CSE532
Supplemental material on Java

CSE 532, Theory of Database Systems
Stony Brook University

http://www.cs.stonybrook.edu/~cse532
Supplemental Java material

  - Learn how to load a driver, connect to a database, execute statements, and process result sets using JDBC (§37.4)
  - Prepared statements to execute precompiled SQL statements (§37.5)
  - Use callable statements to execute stored SQL procedures and functions (§37.6)
  - Explore database metadata using the DatabaseMetaData and ResultSetMetaData interfaces (§37.7).
  - Create a universal SQL client for accessing local or remote database (§38.2).
  - Execute SQL statements in a batch mode (§38.3).
  - Process updateable and scrollable result sets (§38.4).
  - Use RowSet (§38.5).

(c) Pearson and P.Fodor (CS Stony Brook)
Supplemental Java material

- Java Servlets (§39).
  - Deploy servlets on application servers such as Tomcat (§39.3).
  - Describe the servlets API (§39.4).
  - Create simple servlets (§39.5).
  - Create and process HTML forms (§39.6).
  - Develop servlets to access databases (§39.7).
  - Use hidden fields, cookies, and HttpSession to track sessions (§39.8)

- JavaServer Pages (JSP) (§40).
  - How a JSP page is processed (§40.3).
  - Use JSP constructs to code JSP script (§40.4).
  - Use predefined variables and directives in JSP (§§40.5-40.6).
  - Use JavaBeans components in JSP (§40.7-40.9).
  - Develop database applications using JSP (§40.7-40.9).
Supplemental Java material

• JavaServer Faces (JSF) (§41).
  • Create JSF UI components (e.g., Static Text, Text Field, Button, Drop Down List, List Box, Radio Button Group, Check Box Group, Text Area, Table) (§41.3).
  • Use JSF containers Grid Panel, Group Panel, and Layout Panel to group components (§41.4).
  • Bind data with JSF UI components (§41.5).
  • Validate input using Message components (§41.6).
Why Java for Database Programming and Web development?

- First, Java is platform independent. You can develop platform-independent database applications using SQL and Java for any relational database systems.
- Second, the support for accessing database systems from Java is built into Java API, so you can create database applications using all Java code with a common interface.
- Third, Java is taught in almost every university either as the first programming language or as the second programming language.
The Architecture of JDBC

- Java Applications/Applets
  - JDBC API
    - DB2 JDBC Driver
    - JDBC-ODBC Bridge Driver
      - DB2 ODBC Driver
      - Microsoft ODBC Driver
        - Local or remote DB2 DB
        - Microsoft Access Database

(c) Pearson and P.Fodor (CS Stony Brook)
The JDBC Interfaces

Loading drivers

Establishing connections

Creating and executing statements

Processing ResultSet

Driver

Connection

Statement

ResultSet

Statement

ResultSet

Statement

ResultSet

Statement

ResultSet

(c) Pearson and P.Fodor (CS Stony Brook)
Developing JDBC Programs

**Loading drivers**

Statement to load a driver:
```
Class.forName("JDBCDriverClass");
```

A driver is a class. For example:

<table>
<thead>
<tr>
<th>Database</th>
<th>Driver Class</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>Access</td>
<td>sun.jdbc.odbc.JdbcOdbcDriver</td>
<td>Already in JDK</td>
</tr>
<tr>
<td>MySQL</td>
<td>com.mysql.jdbc.Driver</td>
<td>Website</td>
</tr>
<tr>
<td>Oracle</td>
<td>oracle.jdbc.driver.OracleDriver</td>
<td>Website</td>
</tr>
</tbody>
</table>

The JDBC-ODBC driver for Access is bundled in JDK.
MySQL driver class is in mysqljdbc.jar
Oracle driver class is in classes12.jar

To use the MySQL and Oracle drivers, you have to add mysqljdbc.jar and classes12.jar in the classpath using the following DOS command on Windows:
```
classpath=%classpath%;c:\book\mysqljdbc.jar;c:\book\classes12.jar
```
Importing JDBC DB2 Driver into Eclipse

- To use JDBC in your application, you must first download and install a DB2 JDBC driver: db2jcc.jar
  - [http://www.aquafold.com/docs-jdbcdrivers-db2-9-0.html](http://www.aquafold.com/docs-jdbcdrivers-db2-9-0.html)

- Once downloaded, you must import the driver into Eclipse: put the driver's folder in your CLASSPATH variable or add the jar file to your project (Properties->Java Build Paths)
  - To import db2jcc.jar, click on the Window menu in Eclipse and select Preferences. In the resulting dialog box, choose Java then Build Path then User Libraries. Click on New and define a library name, e.g., DB2LIBS. Then click Add JARs, navigate to the folder that contains db2jcc.jar and add the driver to the library.
  - Also add db2jcc.jar to tomcat's lib folder to enable the servlet to access the database.
Importing JDBC DB2 Driver into Eclipse

- To use JDBC in your application, you must first download and install a DB2 JDBC driver: db2jcc.jar
  - [http://www.aquafold.com/docs-jdbcdrivers-db2-9-0.html](http://www.aquafold.com/docs-jdbcdrivers-db2-9-0.html)

- Once downloaded, you must import the driver into Eclipse: put the driver's folder in your CLASSPATH variable or add the jar file to your project (Properties->Java Build Paths)
  - To import db2jcc.jar, click on the Window menu in Eclipse and select Preferences. In the resulting dialog box, choose Java then Build Path then User Libraries. Click on New and define a library name, e.g., DB2LIBS. Then click Add JARs, navigate to the folder that contains db2jcc.jar and add the driver to the library.
  - Also add db2jcc.jar to tomcat's lib folder to enable the servlet to access the database.
Tomcat plugin in Eclipse

- To run your application, you would need to install Tomcat on your machine.
  - [http://tomcat.apache.org](http://tomcat.apache.org)
- The easiest way to debug and run your application is to install the Eclipse Tomcat plugin:
  - [http://marketplace.eclipse.org/content/mongrel](http://marketplace.eclipse.org/content/mongrel)
Other DB2 links

- Stony Brook DB2 server:
  - [http://www.cs.sunysb.edu/facilities/windowslab/services/db2.html](http://www.cs.sunysb.edu/facilities/windowslab/services/db2.html)

- DB2 Express Server (needed if you want to set up a database on your own machine):

- DB2 Client (you need it to connect to DB2 remotely):

  - Using Data Studio:

- DB2 University Training courses: [http://www.db2university.com/courses](http://www.db2university.com/courses)

- DB2 9r7 Database Reference: [http://publib.boulder.ibm.com/infocenter/db2luw/v9r7/index.jsp](http://publib.boulder.ibm.com/infocenter/db2luw/v9r7/index.jsp)

  Click on Database Reference in the left frame, then SQL, then Statements. For user-defined data types (UDTs), see CREATE TYPE (both row and structured).

(c) Pearson and P. Fodor (CS Stony Brook)
Other DB2 links


- More examples of using XQuery and JDBC:
Developing JDBC Programs

---

Connecting to databases:

### Loading drivers

- **Establishing connections**
- **Creating and executing statements**
- **Processing ResultSet**

```java
Connection connection = DriverManager.getConnection(databaseURL);
```

<table>
<thead>
<tr>
<th>Database</th>
<th>URL Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>Access</td>
<td>jdbc:odbc:dataSource</td>
</tr>
<tr>
<td>MySQL</td>
<td>jdbc:mysql://hostname/dbname</td>
</tr>
<tr>
<td>Oracle</td>
<td>jdbc:oracle:thin:@hostname:port#:oracleDBSID</td>
</tr>
</tbody>
</table>

**Examples:**

**For Access:**

```java
Connection connection = DriverManager.getConnection("jdbc:odbc:ExampleMDBDataSource");
```

**For MySQL:**

```java
Connection connection = DriverManager.getConnection("jdbc:mysql://localhost/test");
```

**For Oracle:**

```java
Connection connection = DriverManager.getConnection("jdbc:oracle:thin:@liang.armstrong.edu:1521:orcl", "scott", "tiger");
```

See Supplement IV.D for creating an ODBC data source.
### Developing JDBC Programs

<table>
<thead>
<tr>
<th>Loading drivers</th>
<th>Establishing connections</th>
<th>Creating and executing statements</th>
<th>Processing ResultSet</th>
</tr>
</thead>
</table>

**Creating statement:**

```java
Statement statement = connection.createStatement();
```

**Executing statement (for update, delete, insert):**

```java
statement.executeUpdate("create table Temp (col1 char(5), col2 char(5))");
```

**Executing statement (for select):**

```java
// Select the columns from the Student table
ResultSet resultSet = statement.executeQuery("select firstName, mi, lastName from Student where lastName " + " = 'Smith'" });
```
Developing JDBC Programs

Loading drivers
Establishing connections
Creating and executing statements
Processing ResultSet

Executing statement (for select):

// Select the columns from the Student table
ResultSet resultSet = stmt.executeQuery
("select firstName, mi, lastName from Student where lastName " + " = 'Smith'");

Processing ResultSet (for select):

// Iterate through the result and print the student names
while (resultSet.next())
    System.out.println(resultSet.getString(1) + " " + resultSet.getString(2) + ". " + resultSet.getString(3));
import java.sql.*;
public class SimpleJdbc {
    public static void main(String[] args)
        throws SQLException, ClassNotFoundException {
        // Load the JDBC driver
        Class.forName("com.mysql.jdbc.Driver");
        System.out.println("Driver loaded");

        // Establish a connection
        Connection connection = DriverManager.getConnection
            ("jdbc:mysql://localhost/test");
        System.out.println("Database connected");

        // Create a statement
        Statement statement = connection.createStatement();

        // Execute a statement
        ResultSet resultSet = statement.executeQuery
            ("select firstName, mi, lastName from Student where lastName "
            + " = 'Smith'");

        // Iterate through the result and print the student names
        while (resultSet.next())
            System.out.println(resultSet.getString(1) + "\t" +
                resultSet.getString(2) + "\t" + resultSet.getString(3));

        // Close the connection
        connection.close();
    }
}
Processing Statements

• Once a connection to a particular database is established, it can be used to send SQL statements from your program to the database.

• JDBC provides the Statement, PreparedStatement, and CallableStatement interfaces to facilitate sending statements to a database for execution and receiving execution results from the database.
Processing Statements Diagram

DriverManager.registerDriver (new DriverName())

DriverManager.getConnection()

Connection
  - createStatement()
  - prepareStatement()
  - prepareCall()

Statement
  - PreparedStatement
  - CallableStatement

PreparedStatement
  - executeUpdate()
  - executeQuery()
  - execute()
  - getMoreResults()
  - getResultSet()
  - getUpdateCount()

PreparedStatement

CallableStatement

ResultSet
  - next()
  - getString()
  - getInt()

(c) Pearson and P.Fodor (CS Stony Brook)
The execute, executeQuery, and executeUpdate Methods

• The methods for executing SQL statements are execute, executeQuery, and executeUpdate – each one accepts a string containing a SQL statement as an argument (it is passed to the database for execution).

• The execute method should be used if the execution produces multiple result sets, multiple update counts, or a combination of result sets and update counts.

• The executeQuery method should be used if the execution produces a single result set (such as in the case of a SQL select statement).

• The executeUpdate method should be used if the statement results in a single update count or no update count, such as a SQL INSERT, DELETE, UPDATE, or DDL statement.
The PreparedStatement interface is designed to execute dynamic SQL statements and SQL-stored procedures with IN parameters.

- These SQL statements and stored procedures are precompiled for efficient use when repeatedly executed.

```java
Statement pstmt = connection.prepareStatement
    ("insert into Student (firstName, mi, lastName) +
     values (?, ?, ?)");
```
Retrieving Database Metadata

- Database metadata is the information that describes database itself.
  - JDBC provides the DatabaseMetaData interface for obtaining database wide information and the ResultSetMetaData interface for obtaining the information on the specific ResultSet.
  - The DatabaseMetaData interface provides more than 100 methods for getting database metadata concerning the database as a whole.
    - These methods can be divided into three groups: for retrieving general information, for finding database capabilities, and for getting object descriptions.
DatabaseMetaData dbMetaData = connection.getMetaData();

System.out.println("database URL: " + dbMetaData.getURL());
System.out.println("database username: " +
    dbMetaData.getUsername());
System.out.println("database product name: " +
    dbMetaData.getDatabaseProductName());
System.out.println("database product version: " +
    dbMetaData.getDatabaseProductVersion());
System.out.println("JDBC driver name: " +
    dbMetaData.getDriverName());
System.out.println("JDBC driver version: " +
    dbMetaData.getDriverVersion());
System.out.println("JDBC driver major version: " +
    new Integer(dbMetaData.getDriverMajorVersion()));
System.out.println("JDBC driver minor version: " +
    new Integer(dbMetaData.getDriverMinorVersion()));
System.out.println("Max number of connections: " +
    new Integer(dbMetaData.getMaxConnections()));
System.out.println("MaxTableNameLength: " +
    new Integer(dbMetaData.getMaxTableNameLength()));
System.out.println("MaxColumnsInTable: " +
    new Integer(dbMetaData.getMaxColumnsInTable()));
connection.close();
Batch Updates

- To improve performance, JDBC 2 introduced the batch update for processing nonselect SQL commands (a batch update consists of a sequence of nonselect SQL commands): these commands are collected in a batch and submitted to the database all together.

```java
Statement statement = conn.createStatement();

// Add SQL commands to the batch
statement.addBatch("create table T (C1 integer, C2 varchar(15))");
statement.addBatch("insert into T values (100, 'Smith')");
statement.addBatch("insert into T values (200, 'Jones')");

// Execute the batch
int count[] = statement.executeBatch();
```

The `executeBatch()` method returns an array of counts, each of which counts the number of the rows affected by the SQL command. The first count returns 0 because it is a DDL command. The rest of the commands return 1 because only one row is affected.
A result set maintains a cursor pointing to its current row of data and data can be accessed sequentially:

- Initially the cursor is positioned before the first row.
- JDBC1: The `next()` method moves the cursor forward to the next row (known as sequential forward reading).
- JDBC 2: you can scroll the rows both forward and backward and move the cursor to a desired location using the `first`, `last`, `next`, `previous`, `absolute`, or `relative` methods.
- Additionally, you can insert, delete, or update a row in the result set and have the changes automatically reflected in the database.
Creating Scrollable Statements

To obtain a scrollable or updateable result set, you must first create a statement with an appropriate type and concurrency mode. For a static statement, use:

\[
\text{Statement statement} = \text{connection.createStatement} \quad \text{(int resultSetType, int resultSetConcurrency)};
\]

For a prepared statement, use

\[
\text{PreparedStatement statement} = \text{connection.prepareStatement} \quad \text{(String sql, int resultSetType, int resultSetConcurrency)};
\]

The resulting set is scrollable:

\[
\text{ResultSet resultSet} = \text{statement.executeQuery(query)};
\]
RowSet: JdbcTemplate and CachedRowSet

- JDBC 2 introduced a new RowSet interface that can be used to simplify database programming: the RowSet interface extends java.sql.ResultSet with additional capabilities that allow a RowSet instance to be configured to connect to a JDBC url, username, password, set a SQL command, execute the command, and retrieve the execution result.

![Diagram of RowSet interfaces](image)

(c) Pearson and P.Fodor (CS Stony Brook)
Database can store not only numbers and strings, but also images. SQL3 introduced a new data type BLOB (Binary Large OBject) for storing binary data, which can be used to store images.

Another new SQL3 type is CLOB (Character Large OBject) for storing a large text in the character format.

JDBC 2 introduced the interfaces java.sql.Blob and java.sql.Clob to support mapping for these new SQL types. JBDC 2 also added new methods, such as getBlob, setBinaryStream, getClob, setBlob, and setClob, in the interfaces ResultSet and PreparedStatement to access SQL BLOB, and CLOB values.

To store an image into a cell in a table, the corresponding column for the cell must be of the BLOB type. For example, the following SQL statement creates a table whose type for the flag column is BLOB.

```sql
CREATE TABLE Country (name VARCHAR(30), flag BLOB, description VARCHAR(255));
```
Storing and Retrieving Images in JDBC

To insert a record with images to a table, define a prepared statement like this one:

```java
PreparedStatement pstmt = connection.prepareStatement("insert into Country values(?, ?, ?)";
```

Images are usually stored in files. You may first get an instance of InputStream for an image file and then use the `setBinaryStream` method to associate the input stream with a cell in the table, as follows:

```java
// Store image to the table cell
File file = new File(imageFilenames[i]);
InputStream inputImage = new FileInputStream(file);
pstmt.setBinaryStream(2, inputImage, (int)(file.length()));
```

To retrieve an image from a table, use the `getBlob` method, as shown below:

```java
// Store image to the table cell
Blob blob = rs.getBlob(1);
Icon imageIcon = new ImageIcon(blob.getBytes(1, (int)blob.length()));
```
Java Servlets

- Servlet technology is primarily designed for use with the HTTP protocol of the Web.
- Servlets are Java programs that run on a Web server.
- Java servlets can be used to process client requests or produce dynamic Web pages.
HTTP and HTML
From CGI to Java Servlets

- The Common Gateway Interface, or CGI, was proposed to generate dynamic Web contents.
  - The interface provides a standard framework for Web servers to interact with external programs, known as the CGI programs.

- Java servlets are Java programs that function like CGI programs: they are executed upon the request from Web browser.
  - All the servlets run inside a servlet container (server or engine).
  - A servlet container is a single process that runs a JVM that creates a thread to handle each servlet (all the threads share the same memory allocated to the JVM).
How Does CGI Work?

Web Browser → Web Server

Send a request URL

Web Server

/htdocs/index.html
/cgi-bin/getBalance.cgi

Execute CGI Program

Generate Response

Get CGI Code

Host Machine File System

HTML Page returned

(c) Pearson and P.Fodor (CS Stony Brook)
The GET and POST Methods

• The two most common HTTP requests (methods) are: GET and POST.

  • The Web browser issues a request using a URL or an HTML form to trigger the Web server to execute a CGI program.

  • When issuing a CGI request directly from a URL, the GET method is used with a “query string”: the URL
Query String

- The URL query string consists of the location of the CGI program, parameters and their values.

http://www.webserverhost.com/cgi-bin/getBalance.cgi
  ?accountId=scott+smith&password=tiger

- The ? symbol separates the program from the parameters.
- The parameter name and value are associated using the = symbol.
- The parameter pairs are separated using the & symbol.
- The + symbol denotes a space character.
HTML Forms

HTML forms enable you to submit data to the Web server in a convenient form. The form can contain text fields, text area, check boxes, combo boxes, lists, radio buttons, and buttons.
Creating and Running Servlets

- To run Java servlets, you need a servlet container.
  - Many servlet containers are available.
  - Tomcat, developed by Apache, is a standard reference implementation for Java servlet 2.2 and Java Server Pages 1.1.
The Servlet API

- The servlet API provides the interfaces and classes that support servlets grouped into two packages: `javax.servlet`, and `javax.servlet.http`.

```
javax.servlet.*
javax.servlet.http.*
```
The Servlet Interface

/**Invoked for every servlet constructed*/
public void init(ServletConfig p0) throws ServletException;

/**Invoked to respond to incoming requests*/
public void service(ServletRequest p0, ServletResponse p1)
    throws ServletException, IOException;

/**Invoked to release resource by the servlet*/
public void destroy();

/**Return information about the servlet*/
public String getServletInfo();

/**Return configuration objects of the servlet*/
public ServletConfig getServletConfig();
Servlet Life-Cycle

1. The init method is called when the servlet is first created, and is not called again as long as the servlet is not destroyed.

2. The service method is invoked each time the server receives a request for the servlet. The server spawns a new thread and invokes service.

3. The destroy method is invoked once all threads within the servlet's service method have exited or after a timeout period has passed. This method releases resources for the servlet.
The HTTPServlet Class

- The HttpServlet class defines a servlet for the HTTP protocol.
- It extendsGenericServlet and implements the service method as a dispatcher of HTTP requests.
- The HTTP requests are processed in the following methods: doGet, doPost, doDelete, doPut, doOptions, and doTrace:

  ```java
  protected void doXxx(HttpServletRequest req, HttpServletResponse resp)
  throws ServletException, java.io.IOException
  ```
The HTTPServlet Class

- Every doXxx method in the HttpServlet class has an argument of the HttpServletRequest type, which is an object that contains HTTP request information including parameter name and values, attributes, and an input stream.

- Every doXxx method in the HttpServlet class has an argument of the HttpServletResponse type, which is an object that assists a servlet in sending a response to the client.
Creating Servlets

- The servlet engine controls the servlets using the init, doGet, doPost, destroy, and other methods. By default, the doGet and doPost methods do nothing.

- To handle the GET request, you need to override the doGet method; to handle the POST request, you need to override the doPost method.
Example: Obtaining Current Time Based on Locale and Time Zone
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.*;
import java.text.*;

public class TimeForm extends HttpServlet {
    private static final String CONTENT_TYPE = "text/html";
    private Locale[] allLocale = Locale.getAvailableLocales();
    private String[] allTimeZone = TimeZone.getAvailableIDs();
    /** Process the HTTP Get request */
    public void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
        response.setContentType(CONTENT_TYPE);
        PrintWriter out = response.getWriter();
        out.println("<h3>Choose locale and time zone</h3>"");
        out.println("<form method="post" action="+/liangweb/TimeForm">" +
                "<locale name="locale">" +
                "</select></form>" +
            "<h3>Choose locale and time zone</h3>"");
        out.println("<form method="post" action="+/liangweb/TimeForm">" +
                "<locale name="locale">" +
                "</select></form>" +
            "<h3>Choose locale and time zone</h3>"");
        // Fill in all locales
        for (int i = 0; i < allLocale.length; i++) {
            // Fill in all locales
        }
    }
}

(c) Pearson and P. Fodor (CS Stony Brook)
```java
out.println("<option value="" + i +"">" +
    allLocale[i].getDisplayName() + "</option>");
}
out.println("</select>");
// Fill in all time zones
out.println("<p>Time Zone<select size="1" name="timezone">");
for (int i = 0; i < allTimeZone.length; i++) {
    out.println("<option value="" + allTimeZone[i] +"">" +
        allTimeZone[i] + "</option>");
}
out.println("</select>");
out.println("<p><input type="submit" value="Submit" >
<input type="reset" value="Reset"></p>
</form>
out.close(); // Close stream
/** Process the HTTP Post request */
public void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
    PrintWriter out = response.getWriter();
    response.setContentType(CONTENT_TYPE);
    out.println("<html>"); ...
(c) Pearson and P.Fodor (CS Stony Brook)"
Many dynamic Web applications use databases to store and manage data.

Servlets can connect to any relational database via JDBC. Connecting a servlet to a database is no different from connecting a Java application or applet to a database.

If you know Java servlets and JDBC, you can combine them together to develop interesting and practical Web based interactive projects immediately.
Example: Registering Student into a Database

Please register to your instructor's student address book.

Last Name * Smith
First Name * John
MI M

Telephone 322-223-1212
Email smith@acm.org

Street 100 Main Street

City Savannah
State Georgia-GA
Zip 31419

Submit Reset

* required fields
<html> <!-- SimpleRegistration.html -->
<head><title>Simple Registration without Confirmation</title></head>
<body>
<form method = "post" action = "/liangweb/SimpleRegistration">
  <p>Last Name <font color =="#FF0000">*</font>
      <input type = "text" name = "lastName"></p>
  First Name <font color =="#FF0000">*</font>
      <input type = "text" name = "firstName">
  MI
      <input type = "text" name = "mi" size = "3">
</p>
  Telephone <input type = "text" name = "telephone" size = "20">
      Email <input type = "text" name = "email" size = "28">
</p>
  Street <input type = "text" name = "street" size = "50">
</p>
  City <input type = "text" name = "city" size = "23">
      State
      <select size = "1" name = "state">
        <option value = "GA">Georgia-GA</option>
        <option value = "OK">Oklahoma-OK</option>
        <option value = "IN">Indiana-IN</option>
      </select>
      Zip <input type = "text" name = "zip" size = "9">
</p>
  <p><input type = "submit" name = "Submit" value = "Submit">
      <input type = "reset" value = "Reset"></p>
</form>
<p><font color =="#FF0000">* required fields</font></p>
</body>
</html>
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.sql.*;

public class SimpleRegistration extends HttpServlet {

    private PreparedStatement pstmt;

    public void init() throws ServletException {
        initializeJdbc();
    }

    public void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        // Obtain parameters from the client
        String lastName = request.getParameter("lastName");
        String firstName = request.getParameter("firstName");
        String mi = request.getParameter("mi");
        String phone = request.getParameter("telephone");
        String email = request.getParameter("email");
        String address = request.getParameter("street");
        String city = request.getParameter("city");
        String state = request.getParameter("state");
        String zip = request.getParameter("zip");
        try {
            if (lastName.length() == 0 || firstName.length() == 0) {
                out.println("Last Name and First Name are required");
                return; // End the method
            }
        } finally {
            try {
                if (pstmt != null) {
                    pstmt.close();
                }
            } catch (SQLException e) {
                throw new ServletException(e.getMessage());
            }
        }
    }
}

(c) Pearson and P.Fodor (CS Stony Brook)
storeStudent(lastName, firstName, mi, phone, email, address,
city, state, zip);
out.println(firstName + " " + lastName +
" is now registered in the database");
}
catch(Exception ex) {
    out.println("Error: " + ex.getMessage());
}
finally {
    out.close(); // Close stream
}
/** Initialize database connection */
private void initializeJdbc() {
try {
    // Declare driver and connection string
    String driver = "sun.jdbc.odbc.JdbcOdbcDriver";
    String connectionString = "jdbc:odbc:exampleMDBDataSource";
    // For MySQL
    // String driver = "com.mysql.jdbc.Driver";
    // String connectionString = "jdbc:mysql://localhost/test";
    // For Oracle
    // String driver = "oracle.jdbc.driver.OracleDriver";
    // String connectionString = "jdbc:oracle:thin:scott/tiger@liang.armstrong.edu:1521:orcl";
    // Load the driver
    Class.forName(driver);
    // Connect to the sample database
    Connection conn = DriverManager.getConnection(connectionString);
// Create a Statement
pst = conn.prepareStatement("insert into Address " +
    "(lastName, firstName, mi, telephone, email, street, city, " +
    "state, zip) values (?, ?, ?, ?, ?, ?, ?, ?)"");
}

/** Store a student record to the database */
private void storeStudent(String lastName, String firstName,
    String mi, String phone, String email, String address,
    String city, String state, String zip) throws SQLException {
pstmt.setString(1, lastName);
pstmt.setString(2, firstName);
pstmt.setString(3, mi);
pstmt.setString(4, phone);
pstmt.setString(5, email);
pstmt.setString(6, address);
pstmt.setString(7, city);
pstmt.setString(8, state);
pstmt.setString(9, zip);
pstmt.executeUpdate();
}
Session Tracking

- Web servers use Hyper-Text Transport Protocol (HTTP).
- HTTP is a stateless protocol!
- The HTTP Web server cannot associate requests from a client together.
  - Each request is treated independently by the Web server.
  - This protocol works fine for simple Web browsing, where each request typically results in an HTML file or a text file being sent back to the client.
What is a Session?

- A session can be defined as a series of related interactions between a single client and the Web server over a period of time. To track data among requests in a session is known as session tracking.

- Session Tracking Techniques
  - Using hidden values, using cookies, and using the session tracking tools from servlet API.
Session Tracking Using Hidden Values

- You can track session by passing data from the servlet to the client as hidden value in a dynamically generated HTML form by including a field like this:

```html
<input type="hidden" name="lastName" value="Smith">
```

- So the next request will submit the data back to the servlet.

- The servlet retrieves this hidden value just like any other parameter value using the getParameter method.
Example: Using Hidden Values in the Registration form

- The client first submits the form using the GET method and the server collects the data in the form, displays the data to the client, and asks the client for confirmation.
- The client confirms it by submitting the request with the hidden values using the POST method.
- Finally, the servlet writes the data to a database.
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.sql.*;

public class Registration extends HttpServlet {
    private PreparedStatement pstmt;
    public void init() throws ServletException {
        initializeJdbc();
    }
    public void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        // Obtain data from the form
        String lastName = request.getParameter("lastName");
        String firstName = request.getParameter("firstName");
        // Ask for confirmation
        out.println("You entered the following data");
        out.println("<p> Last name: " + lastName);
        // Set the action for processing the answers
        out.println("<p><form method="post" action="/liangweb/Registration"> ");
        // Set hidden values
        out.println("<p><input type="hidden" value=" + lastName + " name="lastName"> ");
        out.println("<p><input type="submit" value="Confirm" >");
        out.println("</form>");
        out.close(); // Close stream
    }
}
Session Tracking Using Cookies

- You can track sessions using cookies.
- Cookies are small text files that store sets of name=value pairs on the disk in the client’s computer.
- Cookies are sent from the server through the instructions in the header of the HTTP response.
- The instructions tell the browser to create a cookie with a given name and its associated value. If the browser already has the cookie with the key name, the value will be updated.
- The browser will then send the cookie with any request submitted to the same server.
- Cookies can have expiration dates set, after which the cookies will not be sent to the server.
Session Tracking Using the Servlet API

- The problems of session tracking with hidden data and cookies are that data are not secured and difficult to deal with large set of data.

- Java servlet API provides a session tracking tool, which enables tracking of a large set of data.
  - Data can be stored as objects.
  - Data are kept on the server side so they are secure.
The HttpSession Class

- The Java servlet API for session tracking:
  - Create a session object using the getSession method in the HttpServletRequest interface:

    ```java
    HttpSession session = request.getSession(true);
    ```

    - This obtains the session or creates a new session if the client does not have a session on the server.

- The HttpSession class provides the methods for reading and storing data to the session, and for manipulating the session.
Java Server Pages (JSP)

<!-- CurrentTime.jsp -->

<HTML>
<HEAD>
<TITLE>
CurrentTime
</TITLE>
</HEAD>
<BODY>
Current time is <%= new java.util.Date() %>
</BODY>
</HTML>
How Is a JSP Processed?

URL Example
http://www.server.com:8080/servlet/JSPFile

Web Browser
Send a request URL
Web Server
Process Servlet
Generate Response
Get Servlet
Servlet Engine
Get JSP File
Generated Servlets
JSP Translator
Web Server Host
/Servlet/JSPFile.jsp
Host Machine File System
HTML Page returned

(c) Pearson and P.Fodor (CS Stony Brook)
JSP Constructs

There are three types of scripting constructs you can use to insert Java code into the resultant servlet. They are *expressions*, *scriptlets*, and *declarations*.

**expression**

A JSP expression is used to insert a Java expression directly into the output. It has the following form:

```jsp
<%= Java-expression %>
```

The expression is evaluated, converted into a string, and sent to the output stream of the servlet.

**scriptlet**

**declaration**
JSP Constructs

There are three types of scripting constructs you can use to insert Java code into the resultant servlet. They are expressions, scriptlets, and declarations.

expression

scriptlet

declaration

A JSP scriptlet enables you to insert a Java statement into the servlet’s jspService method, which is invoked by the service method. A JSP scriptlet has the following form:

<% Java statement %>
JSP Constructs

There are three types of scripting constructs you can use to insert Java code into the resultant servlet. They are expressions, scriptlets, and declarations.

expression
scriptlet
declaration

A JSP declaration is for declaring methods or fields into the servlet. It has the following form:

```<%! Java method or field declaration %>```
JSP Comment

HTML comments have the following form:

<!-- HTML Comment -->

If you don’t want the comment appear in the resultant HTML file, use the following comment in JSP:

<%-- JSP Comment --%>
Computing Factorials

JSP scriptlet:
```jsp
<% for (int i = 0; i <= 10; i++) { %>
Factorial of <%= i %> is
<%= computeFactorial(i) %> <br /><% } %>
```

JSP expression:
```
<%= computeFactorial(i) %>
```

JSP declaration:
```
<%! public long computeFactorial(int n) {
    if (n == 0)
        return 1;
    else
        return n * computeFactorial(n - 1);
}
%>
```
You can use variables in JSP. For convenience, JSP provides eight predefined variables from the servlet environment that can be used with JSP expressions and scriptlets. These variables are also known as *JSP implicit objects*.

request
response
out
session
application
config
pagecontext
page

**Represented the client’s request, which is an instance of HttpServletRequest. You can use it to access request parameters, HTTP headers such as cookies, hostname, etc.**
JSP Predefined Variables

You can use variables in JSP. For convenience, JSP provides eight predefined variables from the servlet environment that can be used with JSP expressions and scriptlets. These variables are also known as *JSP implicit objects*.

- `request` Represents the servlet’s response, which is an instance of `HttpServletRequest`. You can use it to set response type and send output to the client.
- `response` Represents the servlet’s response, which is an instance of `HttpServletResponse`. You can use it to set response type and send output to the client.
- `out` Represents the servlet’s response, which is an instance of `HttpServletResponse`. You can use it to set response type and send output to the client.
- `session` Represents the servlet’s response, which is an instance of `HttpSession`. You can use it to access client-specific data.
- `application` Represents the servlet’s response, which is an instance of `ServletContext`. You can use it to access application-wide data.
- `config` Represents the servlet’s response, which is an instance of `ServletConfig`. You can use it to access configuration information.
- `pagecontext` Represents the servlet’s response, which is an instance of `PageContext`. You can use it to access page-level information.
- `page` Represents the servlet’s response, which is an instance of `PageContext`. You can use it to access page-level information.
JSP Predefined Variables

You can use variables in JSP. For convenience, JSP provides eight predefined variables from the servlet environment that can be used with JSP expressions and scriptlets. These variables are also known as *JSP implicit objects*.

- **request**
  Represents the character output stream, which is an instance of PrintWriter obtained from `response.getWriter()`. You can use it to send character content to the client.

- **response**

- **out**

- **session**

- **application**

- **config**

- **pagecontext**

- **page**
JSP Predefined Variables

You can use variables in JSP. For convenience, JSP provides eight predefined variables from the servlet environment that can be used with JSP expressions and scriptlets. These variables are also known as *JSP implicit objects*.

- request
- response
- out
- session
- application
- config
- pagecontext
- page

**request**

Represents the HttpSession object associated with the request, obtained from request.getSession().
You can use variables in JSP. For convenience, JSP provides eight predefined variables from the servlet environment that can be used with JSP expressions and scriptlets. These variables are also known as *JSP implicit objects*.

<table>
<thead>
<tr>
<th>request</th>
<th>response</th>
<th>out</th>
<th>session</th>
<th>application</th>
<th>config</th>
<th>pagecontext</th>
<th>page</th>
</tr>
</thead>
</table>

**request**

**response**

**out**

**session**

**application**

**config**

**pagecontext**

**page**

Represents the ServletContext object for storing persistent data for all clients. The difference between session and application is that session is tied to one client, but application is for all clients to share persistent data.
JSP Predefined Variables

You can use variables in JSP. For convenience, JSP provides eight predefined variables from the servlet environment that can be used with JSP expressions and scriptlets. These variables are also known as *JSP implicit objects*.

- **request**
- **response**
- **out**
- **session**
- **application**
- **config**
- **pagecontext**
- **page**

Represented the ServletConfig object for the page.
JSP Predefined Variables

You can use variables in JSP. For convenience, JSP provides eight predefined variables from the servlet environment that can be used with JSP expressions and scriptlets. These variables are also known as *JSP implicit objects*.

- **request**
- **response**
- **out**
- **session**
- **application**
- **config**
- **pagecontext**
- **page**

**Represent the PageContext object.**

PageContext is a new class introduced in JSP to give a central point of access to many page attributes.

(c) Pearson and P. Fodor (CS Stony Brook)
You can use variables in JSP. For convenience, JSP provides eight predefined variables from the servlet environment that can be used with JSP expressions and scriptlets. These variables are also known as *JSP implicit objects*.

request
response
out
session
application
config
pagecontext
page

Page is an alternative to this.
Computing Loan

Write an HTML page that prompts the user to enter loan amount, annual interest rate, and number of years. Clicking the Compute Loan Payment button invokes a JSP to compute and display the monthly and total loan payment.

```html
<!-- ComputeLoan.html -->
<html>
<head>
<title>ComputeLoan</title>
</head>
<body>
Compute Loan Payment

<form method="get" action="ComputeLoan.jsp">
<p>Loan Amount
   <input type="text" name="loanAmount"><br>
Annual Interest Rate
   <input type="text" name="annualInterestRate"><br>
Number of Years <input type="text" name="numberOfYears" size="3"></p>
<p><input type="submit" name="Submit" value="Compute Loan Payment">
   <input type="reset" value="Reset"></p>
</form>
</body>
</html>
```
<html><head>
<title>ComputeLoan</title></head>
<body>
<% double loanAmount = Double.parseDouble(request.getParameter("loanAmount"));
    double annualInterestRate = Double.parseDouble(request.getParameter("annualInterestRate"));
    double numberOfYears = Integer.parseInt(request.getParameter("numberOfYears"));
    double monthlyInterestRate = annualInterestRate / 1200;
    double monthlyPayment = loanAmount * monthlyInterestRate / (1 - 1 / Math.pow(1 + monthlyInterestRate, numberOfYears * 12));
    double totalPayment = monthlyPayment * numberOfYears * 12; %>
Loan Amount: <%= loanAmount %><br>
Annual Interest Rate: <%= annualInterestRate %><br>
Number of Years: <%= numberOfYears %><br>
Monthly Payment: <%= monthlyPayment %><br>
Total Payment: <%= totalPayment %><br></body></html>
JSP Directives

- A JSP directive is a statement that gives the JSP engine information about the JSP page.

  ```
  <%@ directive attribute="value" %>, or
  <%@ directive attribute1="value1"
    attribute2="value2"
    ...
    attributeN="valueN" %>
  ```

- For example, if your JSP page uses a Java class from a package other than the `java.lang` package, you have to use a directive to import this package.
Three JSP Directives

Three possible directives are the following: page, include, and tablib.

- **page** lets you provide information for the page, such as importing classes and setting up content type. The page directive can appear anywhere in the JSP file.

(c) Pearson and P. Fodor (CS Stony Brook)
Three JSP Directives

Three possible directives are the following: page, include, and tablib.

- **include**: lets you insert a file to the servlet when the page is translated to a servlet. The `include` directive must be placed where you want the file to be inserted.
Three JSP Directives

Three possible directives are the following: page, include, and tablib.

*page*

*tablib* lets you define custom tags.

*include*
Attributes for page Directives

import
contentType
session
buffer
autoFlush
isThreadSafe
errorPage
isErrorPage

Specifies one or more packages to be imported for this page. For example, the directive `<%@ page import="java.util.*, java.text.*" %>` imports java.util.* and java.text.*.
### Attributes for page Directives

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>import</td>
<td>Specifies the MIME type for the resultant JSP page. By default, the content type is text/html for JSP. The default content type for servlets is text/plain.</td>
</tr>
<tr>
<td>contentType</td>
<td></td>
</tr>
<tr>
<td>session</td>
<td></td>
</tr>
<tr>
<td>buffer</td>
<td></td>
</tr>
<tr>
<td>autoFlush</td>
<td></td>
</tr>
<tr>
<td>isThreadSafe</td>
<td></td>
</tr>
<tr>
<td>errorPage</td>
<td></td>
</tr>
<tr>
<td>isErrorPage</td>
<td></td>
</tr>
</tbody>
</table>

(c) Pearson and P. Fodor (CS Stony Brook)
Specifies a **boolean** value to indicate whether the page is part of the session. By default, **session** is **true**.
Specifies the output stream buffer size. By default, it is 8KB. For example, the directive `<%@ page buffer="10KB" %>` specifies that the output buffer size is 10KB. The directive `<%@ page buffer="none" %>` specifies that a buffer is not used.
Attributes for page Directives

- import
- contentType
- session
- buffer
- autoFlush
- isThreadSafe
- errorPage
- isErrorPage

Specifies a **boolean** value to indicate whether the output buffer should be automatically flushed when it is full or whether an exception should be raised when the buffer overflows. By default, this attribute is **true**. In this case, the buffer attribute cannot be **none**.
Attributes for page Directives

import
contentType
session
buffer
autoFlush
isThreadSafe
errorPage
isErrorPage

Specifies a **boolean** value to indicate whether the page can be accessed simultaneously without data corruption. By default, it is **true**. If it is set to false, the JSP page will be translated to a servlet that implements the `SingleThreadModel` interface.

(c) Pearson and P. Fodor (CS Stony Brook)
errorPage specifies a JSP page that is processed when an exception occurs in the current page. For example, the directive `<%@ page errorPage="HandleError.jsp" %>` specifies that HandleError.jsp is processed when an exception occurs.

- isErrorPage specifies a boolean value to indicate whether the page can be used as an error page. By default, this attribute is false.
Use the Loan class to simplify ComputeLoan. You can create an object of Loan class and use its monthlyPayment() and totalPayment() methods to compute the monthly payment and total payment.

Import a class. The class must be placed in a package (e.g. package chapter40).
JavaBeans Component in JSP

- A class is a JavaBeans component if it has the following three features:
  - The class is public.
  - The class has a public constructor with no arguments.
  - The class is serializable. (This requirement is not necessary in JSP.)
Using JavaBeans in JSP

- To create an instance for a JavaBeans component, use the following syntax:

  ```jsp
  <jsp:useBean id="objectName"
              scope="scopeAttribute" class="ClassName" />
  ```

- This syntax is equivalent to

  ```jsp
  <% ClassName objectName = new ClassName() %>
  ```

- except that the scope attribute specifies the scope of the object.
Specifies that the object is bound to the application. The object can be shared by all sessions of the application.
Scope Attributes

**application**
**session**
**page**
**request**

Specifies that the object is bound to the client’s session. Recall that a client’s session is automatically created between a Web browser and Web server. When a client from the same browser accesses two servlets or two JSP pages on the same server, the session is the same.
The default scope, which specifies that the object is bound to the page.
Scope Attributes

Specifies that the object is bound to the client’s request.
How Does JSP Find an Object

- When `<jsp:useBean id="objectName" scope="scopeAttribute" class="ClassName" />` is processed, the JSP engine first searches for the object of the class with the same id and scope.

- If found, the preexisting bean is used; otherwise, a new bean is created.
Another Syntax for Creating a Bean

Here is another syntax for creating a bean using the following statement:

```jsp
<jsp:useBean id="objectName" scope="scopeAttribute" class="ClassName" >
    some statements
</jsp:useBean>
```

The statements are executed when the bean is created. If the bean with the same id and className already exists, the statements are not executed.
Example: Testing Bean Scope
This example creates a JavaBeans component named Count and uses it to count the number of visits to a page.

```html
<%@ page import = "chapter40.Count" %>
<jsp:useBean id = "count" scope = "application" class = "chapter40.Count">
</jsp:useBean>
<html>
<head>
    <title>TestBeanScope</title>
</head>
<body>
    <h3>Testing Bean Scope in JSP (Application)</h3>
    <% count.increaseCount(); %>
    You are visitor number <%= count.getCount() %>
    From host: <%= request.getRemoteHost() %>
    and session: <%= session.getId() %>
</body>
</html>
```
<%@ page import = "chapter40.Count" %>
<jsp:useBean id="count" scope="application" class="chapter40.Count">
</jsp:useBean>

package chapter40;

public class Count {
    private int count = 0;

    /** Return count property */
    public int getCount() {
        return count;
    }

    /** Increase count */
    public void increaseCount() {
        count++;
    }
}

<!-- TestBeanScope.jsp -->

You are visitor number <%= count.getCount() %><br>
From host: <%= request.getRemoteHost() %>
and session: <%= session.getId() %>

Testing Bean Scope in JSP (Application)

(c) Pearson and P.Fodor (CS Stony Brook)
Getting and Setting Properties

By convention, a JavaBeans component provides the get and set methods for reading and modifying its private properties. You can get the property in JSP using the following syntax:

```jsp
<jsp:getProperty name="bean" Idproperty="age" />
```

This is equivalent to

```jsp
<%= beanId.getAge() %>
```
Getting and Setting Properties, cont.

You can set the property in JSP using the following syntax:

```xml
<jsp:setProperty name="beanId" property="age" value="30" />
```

This is equivalent to

```jsp
<% beanId.setAge(30); %>
```
Often properties are associated with input parameters. Suppose you want to get the value of the input parameter named score and set it to the JavaBeans property named score. You may write the following code:

```jsp
<% double score = Double.parseDouble(request.getParameter("score")); %>
<jsp:setProperty name="beanId" property="score" value="<%= score %>" />
```
Associating Properties with Input Parameters

This is cumbersome. JSP provides a convenient syntax that can be used to simplify it as follows:

```jsp
<jsp:setProperty name="beanId" property="score" param="score" />
```

Instead of using the value attribute, you use the param attribute to name an input parameter. The value of this parameter is set to the property.
Associating All Properties

Often the bean property and the parameter have the same name. You can use the following convenient statement to associate all the bean properties in beanId with the parameters that match the property names.

\[ <\text{jsp:setProperty name="beanId" property="*" }/> \]
Example: Computing Loan Using JavaBeans

Use JavaBeans to simplify Example 40.3 by associating the bean properties with the input parameters.

<!-- ComputeLoan.jsp -->
<html>
<head><title>ComputeLoan Using the Loan Class</title></head>
<body>
<%@ page import = "chapter40.Loan" %>
<jsp:useBean id="loan" class="chapter40.Loan"></jsp:useBean>
<jsp:setProperty name="loan" property="*" />
Loan Amount: <%= loan.getLoanAmount() %><br>
Annual Interest Rate: <%= loan.getAnnualInterestRate() %><br>
Number of Years: <%= loan.getNumOfYears() %><br>
Monthly Payment: <%= loan.monthlyPayment() %><br>
Total Payment: <%= loan.totalPayment() %><br></body>
</html>
Example: Computing Factorials Using JavaBeans

Create a JavaBeans component named `FactorialBean` and use it to compute the factorial of an input number in a JSP page named FactorialBean.jsp.
<!-- FactorialBean.jsp -->
<%@ page import = "chapter40.FactorialBean" %>
<jsp:useBean id="factorialBeanId" class="chapter40.FactorialBean" >
</jsp:useBean>
<jsp:setProperty name="factorialBeanId" property="*" />

<HTML>
<HEAD>
<TITLE>Associating the bean properties with the input parameters.</TITLE>
</HEAD>
<BODY>
<H3>Compute Factorial Using a Bean</H3>

Compute Factorial Using a Bean
</H3>

<FORM method="post">
Enter new value: <INPUT NAME="number"><BR><BR>
<jsp:getProperty name="factorialBeanId" property="number" />

Getting number

<jsp:getProperty name="factorialBeanId" property="number" />
is

<%@ page import="java.text.*" %>
<% NumberFormat format = NumberFormat.getNumberInstance(); %>
<%= format.format(factorialBeanId.getFactorial()) %>
</FORM>
</BODY>
</HTML>
package chapter40;

public class FactorialBean {
    private int number;
    /** Return number property */
    public int getNumber() {
        return number;
    }
    /** Set number property */
    public void setNumber(int newValue) {
        number = newValue;
    }
    /** Obtain factorial */
    public long getFactorial() {
        long factorial = 1;
        for (int i = 1; i <= number; i++)
            factorial *= i;
        return factorial;
    }
}

(c) Pearson and P.Fodor (CS Stony Brook)
Mixing a lot of Java code with HTML in a JSP page makes the code difficult to read and to maintain. You should move the Java code to a .java file as much as you can.
<html>
  <head>
    <title>FactorialBean</title>
  </head>
  <body>
    <h3>Compute Factorial Using a Bean</h3>
    <form method="post">
      Enter new value: <input name="number" />
      <input type="submit" name="Submit" value="Compute Factorial" />
      <input type="reset" value="Reset" />
      Factorial of
      <jsp:getProperty name="factorialBeanId" property="number" />
      is
      <%= NewFactorialBean.format(factorialBeanId.getFactorial()) %>
    </form>
  </body>
</html>
<!-- DisplayTime.jsp -->
<%@page pageEncoding = "GB18030"%>
<%@ page import = "chapter40.TimeBean" %>
<jsp:useBean id = "timeBeanId"
class = "chapter40.TimeBean" scope = "application"/>
<jsp:setProperty name = "timeBeanId" property = "*"/>
<html>
<head>
<title>Display Time</title>
</head>
<body><h3>Choose locale and time zone</h3>
Current time is
<%=
 timeBeanId.currentTimeString(timeBeanId.getLocaleIndex(),
 timeBeanId.getTimeZoneIndex()) %>
</body>
</html>
Forwarding Requests from JavaServer Pages

Web applications developed using JSP generally consist of many pages linked together. JSP provides a forwarding tag in the following syntax that can be used to forward a page to another page.

<jsp:forward page="destination" />
Example: Browsing Database Tables

This example creates a JSP database application that browses tables. When you start the application, the first page prompts the user to enter the JDBC driver, URL, username, and password for a database. After you login to the database, you can select a table to browse. Upon clicking the Browse Table Content button, the table content is displayed.
<!-- DBLogin.html -->
<html>
<head>
<title>DBLogin</title>
</head>
<body>
<form method = "post" action = "/DBLoginInitialization.jsp">

JDBC Driver
<select name = "driver" size = "1">
<option>sun.jdbc.odbc.JdbcOdbcDriver</option>
<option>oracle.jdbc.driver.OracleDriver</option>
</select><br />

JDBC URL
<select name = "url" size = "1">
<option>jdbc:odbc:ExampleMDBDataSource</option>
<option>jdbc:oracle:thin:@liang.armstrong.edu:1521:orcl</option>
<option>jdbc:oracle:thin:@localhost:1521:test</option>
</select><br />

Username <input name = "username" /><br />
Password <input name = "password" /><br />
<input type = "submit" name = "Submit" value = "Login" />
<input type = "reset" value = "Reset" />
</form>
</body>
</html>
package chapter40;
import java.sql.*;

public class DBBean {
    private Connection connection = null;
    private String username;
    private String password;
    private String driver;
    private String url;

    /** Initialize database connection */
    public void initializeJdbc() {
        try {
            System.out.println("Driver is " + driver);
            Class.forName(driver);
            // Connect to the sample database
            connection = DriverManager.getConnection(url, username, password);
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }

    /** Get tables in the database */
    public String[] getTables() {
        String[] tables = null;
        try {
            DatabaseMetaData dbMetaData = connection.getMetaData();
            ResultSet rsTables = dbMetaData.getTables(null, null, null, new String[] {"TABLE"});
        }
    }
}
int size = 0;
while (rsTables.next()) size++;
rsTables = dbMetaData.getTables(null, null, null,
    new String[] {"TABLE"});
tables = new String[size];
int i = 0;
while (rsTables.next())
    tables[i++] = rsTables.getString("TABLE_NAME");
}
catch (Exception ex) {
    ex.printStackTrace();
}
return tables;

/** Return connection property */
public Connection getConnection() {
    return connection;
}
public void setUsername(String newUsername) {
    username = newUsername;
}
public String getUsername() {
    return username;
}
public void setPassword(String newPassword) {
    password = newPassword;
} ...
<!-- DBLoginInitialization.jsp -->
<%@ page import = "chapter35.DBBean" %>
<jsp:useBean id = "dBBeanId" scope = "session" class = "chapter35.DBBean">
</jsp:useBean>
<jsp:setProperty name = "dBBeanId" property = "*" />
<html>
<head>
    <title>DBLoginInitialization</title>
</head>
<body>

<!-- Connect to the database -->
<%! dBBeanId.initializeJdbc(); %>

<% if (dBBeanId.getConnection() == null) { %>
    Error: Login failed. Try again.
<% } %>
<% } %>
</body>
</html>
Java Server Faces (JSF)

- JSF supports visual Web development.
- You can create a Web user interface using a tool without writing any code.
- JSF completely separates Web UI from Java code so the application developed using JSF is easy to debug and maintain.
Visual Web Design Using NetBeans

Create a Web project with Visual Web JavaServer Faces.
Creating UI in the Design Pane

Design your page by dragging components from the Palette.

Define a component’s behavior by adding code. Either double-click the component or right-click the component and choose its event handler.

Page layout is currently set to Grid Layout, which positions components at specific x and y coordinates. If you would like to use Flow Layout, which positions components left to right and top to bottom, change the Page Layout property to Flow Layout.
Creating UI in the Design Pane

Drop a Static Text and set its properties
Creating UI in the Design Pane

Write the code in the Java tab.
Examining the JSP File

Click the JSP tab to see the JSP file. Whenever you add, remove, or change the UI components in the Design pane, the contents in the JSP are also updated. It is possible to modify the JSP file directly, but it is not recommended for the new users. Modifying the JSP file mistakenly could corrupt the entire project. You can completely ignore the JSP file when using this tool.
JSF UI Components

Student Registration Form

Last Name: Smith
M: T
First Name: John

Gender: Male Female

Major: Computer Science
Minor: Computer Science Mathematics

Hobby: Tennis Golf Ping Pong

Remarks:
No remarks

Register

(c) Pearson and P.Fodor (CS Stony Brook)
JSF UI Components
JSF UI Containers

http://localhost:8080/jsfdemo/faces/Computeloan.jsp - Windows Internet Explorer

Loan Amount: 10000
Number of Year: 5
Annual Interest Rate: 5.5

http://localhost:8080/jsfdemo/faces/Computeloan.jsp - Windows Internet Explorer

Loan Amount: 10000
Number of Year: 5
Annual Interest Rate: 5.5
Monthly Payment: 191.01162171781937
Total Payment: 11469.697303069162

(c) Pearson and P.Fodor (CS Stony Brook)
JSF UI Containers
Binding Data with UI Components
Creating a New Database Connection

[Image of a database connection dialog box showing configuration options for MySQL (Connector/J driver), driver class, database URL, user name, and password.]
Creating a New Database Connection
Designing UI

(c) Pearson and P. Fodor (CS Stony Brook)
Modifying Query
Modifying Query

```
SELECT student.birthDate, student.street, student.zipCode, student.deptId
FROM student, enrollment, course
WHERE course.courseld = ?
AND student.ssn = enrollment.ssn
AND enrollment.courseld = course.courseld
```
Changing Table Layout
Session Tracking
Session Bean

```java
public class SessionBean1 extends AbstractSessionBean {

    /**
     * Construct a new session data bean instance.
     */
    
    public SessionBean1() {
        number = (int)(Math.random() * 100);
    }

    /**
     * This method is called when this bean is initialized.
     * Typically, this occurs as a result of session scope.
     */

    public void init() {
        // implementation
    }
}
```
Validating Input

Form1: tfName: Validation Error: Value is required.
Form1: tfAge: Validation Error: Value is required.
Form1: tfHeight: Validation Error: Value is required.

SSN: 1-2323-2323