( );                  // create a blank row w/o data
rs.updateString( 1, “Tom Jones” );      // change field in the blank row
rs.updateInt( 2, 35244 );               // change field in the blank row
rs.insertRow( );                        // append the new row to both the
                                        // ResultSet and to the database
```

For insert row, you need to update all columns that do not accept null values, otherwise on insertRow( ) you will get *SQLException*. Deleting a row from an updatable *ResultSet* is simple: move to the desired row then call deleteRow( );

6. JDBC 2.0

- **Batch Updates**:
  - Can be used with SQL update statement (e.g., CREATE, DROP, INSERT, UPDATE, DELETE) and not with queries (SELECT):
  - **Statement:**

```
conn.setAutoCommit( false );
Statement  stmt  =  conn.createStatement( );
stmt.addBatch( "INSERT INTO CUSTOMERS VALUES ( 1, "J. Smith", "617-555-1323" );
stmt.addBatch( "INSERT INTO CUSTOMERS VALUES ( 2, "A. Smith", "617-555-1132" );
stmt.addBatch( "INSERT INTO CUSTOMERS VALUES ( 3, "C. Smith", "617-555-1238" );

// stmt.clearBatch( );          // remove above statements from the pending batch job
int[]  upCounts = stmt.executeBatch( );  // returned int[] tells how many records are
                                         // affected by each SQL stmt in the Batch
conn.commit( );
```

6. JDBC 2.0

- **Prepared Statement:**

```
conn.setAutoCommit( false );
PreparedStatement  pstmt  =  conn.prepareStatement(
                              "INSERT INTO CUSTOMERS VALUES ( ?, ?, ? )" );
pstmt.setInt( 1, 1 );
pstmt.setString( 2, "J. Smith" );
pstmt.setString( 3, "617-555-1323" );
pstmt.addBatch( );

pstmt.setInt( 1, 2 );
pstmt.setString( 2, "A. Smith" );
pstmt.setString( 3, "617-555-1132" );
pstmt.addBatch( );

int[]    upCounts =  pstmt.executeBatch( );
conn.commit( );
```

6. JDBC 2.0

- **Callable Statement:**
  Each stored procedure must return an update-count and may take any OUT or INOUT parameters.

- **Java Aware Database:**
  If the database is an OODBMS and it provides support to store and retrieve Java objects, then you can use getObject( ) and setObject( ) methods as follows:

```
ResultSet  rs = stmt.executeQuery(
                        "SELECT ACCOUNT FROM ACCOUNTS" );
rs.next( );
Account  a  = (Account) rs.getObject( );
```

6. JDBC 2.0

```
// Store an object
Account  a  = new Account( );

// Fill in “a” appropriate fields …
PreparedStatement  pstmt = conn.prepareSatement(
            INSERT INTO ACCOUNTS (ACCOUNT) VALUES (?)” );

pstmt.setObject( 1, a );
pstmt.executeQuery( );
```

6. JDBC 2.0

- **BLOBs and CLOBs:**
  Binary Large Objects (**BLOB**s) and Character Large Objects (**CLOB**s) store large amount of binary or character data. Physical RDBMS have different names for these data types.

  JDBC 2.0 ResultSet interface include: getBlob() and getClob() methods. The Blob and Clob objects allow access to the data via streams. getBinaryStream( ) and getCharacterStream( ) methods or through direct-read methods: getBytes( ) and getSubString( ). With PreparedStatement, you have additional setBlob( ) and setClob( ) methods. Also, you have updateBlob( ) and updateClob( ) methods.

  P.S. Lifespan of a **Blob** or **Clob** objects is linked to the transaction that created them.

7. JDBC 2.0 Optional Package (javax.sql)

7. JDBC 2.0 Optional Package (javax.sql)

- **DataSource**:
  This is a new way for an application to obtain a database connection:

```
try {
        InitialContext   ic = new InitialContext( );
        DataSource     ds = (DataSource)  ic.lookup(
                                  “java.comp/env.jdbc/books” );
        Connection  conn = ds.getConnection( );
        ….
```

7. JDBC 2.0 Optional Package (javax.sql)

- **Connection Pooling**:
  JDBC 2.0 drivers that support connection pooling implements the *ConnectionPoolDataSource* and *PooledConnection* interfaces:

```
try {
        InitialContext   ic = new InitialContext( );
        ConnectionPoolDataSource     cpds =
                              (ConnectionPoolDataSource)  ic.lookup( … );
        PooledConnection  pc = cpds.getPooledConnection( );
        Connection   conn = pc.getConnection( );
        Statement….
        PreparedStatement…..
        CallableStatement…..
```

7. JDBC 2.0 Optional Package (javax.sql)

- **RowSet**:
  Rowset inteface extends the *ResultSet* interface and is not implemented as part of the RDBMS driver. Instead it is implemented as *JavaBeans* that encapsulates the tabular data source.

  Rowset interface allows scrolling and updatable operations even if the underlying RDBMS does not support these operations. *Rowset* allows disconnecting from the RDBMS, update the data in the *Rowset*, then connect back to the RDBMS and update it (disconnected *RowSet*).

7. JDBC 2.0 Optional Package (javax.sql)

Unlike *ResultSet*, *Rowsets* are serializable objects so they can be saved locally or transmitted over the network. Also, *RowSets* support *JavaBeans* event model (RowSetListener interface and RowSetEvent class) that enable applications to be notified when the *RowSet* cursor moves, or a record is inserted or updated or deleted, etc.).

Sun provides three *RowSet* implementations:

- ***CachedRowSet* class**: defines a disconnected *Rowset*.
- ***WebRowSet* class**: subclass of *CachedRowSet* that allows *RowSet* data to be output as an XML document.
- ***JDBCRowSet* class**: defines a connected *RowSet* that encapsulates a *ResultSet* but looks like *JavaBean*.

7. JDBC 2.0 Optional Package (javax.sql)

- **Distributed Transaction Support (JTA)**:
  Allowing the JDBC driver to utilize the standard 2PC protocol provided by **JTA** that is part of the **EJB** infrastructure.

END