

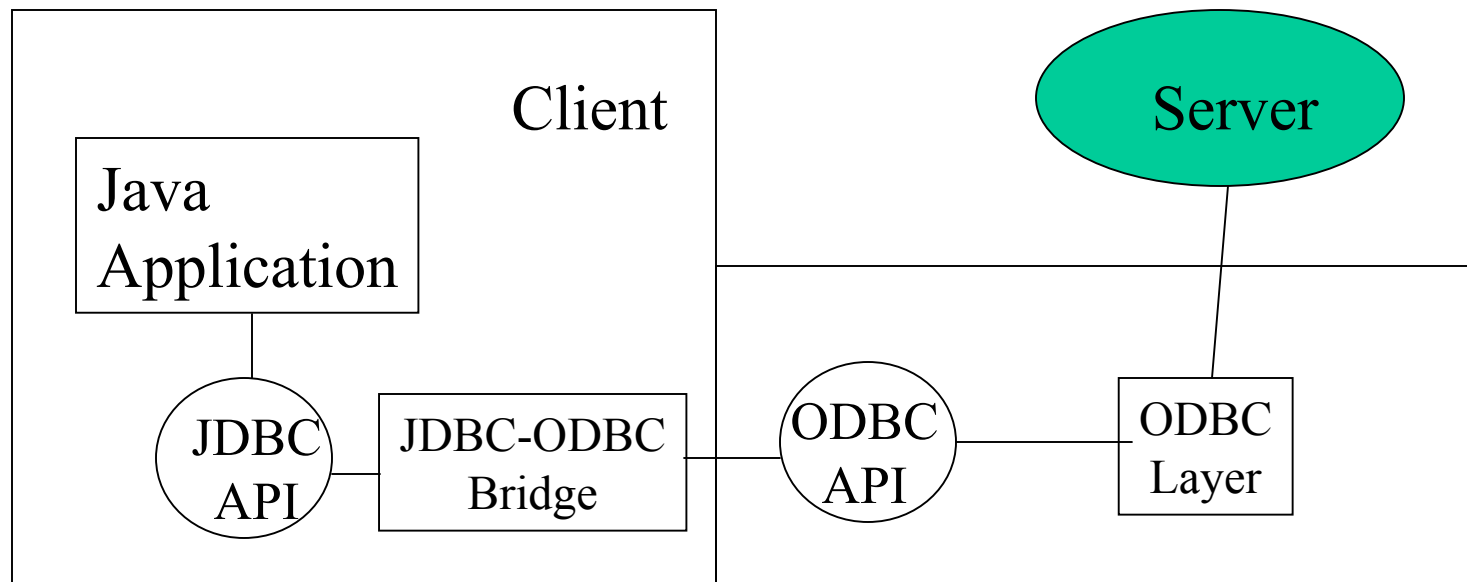
# **INFS 740**

## **JDBC & Servlets**

Prof. Alex Brodsky  
IT&E, GMU

# JDBC

- JDBC is a specialized API for Java programs to access RDBMS



# JDBC Introduction

- 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

# 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/1.3/docs/api/java/sql/package-summary.html>

# JDBC Drivers

- 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.
    - Point: translation to vendor format is performed on the client
      - No changes needed to server
      - Driver (translator) needed in client

# 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("com.mysql.jdbc.Driver");  
} catch (ClassNotFoundException cnfe) {  
    System.out.println("Error loading driver: " + cnfe);  
}
```

## 2. Define the Connection URL

```
String host = "natalie.emba.uvm.edu";  
String dbName = "someName";  
int port = 1234;  
String mysqlURL = "jdbc:mysql://" + host +  
    ":" + port + "/" + dbName;
```

# JDBC: Details of Process, cont.

## 3. Establish the Connection

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

## 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.

## 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://natalie.emba.uvm.edu:3307/try",
                "me", "xyz"); //?user=me&password=xyz");
```

# Basic JDBC Example, cont.

```
Statement s = C.createStatement();
String sql="select * from pet";
s.execute(sql);
ResultSet res=s.getResultSet();
if (res!=null) {
    while(res.next()){//note MySql start with 1
        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 (Continued)

- 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 (Continued)

- 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?
  - What is the data type of a specific column?
  - What is the maximum character size of a column?
  - Etc.

# 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/getColumnLabel
  - The **getColumnName** method returns the database name of the column
  - The **getColumnLabel** method returns the suggested column label for printouts
- getColumnType
  - Returns the SQL type for the column to compare against types in **java.sql.Types**

# Useful MetaData Methods (Continued)

- 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()) {
    for(int i=1; i<columnCount+1; i++) {
        switch (resultsMetaData.getColumnType(i)) {
            ...
        }
    }
}
}
```

# Using the Statement Object

- Overview
  - Through the Statement object, SQL statements are sent to the database.
  - Three 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
- `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

```
ResultSet results =
```

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

```
int rows =
```

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

# Useful Statement Methods (Continued)

- `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();
}
```

# Useful Prepared Statement Methods

- `setXxx`
  - Sets the indicated parameter (?) in the SQL statement to the value
- `clearParameters`
  - Clears all set parameter values in the statement

# 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.printStackTrace(System.out);  
        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

# 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

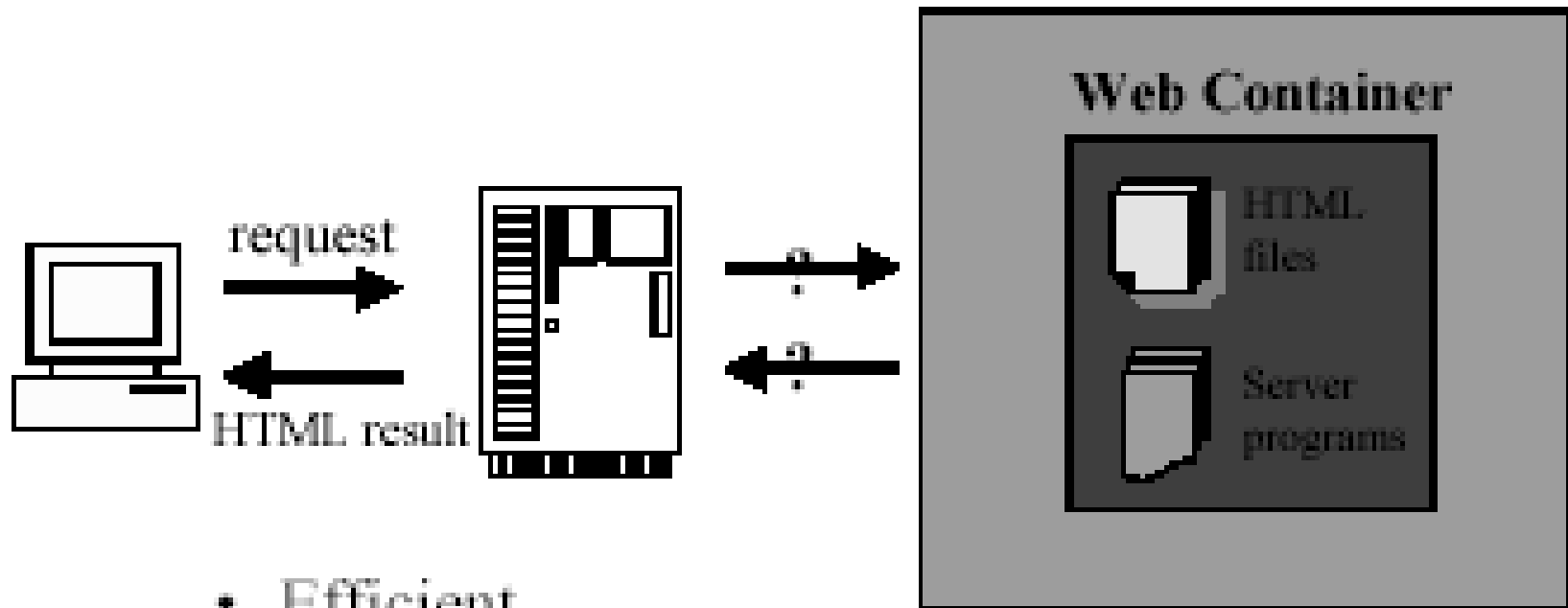
# Transactions: Example

```
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

# Java Servlets



- Efficient
- Powerful
- Portable

# Server Side

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
public class Hello extends HttpServlet {
    protected void doPost(HttpServletRequest req,
                           HttpServletResponse res)
        throws ServletException, IOException {
        String name = req.getParameter("name");
        res.setContentType("text/html");
    }
}
```

```
PrintWriter out = res.getWriter();  
out.println("<HTML>");  
out.println("<body>");  
out.println("<p> Hello " + name + "</p>");  
out.println("</body>");  
out.println("</html>");  
out.close();  
  
}  
  
}
```

# GET and POST Methods

- GET requests put form data on the URL as parameters
  - `http://www .../RunForm?NAME=Ye&TITLE=prof`
- GET parameters are limited to 1024 bytes
- POST requests put form data in body of request

# Servlet Program Data Handling

- Input and output through parameters passed in to the doPost and doGet methods.

HttpServletRequest req

HttpServletResponse res

# Servlet using JDBC Example

```
package cwp;
import java.io.*;
import java.sql.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class Menagerie extends HttpServlet {
    public static void main(String[] args) {
        System.out.println(doQuery());
    }
    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
                      throws ServletException, IOException {
        PrintWriter out = response.getWriter();
        out.println(doQuery());
    }
    public static String doQuery() {
        StringBuffer buffer = new StringBuffer();
        try {
            Class.forName("com.mysql.jdbc.Driver");
```

# Servlet using JDBC Example (Continued)

```
Connection connection =
    DriverManager.getConnection(
        "jdbc:mysql://almaak.usc.edu:3307/menagerie",
        "root", "xyz");
Statement statement = connection.createStatement();
String query = "SELECT * from pet";
ResultSet result = statement.executeQuery(query);
buffer.append("Pet Table from menagerie Database\n\n");
while (result.next()) {
    buffer.append(result.getString(1) + " " +
        result.getString(2) + " " +
        result.getString(3) + " " +
        result.getString(4) + " " +
        result.getString(5) + "\n");
}
connection.close();
} catch (ClassNotFoundException cnfe) {
    buffer.append("Couldn't find class file" + cnfe);
} catch (SQLException sqle) {
    buffer.append("SQL Exception: " + sqle);
}
return buffer.toString();
}
}
```