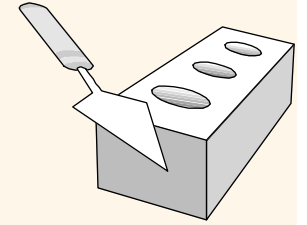


Database Application Programming

Excerpt from
“Introduction to JDBC” by Marty Hall and Larry Brown

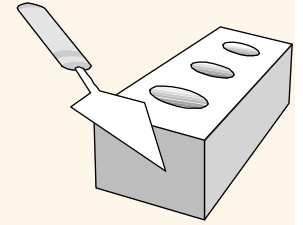
Agenda



- ❖ Database Application Programming Overview

- ❖ Introduction to JDBC Technology
 - JDBC drivers
 - Seven basic steps in using JDBC
 - Retrieving data from a ResultSet
 - Using prepared and callable statements
 - Handling SQL exceptions
 - Submitting multiple statements as a transaction

Database Application Architecture



❖ Client-Server Architectures:

▪ 2-Tier: Client and Data-Server

▪ 3-Tier:

• Tier 1: Client

- User interface : responsible for user interaction and data presentation

• Tier 2: Application-Server

- Middleware : protects the data from direct access by the clients

• Tier 3: Data-Server

- DB server : responsible for data storage

❖ Clear separation of user-interface-control and data presentation from application-logic.

❖ Boundaries between tiers are logical. It is quite easily possible to run all three tiers on the same (physical) machine.

3-Tier Architecture

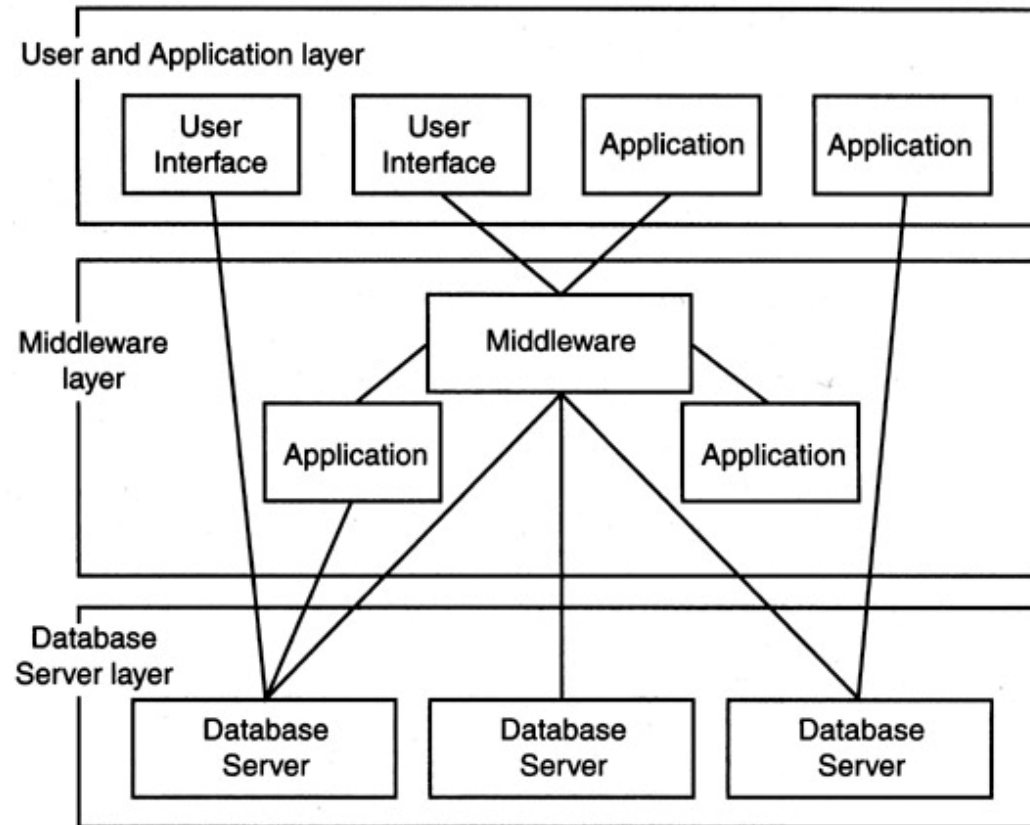
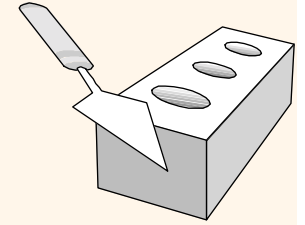


FIGURE 8.1
A variety of client-server architectures for information systems

3-Tier Architecture: Example

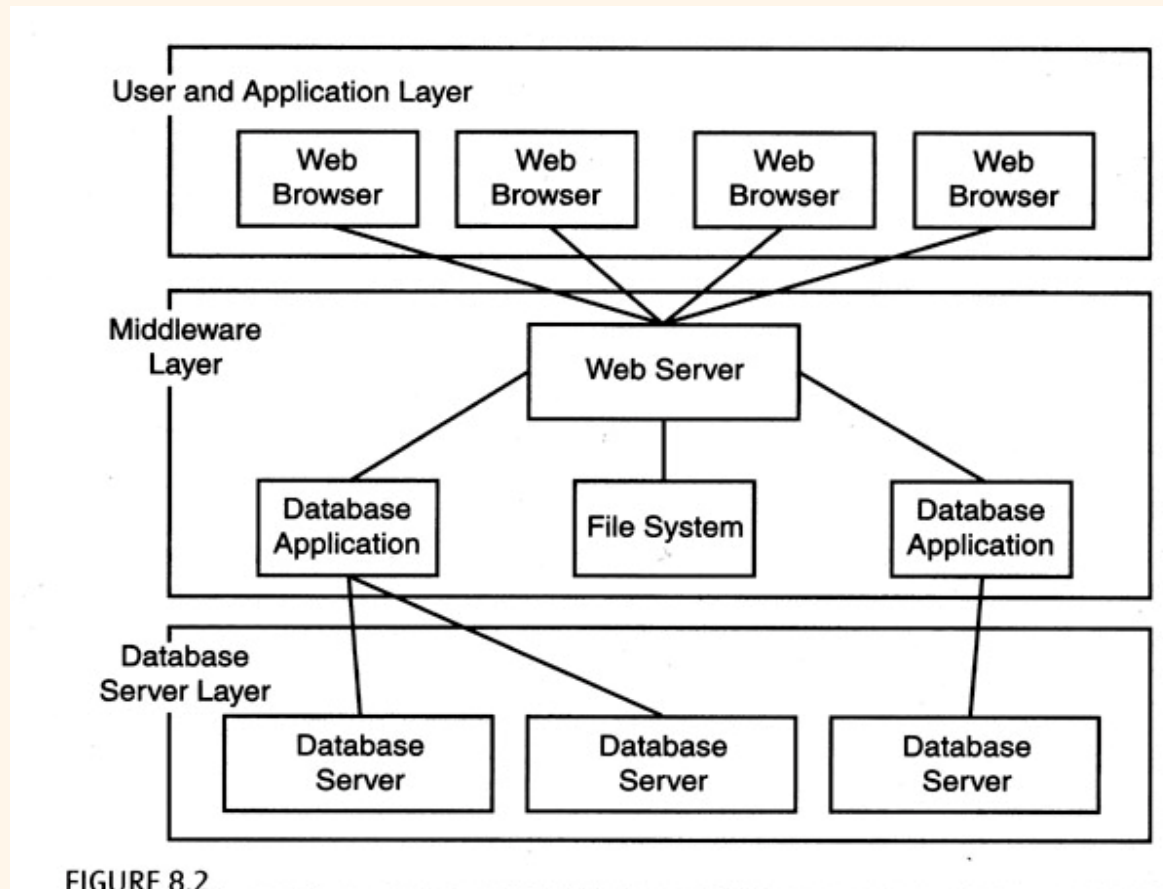
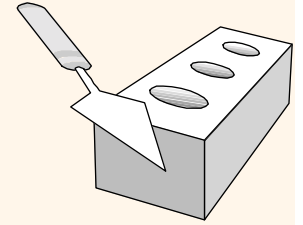
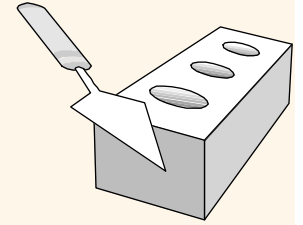


FIGURE 8.2
Architecture of a Web site supported by databases

How to Interact with Database: ODBC



❖ ODBC :

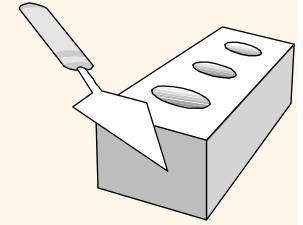
■ ODBC (Open Database Connectivity)

- Provides a way for the client programs to access a wide range of databases and data sources

■ ODBC stack

- ODBC Application :Visual Basic, Excel, Access, ...
- Driver Manager :ODBC.DLL
- ODBC Driver :ODBC drivers vary for various data sources
- Database Transport :Database transport
- Network Transport :TCP/IP or other communication protocols
- Data Source :Oracle, MySQL, ...

Interaction Set-up



- ❖ Making data source available to ODBC:
 - Install ODBC driver manager
 - Install specific driver for a data source (e.g., a DB server)
 - Register the data source driver to the ODBC driver manager
- ❖ How application works with data source:
 - Contacts driver manager to request for specific data source
 - Manager finds appropriate driver for the data source

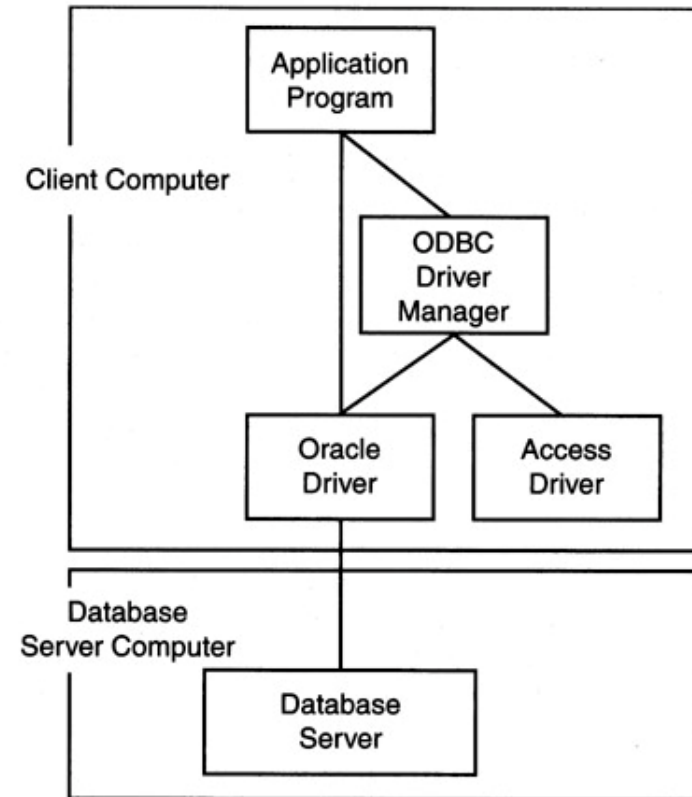
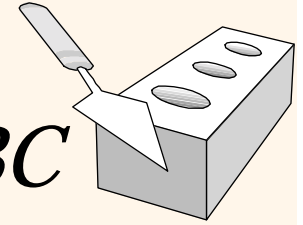


FIGURE 8.3
The ODBC architecture for database access

How to Interact with Database in Java: JDBC

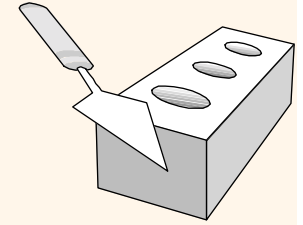


❖ JDBC provides a standard library for accessing relational databases

- API standardizes
 - Way to establish connection to database
 - Approach to initiating queries
 - Method to create stored (parameterized) queries
 - The data structure of query result (table)
 - Determining the number of columns
 - Looking up metadata, etc.
- API does *not* standardize SQL syntax
- JDBC class located in `java.sql` package

Note: JDBC is not officially an acronym; unofficially, “Java Database Connectivity” is commonly used

JDBC Usage Strategies



- ❖ JDBC-ODBC bridge
 - Con: ODBC must be installed
- ❖ JDBC database client
 - Con: JDBC driver for each server must be available
- ❖ JDBC middleware client
 - Pro: Only one JDBC driver is required
 - Application does not need direct connection to DB (e.g., applet)

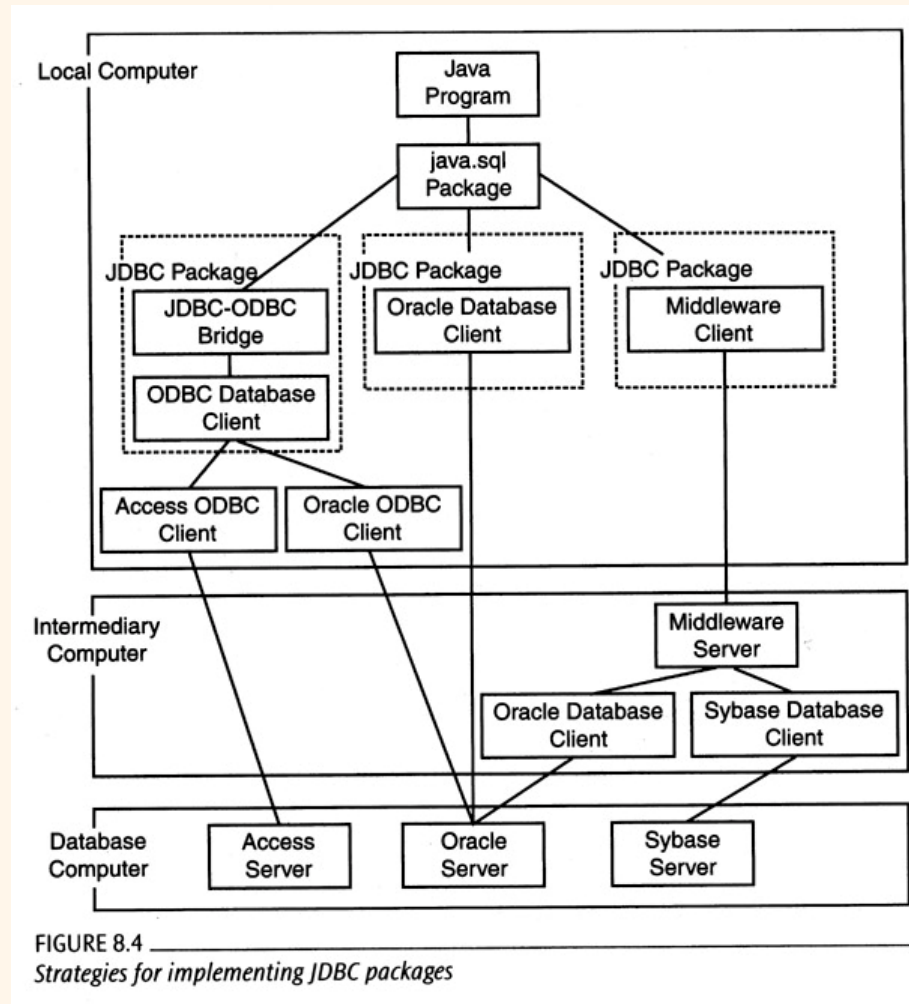
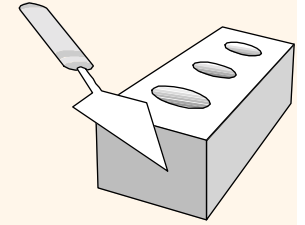
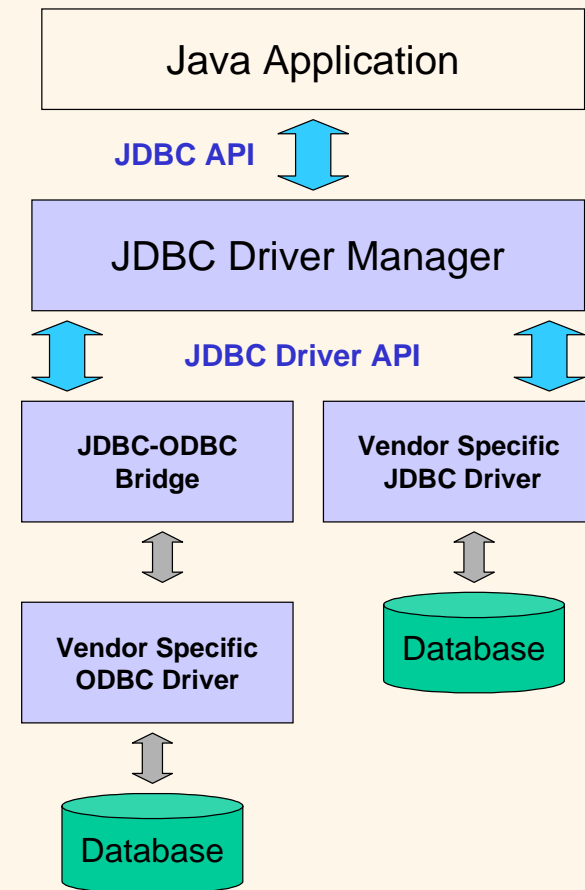


FIGURE 8.4 Strategies for implementing JDBC packages

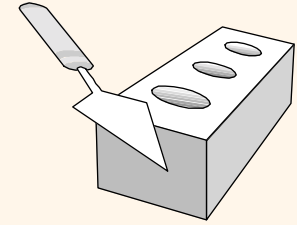
JDBC Components



- ❖ JDBC consists of two parts:
 - JDBC API, a purely Java-based API
 - JDBC Driver Manager, which communicates with vendor-specific drivers that perform the real communication with the database
 - Translation to vendor format is performed on the client
 - No changes needed to server
 - Driver (translator) needed on client



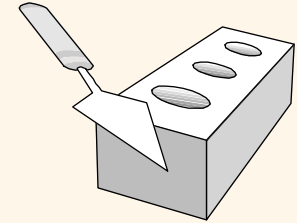
JDBC API



❖ Collection of interfaces and classes:

- DriverManager: Loads the driver
- Driver: Creates a connection
- Connection: Represents a collection
- DatabaseMetaData: Information about the DB server
- Statement: Executing queries
- PreparedStatement: Precompiled and stored query
- CallableStatement: Execute SQL stored procedures
- ResultSet: Results of execution of queries
- ResultSetMetaData: Meta data for ResultSet

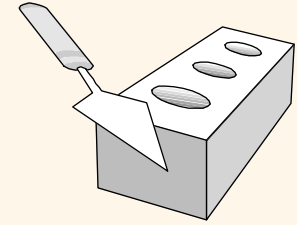
JDBC Data Types



JDBC Type	Java Type
BIT	boolean
TINYINT	byte
SMALLINT	short
INTEGER	int
BIGINT	long
REAL	float
FLOAT DOUBLE	double
BINARY VARBINARY LONGVARBINARY	byte[]
CHAR VARCHAR LONGVARCHAR	String

JDBC Type	Java Type
NUMERIC DECIMAL	BigDecimal
DATE	java.sql.Date
TIME TIMESTAMP	java.sql.Timestamp
CLOB	Clob*
BLOB	Blob*
ARRAY	Array*
DISTINCT	mapping of underlying type
STRUCT	Struct*
REF	Ref*
JAVA_OBJECT	underlying Java class

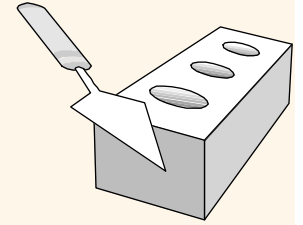
*SQL3 data type supported in JDBC 2.0



Seven Basic Steps in Using JDBC

1. Load the driver
2. Define the Connection URL
3. Establish the Connection
4. Create a Statement object
5. Execute a query
6. Process the results
7. Close the Connection

JDBC: Details of Process



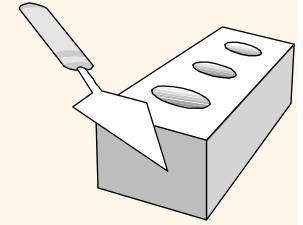
1. Load the Driver

```
try {
    Class.forName("oracle.jdbc.driver.OracleDriver");
    Class.forName("com.mysql.jdbc.Driver");
} catch { ClassNotFoundException cnfe) {
    System.out.println("Error loading driver: " cnfe);
}
```

2. Define the Connection URL

```
String host = "dbhost.yourcompany.com";
String dbName = "someName";
int port = 1234;
String oracleURL = "jdbc:oracle:thin:@" + host +
    ":" + port + ":" + dbName;
String mysqlURL = "jdbc:mysql://" + host +
    ":" + port + "/" + dbName;
```

JDBC: Details of Process (cont'd)

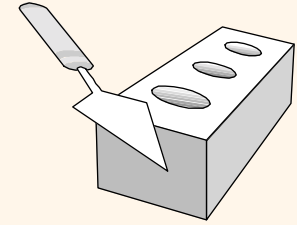


3. Establish the Connection

```
String username = "jay_debese";
String password = "secret";
Connection connection =
    DriverManager.getConnection(oracleURL,
                               username,
                               password);
```

❖ Optionally, get information about the db system

```
DatabaseMetaData dbMetaData = connection.getMetaData();
String productName =
    dbMetaData.getDatabaseProductName();
System.out.println("Database: " + productName);
String productVersion =
    dbMetaData.getDatabaseProductVersion();
System.out.println("Version: " + productVersion);
```



JDBC: Details of Process (cont'd)

4. Create a Statement

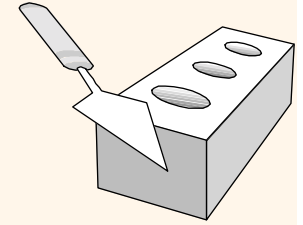
```
Statement statement = connection.createStatement();  
// discuss PreparedStatements later
```

5. Execute a Query

```
String query = "SELECT col1, col2, col3 FROM sometable";  
ResultSet resultSet = statement.executeQuery(query);
```

- To modify the database, use `executeUpdate`, supplying a string that uses `UPDATE`, `INSERT`, or `DELETE`
- Use `statement.setQueryTimeout` to specify a maximum delay to wait for results

JDBC: Details of Process (cont'd)



6. Process the Result

```
while(resultSet.next()) {  
    System.out.println(resultSet.getString(1) + " " +  
                        resultSet.getString(2) + " " +  
                        resultSet.getString(3));  
}
```

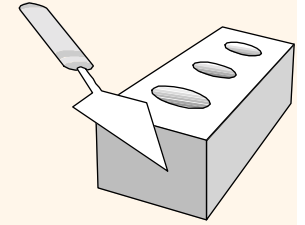
- First column has index 1, not 0
- **ResultSet** provides various **getXxx** methods that take a column index or name and returns the data

7. Close the Connection

```
connection.close();
```

- As opening a connection is expensive, postpone this step if additional database operations are expected

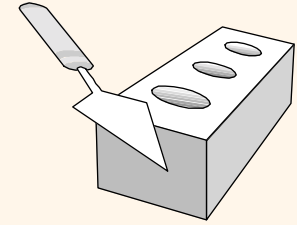
Basic JDBC Example



```
import java.sql.*;

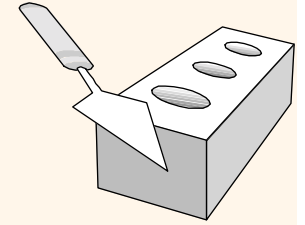
public class TestDriver    {
    public static void main(String[] Args)    {
        try {
            Class.forName("com.mysql.jdbc.Driver").newInstance();
        } catch (Exception E) {
            System.err.println("Unable to load driver.");
            E.printStackTrace();
        }
        try {
            Connection C = DriverManager.getConnection(
                "jdbc:mysql://almaak.usc.edu:3307/menagerie",
                "root", "xyz"); //?user=root&password=xyz");
        }
    }
}
```

Basic JDBC Example (cont'd)



```
Statement s = C.createStatement();
String sql="select * from pet";
s.execute(sql);
ResultSet res=s.getResultSet();
if (res!=null) {
    while(res.next()){
        System.out.println("\n"+res.getString(1)
            + "\t"+res.getString(2));
    }
}
c.close();
}
catch (SQLException E) {
    System.out.println("SQLException: " + E.getMessage());
    System.out.println("SQLState:      " + E.getSQLState());
    System.out.println("VendorError:  " + E.getErrorCode());
}
}
```

ResultSet



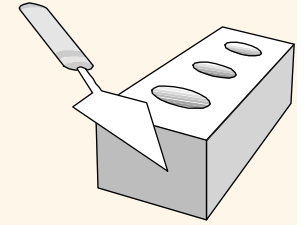
❖ Overview

- A **ResultSet** contains the results of the SQL query
 - Represented by a table with rows and columns
 - In JDBC 1.0 you can only proceed forward through the rows using **next**

❖ Useful Methods

- All methods can throw a **SQLException**
- **close**
 - Releases the JDBC and database resources
 - The result set is automatically closed when the associated **Statement** object executes a new query
- **getMetaDataObject**
 - Returns a **ResultSetMetaData** object containing information about the columns in the **ResultSet**

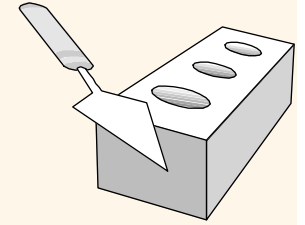
ResultSet (cont'd)



❖ Useful Methods

- **next**
 - Attempts to move to the next row in the **ResultSet**
 - If successful **true** is returned; otherwise, **false**
 - The first call to next positions the cursor at the first row
 - Calling **next** clears the **SQLWarning** chain
- **getWarnings**
 - Returns the first **SQLWarning** or **null** if no warnings occurred

ResultSet (cont'd)



❖ Useful Methods

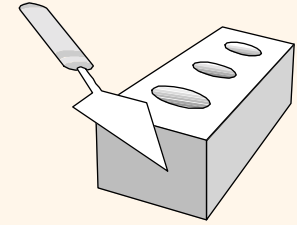
- **findColumn**
 - Returns the corresponding integer value corresponding to the specified column name
 - Column numbers in the result set do not necessarily map to the same column numbers in the database

- **getXxx**
 - Returns the value from the column specified by column name or column index as an **Xxx** Java type
 - Returns 0 or **null** (if the value is a SQL NULL)
 - Legal **getXxx** types:

double	byte	int	Date	String
float	short	long	Time	Object

- **wasNull**
 - To check if the last **getXxx** read was a SQL **NULL**

Using MetaData



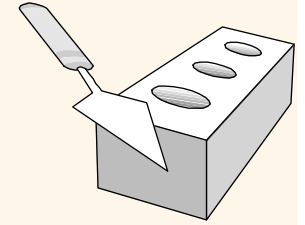
❖ Idea

- From a **ResultSet** (the return type of **executeQuery**), derive a **ResultSetMetaData** object
- Use that object to look up the number, names, and types of columns

❖ ResultSetMetaData answers the following questions:

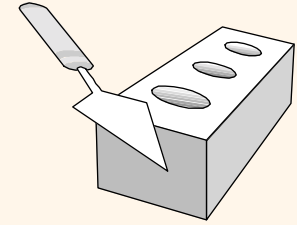
- How many columns are in the result set?
- What is the name of a given column?
- Are the column names case sensitive?
- What is the data type of a specific column?
- What is the maximum character size of a column?
- Can you search on a given column?

Useful MetaData Methods



- ❖ **getColumnCount**
 - Returns the number of columns in the result set
- ❖ **getColumnDisplaySize**
 - Returns the maximum width of the specified column in characters
- ❖ **getColumnName**
 - The **getColumnName** method returns the database name of the column
- ❖ **getColumnType**
 - Returns the SQL type for the column to compare against types in **java.sql.Types**

Useful MetaData Methods (cont'd)



❖ isNullable

- Indicates whether storing a **NULL** in the column is legal
- Compare the return value against **ResultSet** constants:
columnNoNulls, **columnNullable**, **columnNullableUnknown**

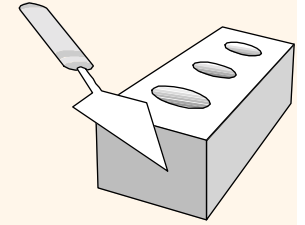
❖ isSearchable

- Returns **true** or **false** if the column can be used in a WHERE clause

❖ isReadOnly/isWritable

- The **isReadOnly** method indicates if the column is **definitely not writable**
- The **isWritable** method indicates whether it is **possible for a write to succeed**

Using MetaData: Example

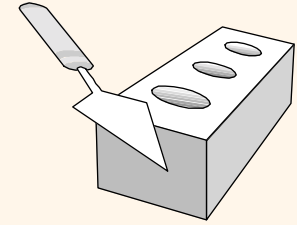


```
Connection connection =
DriverManager.getConnection(url, username, password);

// Look up info about the database as a whole.
DatabaseMetaData dbMetaData =
    connection.getMetaData();

String productName =
    dbMetaData.getDatabaseProductName();
System.out.println("Database: " + productName);
String productVersion =
    dbMetaData.getDatabaseProductVersion();
...
Statement statement = connection.createStatement();
String query = "SELECT * FROM pet";
ResultSet resultSet = statement.executeQuery(query);
```

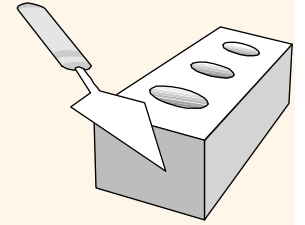
Using MetaData: Example



```
// Look up information about a particular table.
ResultSetMetaData resultsMetaData =
    resultSet.getMetaData();
int columnCount = resultsMetaData.getColumnCount();
// Column index starts at 1 (a la SQL) not 0 (a la Java).
for(int i=1; i<columnCount+1; i++) {
    System.out.print(resultsMetaData.getColumnName(i) +
        " ");
}
System.out.println();

// Print results.
while(resultSet.next()) {
    // Quarter
    System.out.print("    " + resultSet.getInt(1));
    // Number of Apples
    ...
}
```

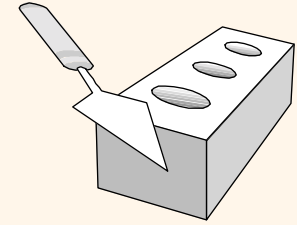
Using the Statement Object



❖ Overview

- Through the Statement object, SQL statements are sent to the database.
- Different types of statement objects are available:
 - Statement
 - for executing a **simple SQL** statements
 - PreparedStatement
 - for executing a **precompiled SQL statement** passing in parameters
 - CallableStatement
 - for executing a **database stored procedure**

Useful Statement Methods



❖ `executeQuery`

- Executes the SQL query and returns the data in a table (ResultSet)
- The resulting table may be empty but never null

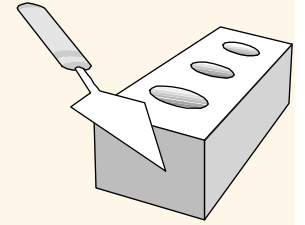
```
ResultSet results =  
    statement.executeQuery("SELECT a, b FROM table");
```

❖ `executeUpdate`

- Used to execute for INSERT, UPDATE, or DELETE SQL statements
- The return is the number of rows that were affected in the database
- Supports Data Definition Language (DDL) statements CREATE TABLE, DROP TABLE and ALTER TABLE

```
int rows =  
    statement.executeUpdate("DELETE FROM EMPLOYEES" +  
                            "WHERE STATUS=0");
```

Useful Statement Methods (cont'd)



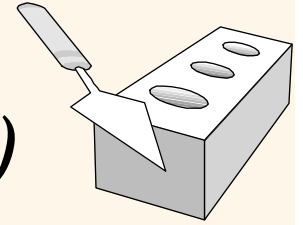
❖ getMaxRows/setMaxRows

- Determines the number of rows a **ResultSet** may contain
- Unless explicitly set, the number of rows are unlimited (return value of 0)

❖ getQueryTimeout/setQueryTimeout

- Specifies the amount of a time a driver will wait for a **STATEMENT** to complete before throwing a **SQLException**

Prepared Statements (Precompiled Queries)



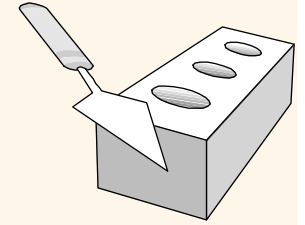
❖ Idea

- If you are going to execute **similar SQL statements** multiple times, using **“prepared” (parameterized) statements** can be more efficient
- Create a statement in standard form that is sent to the database for compilation before actually being used
- Each time you use it, you simply replace some of the marked parameters using the **setXxx** methods

❖ PreparedStatement's execute methods have no parameters

- `execute()`
- `executeQuery()`
- `executeUpdate()`

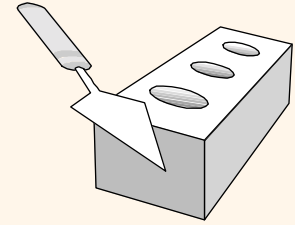
Prepared Statement, Example



```
Connection connection =
    DriverManager.getConnection(url, user, password);
PreparedStatement statement =
    connection.prepareStatement("UPDATE employees " +
                                "SET salary = ? " +
                                "WHERE id = ?");

float[] newSalaries = getSalaries();
int[] employeeIDs = getIDs();
for(int i=0; i<employeeIDs.length; i++) {
    statement.setFloat(1, newSalaries[i]);
    statement.setInt(2, employeeIDs[i]);
    statement.executeUpdate();
}
```


Exception Handling



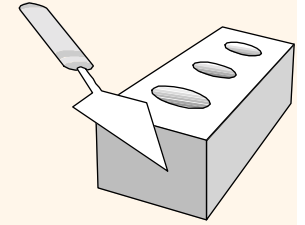
❖ SQL Exceptions

- Nearly every JDBC method can throw a **SQLException** in response to a data access error
- If more than one error occurs, they are **chained together**
- SQL exceptions contain:
 - Description of the error: **getMessage**

❖ The SQLState (Open Group SQL specification) identifying the exception: **getSQLState**

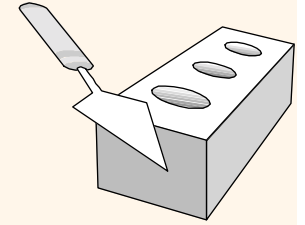
- A vendor-specific integer error code:, **getErrorCode**
- A chain to the next exception: **getNextException**

SQL Exception Example



```
try {
    ... // JDBC statement.
} catch (SQLException sqle) {
    while (sqle != null) {
        System.out.println("Message: " + sqle.getMessage());
        System.out.println("SQLState: " + sqle.getSQLState());
        System.out.println("Vendor Error: " +
            sqle.getErrorCode());
        sqle = sqle.getNextException();
    }
}
```

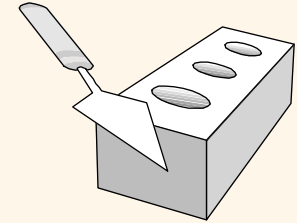
- ❖ Don't make assumptions about the state of a transaction after an exception occurs
- ❖ The safest best is to attempt a rollback to return to the initial state



SQL Warnings

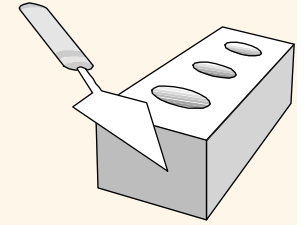
- ❖ SQLWarnings are rare, but provide information about the database access warnings
- ❖ Chained to object whose method produced the warning
- ❖ The following objects can receive a warning:
 - Connection
 - Statement (also, PreparedStatement, CallableStatement)
 - ResultSet
- ❖ Call `getWarning` to obtain the warning object, and `getNextWarning` (on the warning object) for any additional warnings
- ❖ Warnings are **cleared** on the object each time the **statement is executed**

SQL Warning, Example



```
ResultSet results = statement.executeQuery(someQuery);
SQLWarning warning = statement.getWarnings();
while (warning != null) {
    System.out.println("Message: " + warning.getMessage());
    System.out.println("SQLState: " + warning.getSQLState());
    System.out.println("Vendor Error: " +
        warning.getErrorCode());
    warning = warning.getNextWarning();
}
while (results.next()) {
    int value = rs.getInt(1);
    ... // Call additional methods on result set.
    SQLWarning warning = results.getWarnings();
    while (warning != null) {
        System.out.println("Message: " + warning.getMessage());
        System.out.println("SQLState: " + warning.getSQLState());
        System.out.println("Vendor Error: " +
            warning.getErrorCode());
        warning = warning.getNextWarning();
    }
}
```

Transactions



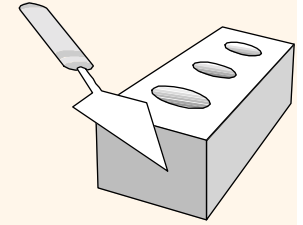
❖ Idea

- By default, after each SQL statement is executed the changes are **automatically committed** to the database
- Turn auto-commit off to group two or more statements together into a transaction

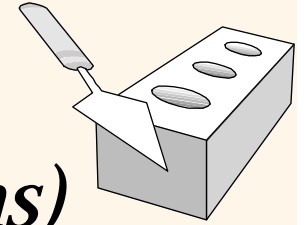
```
connection.setAutoCommit(false)
```

- Call **commit** to permanently record the changes to the database after executing a group of statements
- Call **rollback** if an error occurs

Transactions: Example



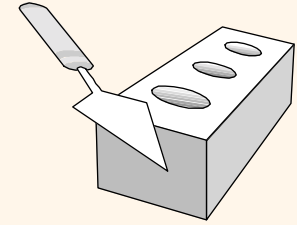
```
Connection connection =
    DriverManager.getConnection(url, username, passwd);
connection.setAutoCommit(false);
try {
    statement.executeUpdate(...);
    statement.executeUpdate(...);
    ...
} catch (Exception e) {
    try {
        connection.rollback();
    } catch (SQLException sqle) {
        // report problem
    }
} finally {
    try {
        connection.commit();
        connection.close();
    } catch (SQLException sqle) { }
}
```



Useful Connection Methods (for Transactions)

- ❖ **getAutoCommit/setAutoCommit**
 - By default, a connection is set to auto-commit
 - Retrieves or sets the auto-commit mode
- ❖ **commit**
 - Force all changes since the last call to commit to become permanent
 - Any database locks currently held by this **Connection** object are released
- ❖ **rollback**
 - Drops all changes since the previous call to commit
 - Releases any database locks held by this **Connection** object

Some JDBC Utilities



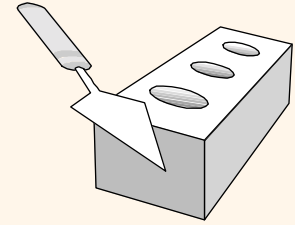
❖ Idea

- Performing JDBC queries and formatting output are common tasks, so create helper classes to perform this function: **DatabaseUtilities** and **DBResults**

❖ Class methods

- **getQueryResults**
 - Connects to a database, executes a query, retrieves all the rows as arrays of strings, and puts them inside a **DBResults** object
- **createTable**
 - Given a table name, a string denoting the column formats, and an array of strings denoting row values, this method issues a **CREATE TABLE** command and then sends a series of **INSERT INTO** commands for each row

Some more JDBC Utilities



❖ More Class methods

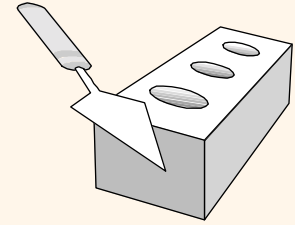
▪ printTable

- Given a table name, this method connects to the database, retrieves all the rows, and prints them on the standard output

▪ printTableData

- Given a DBResults object from a previous query, prints the results to standard output. Useful for debugging

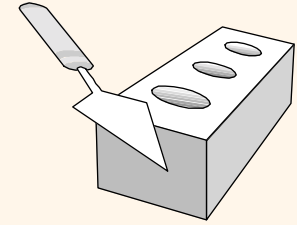
Using JDBC Utilities



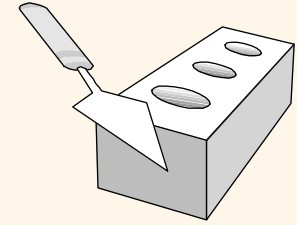
❖ Usage Example

```
DBResults results =  
    DatabaseUtilities.getQueryResults(driver, url,  
                                     username, password,  
                                     query, true);  
out.println(results.toHTMLTable("CYAN"));
```

Summary of Hints



- ❖ In JDBC 1.0, can only step forward (`next`) through the `ResultSet`
- ❖ `MetaDataResultSet` provides details about returned `ResultSet`
- ❖ Improve performance through prepared statements
- ❖ Be sure to handle the situation where `getXxx` returns a `NULL`
- ❖ By default, a connection is auto-commit
- ❖ SQL Exceptions and Warnings are chained together



On-line Resources

- ❖ Sun's JDBC Site
 - <http://java.sun.com/products/jdbc/>
- ❖ JDBC Tutorial
 - <http://java.sun.com/docs/books/tutorial/jdbc/>
- ❖ List of Available JDBC Drivers
 - <http://industry.java.sun.com/products/jdbc/drivers/>
- ❖ API for java.sql
 - <http://java.sun.com/j2se/1.4.2/docs/api/java/sql/package-summary.html>
- ❖ JDBC Sample Code for Oracle
 - http://www.oracle.com/technology/sample_code/tech/java/sqlj_jdbc/index.html