SENG2050
Introduction to Web Engineering

Week 3

JSPs

Computer Labs

- Computer labs begin this week
- Task: Setting up Tomcat and writing your first set of Servlets
- **Vital for Assignment 1**

Assignment 1

- **Please use lab time to start A1!**
  - Good opportunity to get hints from instructor

- Don’t leave it until the last minute!!!

This Week

- More Servlets
- Java Server Pages (JSP)
- Java Beans

Disclaimer

DO NOT USE JSPs OR BEANS IN ASSIGNMENT 1

Part 1 - More Servlets

Recap: HTTP Request Data

- HTTP requests pass data through
  - **Query String**
  - **Message Body**
  - Headers
  - Cookies

- Headers and Cookies are mostly meta-data
  - Covered later in this course …

- Query String & Message Body data referred to as **‘Request Parameters’**
  - The primary method of sending data in a HTTP request

- We can access and use these from our Servlets

Query String Parameters

- All URI's have an optional **Query String** component
  - Starts after a '?' character

- Example: Google Search
  - www.google.com/search?q=Newcastle%20University
  - Domain: www.google.com
  - Path: search
  - Query: q=Newcastle%20University
    - One parameter named 'q' with value 'Newcastle University'

- The query string is composed of key-value pairs
  - Separated by '&' characters
  - E.g. www.foo.bar/path?name=Lin&course=SENG2050
    - Two parameters: 'name' and 'course'

- Pass data using **GET method**

Message Body Parameters

- The HTTP protocol defines ‘body parameters’
- Body parameters are sent as part of the HTTP body
  - Generally used to send ‘private’ data
- In HTML, sent through a **POST FORM** element
  - Represented by form inputs
- Best used for a **POST** request

```
<form action="/my-servlet" method="POST">
  Name: <input type="text" name="name" /> <br />
  Age: <input type="number" name="age" /> <br />
  <input type="submit" />
</form>
```

Accessing Request Parameters

- Both Query & Body parameters exposed through HttpServletRequest
  - Single API
  - Always sent as Strings – You may need to convert parameters!
- Two options:
  - getParameter(name): String - Get a single parameter
  - getParameterValues(name): String[] – Get an array of a parameter
    - Multiple values can be sent with the same parameter name!
- Do not need to differentiate between Query and Body parameters!
  - Tip: Avoid mixing query and body parameters

Accessing Request Parameters

```
public class MyServlet {
  public void doGet(HttpServletRequest req, HttpServlet Response resp) {
    String id = req.getParameter("id");

    // Do something with the parameters
    // Produce a response
  }

  public void doPost(HttpServletRequest req, HttpServlet Response resp) {
    String name = req.getParameter("name");
    String ageStr = req.getParameter("age");
    int age = Integer.parseInt(ageStr);

    // Do something with the parameters
    // Produce a response
  }
}
```

Linking Servlets

- **Servlets themselves are not linked**

- But the pages they generate can be
  - And used to send data between them

- Can result in some ‘re-sending’ of data
  - i.e. store some data on a page to be re-sent
  - But this is normal in a web application …
  - Tip: Use ‘hidden’ form inputs!

- To ’link’ the servlets, we simply pass request parameters
  - Either through the query string, or request body

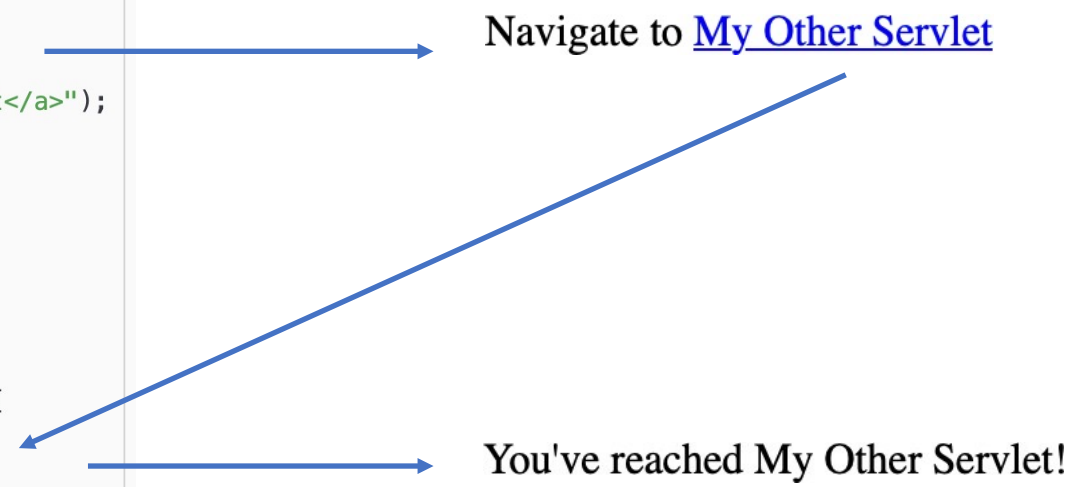
Linking Servlets (Hyperlink - Query)

```java
@WebServlet(urlPatters = {"/MyServlet"})
public class MyServlet {
  public void doGet(HttpServletRequest req, HttpServlet Response resp) {
    PrintWriter out = resp.getWriter();

    out.println(...);
    out.println("<p>");
    out.println("Navigate to <a href=''/MyOtherServlet'>My Other Servlet</a>");
    out.println("</p>");
    out.println(...);
  }
}

@WebServlet(urlPatters = {"/MyOtherServlet"})
public class MyOtherServlet {
  public void doGet(HttpServletRequest req, HttpServlet Response resp) {
    PrintWriter out = resp.getWriter();

    out.println(...);
    out.println("<p>You've reached My Other Servlet!</p>");
    out.println(...);
  }
}
```



Linking Servlets Example (Form - Body)

```
@WebServlet(urlPatters = {"/MyServlet"})
public class MyServlet {
  public void doGet(HttpServletRequest req, HttpServlet Response resp) {
    PrintWriter out = resp.getWriter();

    out.println(...);
    out.println("<form action='/MyServlet' method='POST'>");
    out.println("Name: <input type='text' name='name' /> <br />");
    out.println("Age: <input type='text' name='age' /> <br />");
    out.println("<input type='submit' />");
    out.println("</form>");
    out.println(...);
  }

  public void doPost(HttpServletRequest req, HttpServlet Response resp) {
    String name = req.getParameter("name");
    String age = req.getParameter("age");

    // Do something with the parameters
    // Produce a response
  }
}
```

Navigate to /MyServlet

On submit

Validating Request Parameters

- All request parameters MUST be validated on the server
  - **Validation with JavaScript does not secure an application**
- Web requests are inherently unreliable by design
  - A malicious user can change anything!
  - You can never assume request parameters are correct!
  - **I will break your assignments if you assume!**
- **Three cases must be accounted for**
  1. The parameter is valid
  2. The parameter is invalid
  3. The parameter is not present
- The we need to validate it:
  - Validation is through simple Java logic

Think of Assignment 1!!!

Validating Request Parameters

```
public void doPost(HttpServletRequest req, HttpServlet Response resp) {
  // List of errors
  List<String> errors = new ArrayList<>();

  // E.g. expect a name of at least length 10
  String name = req.getParameter("name");
  if (name == null) {
      // Name is invalid - not passed
      errors.add("Name is null");

  } else if (name == null || name.length() < 10) {
      // Name is invalid - less than 10 characters
      errors.add("Name is less than 10 characters");
  }

  // Expect an age greater than 0, but less than 100
  String ageStr = req.getParameter("age");
  int age;

  if (ageStr == null) {
    // No age passed
    errors.add("Age is null");
  } else {
    try {
      age = Integer.parseInt(ageStr);
    } catch (NumberFormatException ex) {
      // Age is invalid
      errors.add("Age is not a valid integer");
    }
  }

  if (age < 0 || age > 100) {
    // Age is outside of expected range
    errors.add("Age is less than 0 or greater than 100");
  }

  if (errors.isEmpty()) {
    // Success - we can process this request!
  } else {
    // Error - we need to present an error page
  }
}
```

Tip:
Make Utility methods!

Storing Application Data

- Two general options:
  - In memory (not persistent)
  - In a datastore (e.g. SQL database or file on disk)

- You can use standard Java IO in a Servlet application
  - But finding the correct file path is tricky …

File IO with Servlets

```
public class FileReadingServlet extends HttpServlet {

  public void doGet(HttpServletRequest req, HttpServlet Response resp) {
    ServletContext ctx = this.getServletContext();
    String fpath = ctx.getRealPath("/WEB-INF/test.txt");

    BufferedReader freader = new BufferedReader(new FileReader(fpath));

    String line = freader.readLine();
    // ... Continue reading the file ...

    freader.close();
  }
}
```

The ServletContext provides utility methods

getRealPath(String) finds a file in the application

You may need to create the file beforehand …

DO NOT HARD CODE FILE PATHS!

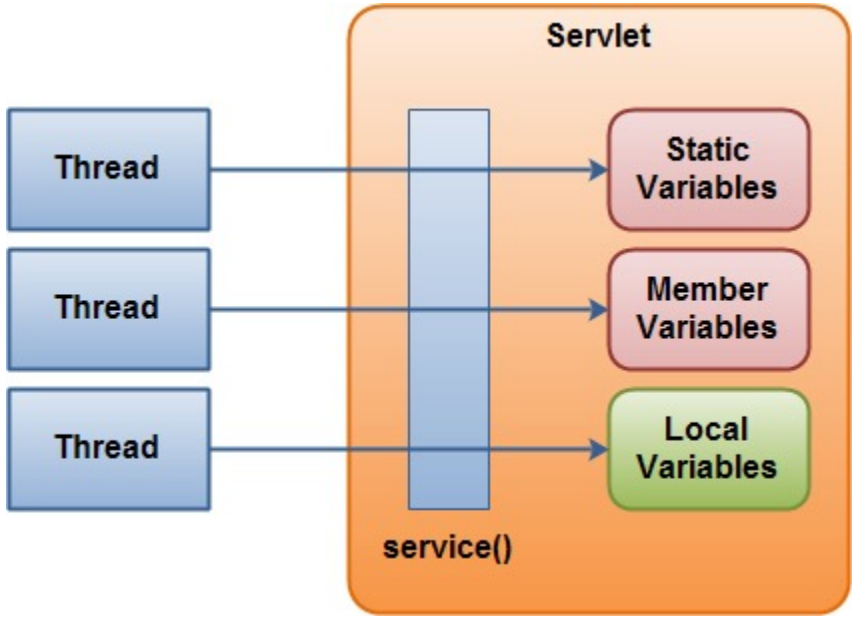
Storing Data in Servlets

- A Servlet is a ‘singleton’
  - There is only ever one instance of a Servlet
  - Created by the container (Tomcat) on startup
  - **The same Servlet object handles *all* mapped requests**

- Many different requests can ‘pass through’ the same servlet concurrently!

- This can cause problems in storing data *in* the Servlet class
  - i.e. a member - **member variables on Servlets is bad practice!**
  - **If a servlet has member variables – they can be concurrently accessed and modified**
  - **This can lead to some ‘strange’ and ‘unpredictable’ behavior!**

- We need to consider thread safety



Thread Safety

- Thread Safe code must be used in a multi-threaded environment
  - i.e. with Servlets
- When multiple threads are trying to ‘do the same thing’
  - i.e. executing the same code, trying to store/update the same data
  - There will not be any undefined behavior!
- This is through coordinating a threads ‘access’ to code
  - Using ‘threading’ or ‘synchronization’ primitive
- The simplest method in Java is the ‘**synchronized**’ keyword
  - But Java has a comprehensive set of classes and utilities to enforce thread safety
  - Covered in Operating Systems course

Java Synchronized Example – Not Thread Safe

```
public class ThreadSafeServlet extends HttpServlet {

  // Variable to store how many times
  // this servlet is accessed
  private long counter;

  public void doGet(HttpServletRequest req, HttpServlet Response resp) {
    // Generate the HTML page + print out counter
    PrintWriter out = resp.getWriter();
    out.println(...);
    out.println("<p>Times accessed: " + counter++ + "</p>");
    out.println(...);
  }
}
```

Java Synchronized Example – Not Thread Safe

```
public class ThreadSafeServlet extends HttpServlet {

  // Variable to store how many times
  // this servlet is accessed
  private long counter;

  public void doGet(HttpServletRequest req, HttpServlet Response resp) {
    // We need to get a copy of the counter
    // And store it as a variable in the method
    synchronized (this) {
      counter++;
    }

    // Generate the HTML page + print out counter
    PrintWriter out = resp.getWriter();
    out.println(...);
    out.println("<p>Times accessed: " + counter + "</p>");
    out.println(...);
  }
}
```

Java Synchronized Example – Thread Safe

```
public class ThreadSafeServlet extends HttpServlet {

  // Variable to store how many times
  // this servlet is accessed
  private long counter;

  public void doGet(HttpServletRequest req, HttpServlet Response resp) {
    // We need to get a copy of the counter
    // And store it as a variable in the method
    int countToPrint;
    synchronized (this) {
      countToPrint = counter++;
    }

    // Generate the HTML page + print out counter
    PrintWriter out = resp.getWriter();
    out.println(...);
    out.println("<p>Times accessed: " + countToPrint + "</p>");
    out.println(...);
  }
}
```

A member variable that is not thread safe

Needs to be synchronized on ALL read + write (this is why we use a ‘local’ copy)

Thread Safety

- Thread safety takes YEARS to understand
  - Lots of practice involved …
- **As a minimum, you need to make sure the method(s) saving data are thread safe!**
- Suggestion: Use a static method on another Java class

Thread Safety

```
public class MyFileManager {
  public static synchronized boolean saveSomething(String something) {
      // Handle saving something ...
      // But you need to do some validation first ...
      // And return an error if not ...
  }
}
```

Part 2 – Java Server Pages

Problem

- Good practice to separate the user interface (HTML+CSS) from the "business" logic (XML+Java)
  - Lets you develop each part independently – faster development
  - Lets graphic designers work on the interface while software engineers work on the logic – better end product
  - Lets you completely redesign the "look and feel" without changing the business logic – easier maintenance

So Far …

- Servlets contain
  - Business logic (Java code, request processing)
  - User interface generation (HTML)

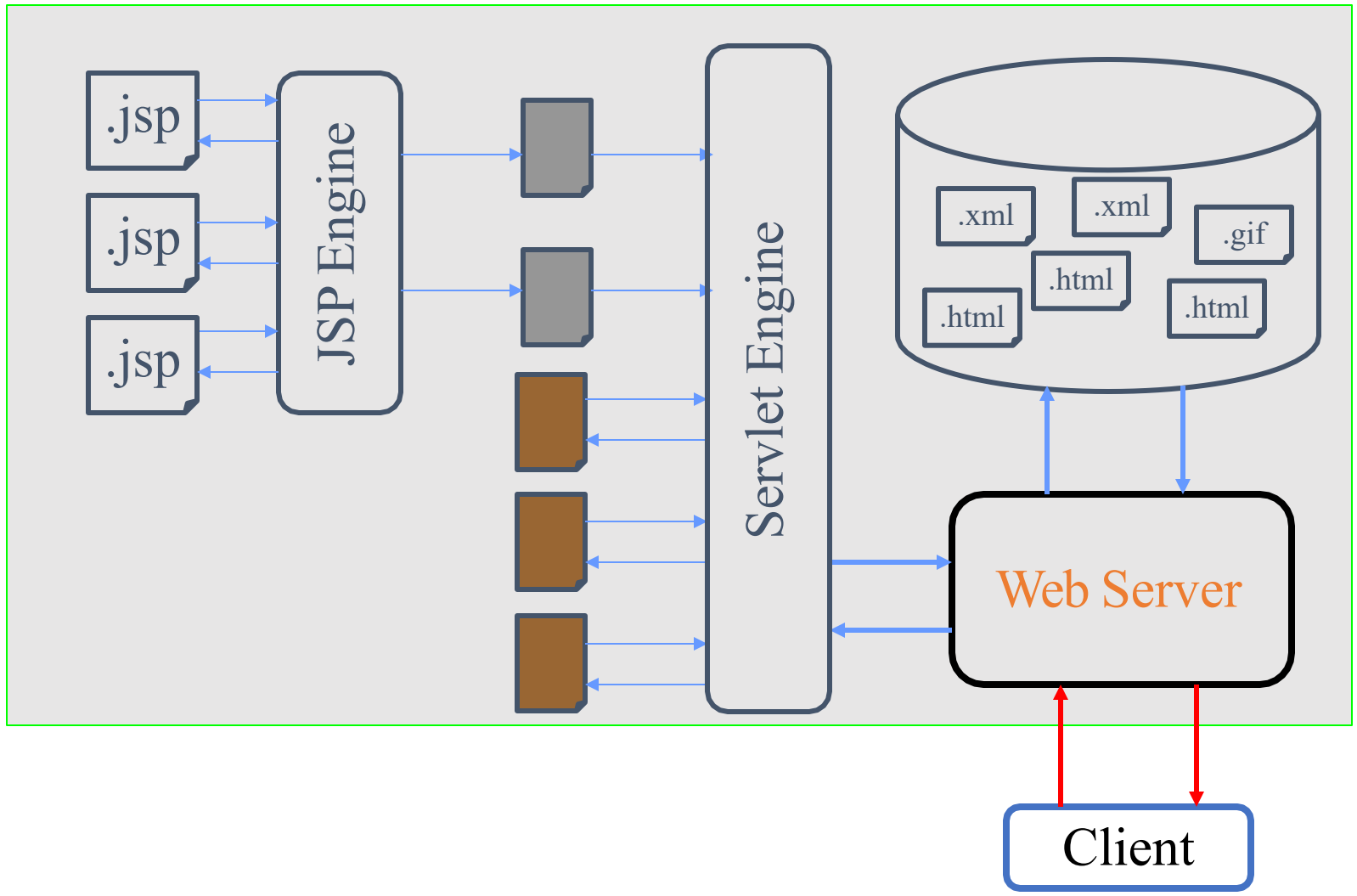
- Problems:
  - Very tedious to generate HTML
  - HTML generation combined with business logic
  - Not very maintainable

- Solution:
  - Decouple the HTML generation from Servlets
  - This is supported through Java Server Pages (JSP)

Java Server Pages

- JSP pages embed Java code
  - Allows generation of dynamic HTML pages
  - Use Java statements to manage generation of HTML
  - Similar to PHP
- Allows embedding of ‘special tags’ containing Java code
  - Servlet container finds these tags, compiles them & runs Java code
- Essentially just a servlet in disguise
  - JSPs are compiled to servlets
- Think:
  - Servlets as HTML embedded in Java
  - Think JSPs as Java embedded in HTML

Java Server Pages – JSP



Clarification - The Many ‘Javas’

- We have encountered many `Javas`

- Java Servlets
  - Java classes which process HTTP request *on the server*
  - Produces dynamic response to client

- JavaScript
  - A java-like language
  - Allows for dynamic behaviour *on the client*

- JSP
  - HTML page with embedded Java
  - Generates HTML *on the server*

Clarification - Overlap

- JavaScript vs JSP
  - There is no overlap
  - JavaScript executed on the client, JSP on the server
  - Complementary – JSP produces dynamic page, JavaScript dynamic behaviour

- Servlets vs JSP
  - Can overlap, but depends on how they're used
  - Servlets are only for processing a request (business logic)
    - **NOT FOR PRODUCING HTML**
  - JSPs are for producing HTML
    - **NOT FOR BUSINESS LOGIC**

Java Server Pages

- Dynamically Compiled by the Servlet Container
  - Cached between accesses
  - Recompiled upon changes (live)

- Best Practices:
  - Place business logic into separate Java objects
    - These are our Java Beans
    - Helps separate presentation from business logic
  - Defines HTML pages as JSPs
    - Compiled to a Servlet, used to generate pages
  - Use Servlets to ‘control’ request handling

JSP Example

```
<html>
<head><title>First JSP</title></head>
<body>
  <% double num = Math.random(); %>
  <% if (num > 0.95) { %>
      <h2>You'll have a luck day!</h2><p>(<%= num %>)</p>
  <% } else { %>
      <h2>Well, life goes on ... </h2><p>(<%= num %>)</p>
  <% } %>
  <a href="<%= request.getRequestURI() %>">
    <h3>Try Again</h3>
  </a>
</body>
</html>
```

HTML Page

Special tags with embedded Java code

Use of control statements

JSP Scripting Elements

- The ‘special tags’ are JSP Scripting Elements

- The JSP file defines the ‘template’ of a HTML page
  - With embedded Java to control generation

- JSP Processor will ‘pass through’ HTML
  - Converted to out.write(…)
  - Referred to as the ‘template text’
  - **But will execute the Java code**

- Declares Scripting elements to control generation of page
  - Used to nest Java code to modify output
  - Typically just control statements (e.g. if, for, switch, while, do)
  - Or print values
  - But can be *any* Java code (with some restrictions …)

JSP Scripting Elements

- JSP declares 3 scripting elements
  - Scriptlets – Blocks of Java code
  - Expressions – Computed values printed to the page
  - Declarations – Fields and Methods

- Each has its own use case:
  - Scriplets – Block-level statements (e.g. if) and local variables, method calls, …
  - Expressions – Printing some value (could be local or result of method call)
  - Declarations – Declare logic reused in JSP page **(NOT FOR BUSINESS LOGIC)**

JSP Scriptlets

- Used to control JSP page
  - Copied into generated servlet
  - Executed upon request processing
  - Think loops used to generate a HTML table

- Syntax
  - `<% java statement %>`
  - `<jsp:scriptlet>java statement</jsp:scriptlet>`

JSP Scriptlets

Consider a for (counting) loop:

```
<h1>Counting</h1>
<ul>
  <% for (int i = 1; i <= 10; i++) { %>
    <li><%= i %></li>
  <% } %>
</ul>
```

JSP File (Counting.jsp)

```
<h1>Counting</h1>
<ul>
  <li>1</li>
  <li>2</li>
  <li>3</li>
  <li>4</li>
  <li>5</li>
  <li>6</li>
  <li>7</li>
  <li>8</li>
  <li>9</li>
  <li>10</li>
</ul>
```

Renders

JSP Scriptlets

```
<h1>Random Greeting</h1>
<% if (Math.random() < 0.5) { %>
  <h2>Have a nice day!</h2>
<% } else { %>
  <h2>Have a perfect day!</h2>
<% } %>
```

JSP File (Counting.jsp)

```
<h1>Random Greeting</h1>
<h2>Have a perfect day!</h2>
```

Renders

… or …

```
<h1>Random Greeting</h1>
<h2>Have a nice day!</h2>
```

Renders

JSP Expressions

- Simply used to display a value
  - Evaluated and inserted into out.write(..)
- Syntax
  - `<%= java expression %>`
  - `<jsp:expression>java expression</jsp:expression>`
- E.g.
  - `<%= Math.random()*10 %>`
- Shorthand for:
  - `<% out.println(expr); %>`

JSP Expressions

- Example: localhost:8080/Expressions.jsp?param=Columbia

```
<H2>JSP Expressions</H2>
<UL>
  <LI>
    Current time:<%= new java.util.Date() %>
  </LI>
  <LI>
    Your hostname:<%= request.getRemoteHost() %>
  </LI>
  <LI>
    Your session ID:<%= session.getId() %>
  </LI>
  <LI>
    Your Parameter is:<%= request.getParameter("param") %>
  </LI>
</UL>
```

JSP File (Expressions.jsp)

```
<H2>JSP Expressions</H2>
<UL>
  <LI>
    Current time: Thu Dec 05 08:21:00 UTC 1996
  </LI>
  <LI>
    Your hostname: 127.0.0.1
  </LI>
  <LI>
    Your session ID: (some id value)
  </LI>
  <LI>
    Your Parameter is: Columbia
  </LI>
</UL>
```

Renders

JSP Declarations

- Used to declare methods or fields
  - Code pasted at the ‘top level’ (i.e. in the class body)
  - Does not produce output to the client
  - Typically just a method
  - But can even be used for an inner class

- Syntax
  - `<%! java method/field %>`
  - `<jsp:declaration>java method/field</jsp:declaration>`

JSP Declarations

Field Declaration

Field Access

```
<%! private int accessCount = 0; %>
<H2>Accesses to page since server reboot: <%= ++accessCount %></H2>
```

Method Declaration

```
<%!
  public java.util.Date PrintDate() {
      return (new java.util.Date());
  }
%>

<p style="font-family:Arial,sans-serif">
The current time is <%= PrintDate() %> </p>
```

Method Call
(JSP Expression)

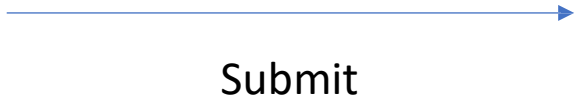
Where Do the JSPs Go?

- JSPs can be public or private
- Public JSPs go in the web root, accessed like a static resource
- Private JSPs go in WEB-INF, generally used by a Servlet (more in later weeks)

Using JSPs

- To start using JSPs, simply drop into the root of your web application
- Because JSPs are servlets ...
  - They can do *almost* anything a servlet can do
  - This means, they can handle both GET and POST requests
- But, the power of JSPs is generating pages
  - As a rule of thumb, they should only handle GET requests
  - Not the processing of a form!

```html
<h1>Feedback Form</h1>
<form action="/webapp/page.jsp" method="POST">
  Name: <input type="text" name="name" /> <br />
  Feedback: <br />
  <textarea name="feedback"></textarea> <br />
  <input type="submit" />
</form>
```



page.jsp

```jsp
<%
  String name = request.getParameter("name");
  String feedback = request.getParameter("feedback");
%>

<html>
<body>
  <p>
    Thank you, <%= name %>, for your feedback!
  </p>
</body>
</html>
```

JSP Resources

- Java Server Pages (JSP)
  - http://java.sun.com/products/jsp/
- Training Materials from the textbook
  - http://courses.coreservlets.com/Course-Materials/
- Web
  - http://www.jsptut.com/

Part 3 - Sessions & Cookies

Sessions & Cookies

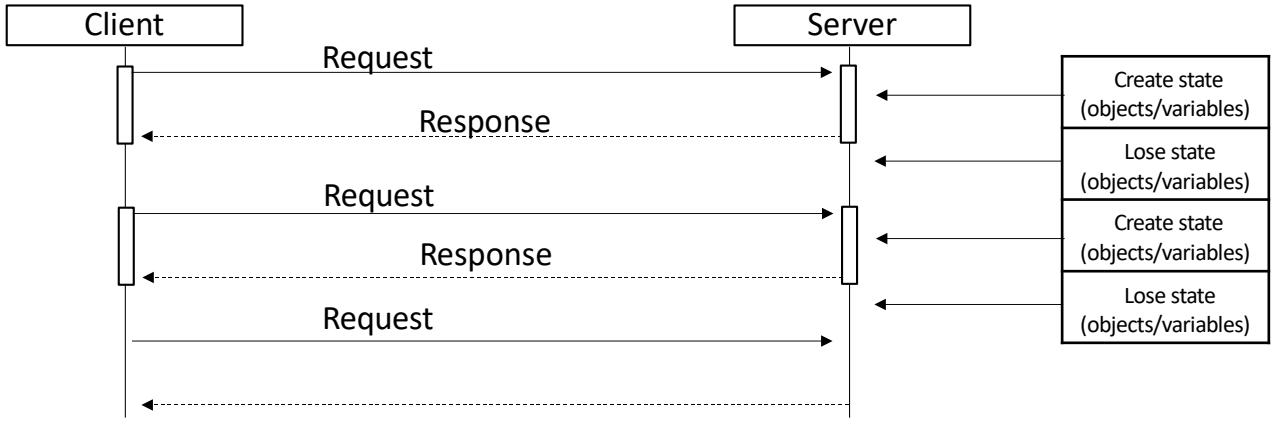
- Two separate, but interrelated concepts

- Both are mechanisms to provide state
  - Session - server state stored per user (e.g. user login, workflow data)
  - Cookies - client state in browser (e.g. cached data, `local` vars, session ids)

- State – remembering ‘who’ the client is
  - Something not provided by HTTP
  - Needed to remember who the user is

Stateless HTTP

- HTTP is request-response oriented
  - Stateless
  - Server does not store request state between requests
  - ‘forgets’ the client after processing request



Session

- Place to store data per user
  - Key-value store (stores data indexed by a string key)
- Commonly used to store username
  - After login – simply place a username in the store
- Will persist until: invalidated (logout), timeout, or server shutdown

Using Sessions

- Using sessions in servlets is quite straightforward, and involves:
  1. Accessing the ***session object*** associated with the current request
  2. Looking up information associated with the session
  3. Storing information in the session

- To access the ***session object***:
  - In a servlet ➔ *request.getSession();*
  - In a JSP ➔ *session;*
  - Both are *HttpSession* objects

Important Session Methods

- **`String getId()`**
  - Get the unique id for this session
- **`void invalidate()`**
  - Destroy the session associated with this request
- **`void setAttribute("name", value)`**
  - Store a value indexed by "name"
- **`Object getAttribute("name")`**
  - Get a value indexed by "name"
- **`Object removeAttribute("name")`**
  - Remove the value indexed by "name"

Session Example

```
public void doGet(HttpServletRequest request, HttpServletResponse response) {
  HttpSession session = request.getSession();
  String username = request.getParameter("username");
  session.setAttribute("username", username);
}
```

Storing an attribute in the session

```
public void doGet(HttpServletRequest request, HttpServletResponse response) {
  HttpSession session = request.getSession();
  String username = (String) session.getAttribute("username");
}
```

Retrieving an attribute from the session
This will be persistent for each user (until session destroyed)

Linking Users & Sessions

- As mentioned: HTTP is stateless
  - No way to correlate user with session
- How does Tomcat link sessions to users?
  - Cookies
  - But can also use a technique ‘URL Rewriting’
- Upon creation:
  - Cookie `JSESSIONID` is created, with unique session id per user
  - Sent on each request
  - Allows tracking sessions for user requests

Cookies

A cookie is…

- Small pieces of textual information (key-value pair)
- Created by client as part of the response
- When the client requests (again) something from the same server (or domain), it also sends the cookie information back to the server
  - All cookies sent on new request
  - Cookies updated from response (e.g. created, updated, or destroyed)
- Allows the server to store user-dependent information locals
  - Persists between `requests`
- Data can persist for (milli)seconds up to months

Cookies

Uses

- Identifying a user during an e-commerce session
  - E.g. putting items into a shopping cart
- Avoiding username and password – popular with low-security sites
- Customizing a site
  - Used by portals to remember look and feel selections
- Targeted advertising - directed rather than random ads
  - Has caused privacy concerns in prior years
  - E.g. Facebook and Google

Cookie Attributes

- Name
  - An identifier (e.g. JSESSIONID)
- Value
  - String of characters
- Domain
  - HTTP domain the cookie belongs to (e.g. localhost, google.com, facebook.com)
- Path
  - Valid url path for this cookie (e.g. /lab01 & /lab02 can have different cookies!)
- Expiry time
  - How long until this cookie is discarded
- Secure flag
  - Tells the browser to send this via SSL

Cookies

- Java provides `javax.servlet.http.Cookie`
  - Public methods for manipulating cookies in both Java Servlets and JSPs
- `Cookie(String` ***name*****`,`** `String` ***value***`)`
  - Create a new cookie with ***name*** = ***value***
- `Cookie[] request.getCookies()`
- `response.addCookie(Cookie` ***cookie***`)`
  - Pass cookies to and from the browser

Cookies

- The name of the cookie
  - `String getName()`
  - `void setName(String` ***name***`)`
- Sets the value of the cookie
  - `String getValue()`
  - `void setValue(String` ***value***`)`
- Time in seconds before cookie expires
  - `int getMaxAge()`

Cookies

1. Create a cookie using the constructor `Cookie`

```
Cookie c = new Cookie("userID", "c3014254");
```

1. Set life span for the cookie

```
c.setMaxAge(60*60*24*7); // one week
c.setMaxAge(0); //To discard cookie
```

3. Add a cookie

```
response.addCookie(c);
```

Part 4 – Java Beans

Business Logic

- Business logic is the part of a program that implements the requirements of the program
  - Solves the ‘businessy part’ of the task
- E.g. in A1, your business logic includes:
  - Managing the parent/child relationship
  - Storing the messages
  - …
- But your application will be more than just these features!
  - Need code for getting request parameters, validating …
  - Need code for persisting data …
  - Generating HTML pages …
- Good practice to separate these different types of code – **Good Software Engineering**
  - i.e. the business logic away from everything else

Java Beans

- JSPs were created to separate HTML from Java code
  - Start of Good Software Engineering

- But, where do we place the business logic?
  - In the servlet?
    - -> Becomes tangled with request processing.
  - In the JSP?
    - -> Becomes tangled with page generation.

Good Design – Web Engineering

E.g. consider a feedback form submitted to a JSP

```
<%
   String name =
    request.getParameter("name");
%>
```

…

```
<p>
 <%= name %>.
 Thank you for your feedback,
 Please visit us again.
</p>
```

…

Good Design – Web Engineering

E.g., the “temperature” form is submitted…

```
<%!
  double kelvinToCelcius(double k) {…}
  double kelvinToFarenheit(double k) {…}
%>
<%
  double kelvin =
    request.getParameter("temperature");
%>
```

…

```
  <td><%= kelvinToCelcius() %></td>
  <td><%= kelvinToFarenheit() %></td>
```

Good Design – Web Engineering

- The idea was to separate the Java from the HTML; however, in these examples, the Java and HTML are all mixed together!
  - Exactly! JSP itself is not enough to provide real separation of interface and business logic
- A solution:
  - Create a new Java class that does all the calculating
  - Create an instance of the class in your JSP, then access its methods to get the required dynamic values

Java Beans

- Three components of web applications
  - Request Handling (Servlets)
  - Page Generation (JSPs)
  - Business Logic **(Beans)**

- Servlets – Good at processing request
- JSP – Good at generating pages
- Beans – Good at implementing program logic

- But, what is a bean?

Java Beans

- Beans
  - Just a `Plain Old Java Class' or POJO
  - Create a managed instance in JSP or manual instance in Servlets
  - Use it to call business logic (i.e. process the *request input*)

- Beans are a convention
  - Encapsulates one or more objects into a single object (the bean)
  - Allows the bean to be passed around, and shared
  - JSPs have first-class support for beans
  - Can also use them from Servlets

Bean Rules

- Zero argument constructor
- All private fields
- Properties through getters and setters
  - E.g. field ***name*** accessed through ***getName(), setName(String name)***
- Serializable (able to be persistently stored and loaded)
- It should not contain any required event-handling methods

Bean Example

```
public class PersonBean implements java.io.Serializable {
  private String name;
  private boolean deceased;

  public PersonBean() { }

  public String getName() { return this.name; }
  public void setName(String name) { this.name = name; }

  public boolean isDeceased() { return this.deceased; }
  public void setDeceased(boolean deceased) { this.deceased = deceased; }
}
```

Bean Rules

- All these rules are fallacies
  - None of these are required …
  - More a set of guidelines
- Zero argument constructor …
  - If your JSP instantiates the bean
  - Beans can be `injected` by a Servlet, or other constructs …
- No public fields …
  - More of a convention for good design
  - Can access fields directly with Java code
  - But not with JSP tags (coming up!)
- Accessing values with getXXX(), isXXX(), setXXX(…)
  - Only if you need to get or set
  - Can have getter without setter, and vice versa
- Class should be `Serializable`
  - Only if it needs to be persisted between restarts
- No event handlers …
  - Everything is a type of `event`
  - More concerned with UI events

Why Use Beans

- If you obey the Java Bean rules...
  - Other programmers, and even automatic programming tools, will be able to use your classes more easily
  - You will be able to reuse others' beans

- If you do not...
  - You have to explicitly write interfaces between JSP and your classes
  - It will be harder for others to use your classes

Using Beans

- JSP has first-class support
  - Dedicated tags for interacting with beans

- Declaration:

```
<jsp:useBean id="beanName"
  scope="page|request|session|application"
  class="my.bean.classname" />
```

- Get Property:

```
<jsp:getProperty name="beanName"
  property="propertyName" />
```

- Set Property:

```
<jsp:setProperty name="beanName"
  property="propertyName"
  value="newValue" />
```

jsp:useBean

- `<jsp:useBean id="`*beanName*`" scope="page|request|session|application" class="`*Classname*`" />`
  - Creates an instance of *Classname* and binds it to the variable *beanName*
  - The bean is then exposed as a "local variable"

```
<jsp:useBean id="pb" class="PersonBean"/>
<% pb.setName("John"); %>
<% pb.setDeceased(False); %>
Name: <%= pb.getName() %> is <%= pb.isDeceased() %>
```

jsp:useBean

- ‘scope’ controls the visibility of the bean
  - Determines how it is shared between JSPs and Servlets
- scope="page" – the bean can be used within this page only.
- scope="request" – the bean can be used in any JSP processing the same request.
- scope="session" – the bean is stored in the users session
  - Stores data unique for the user
- scope="application" – the bean can be used in any page in the current application
- **Need to consider thread safety for page, session and application scopes!**

jsp:getProperty

Once you have a Java Bean instance, you can use special JSP actions to set and get its values

```
<jsp:getProperty name="beanInstanceName"
      property="propertyName" />
```

- Uses the bean's accessor method
- Calls *beanInstanceName*`.get`*PropertyName*`()` or *beanInstanceName*`.is`*PropertyName*`()`

```
<jsp:getProperty name="pb" property="name" />
```

is equivalent to `<%= pb.getName() %>`

jsp:setProperty

```
<jsp:setProperty name="beanInstanceName"
     property="propertyName"
     value="newValue" />
```

- Uses the bean's mutator method, *beanInstanceName*`.set`*PropertyName*`(`*newValue)*

```
<jsp:setProperty name="pb"
     property="name"
     value="Hayden" />
```

is equivalent to: `<% pb.setName(" Hayden "); %>`

jsp:setProperty

```
<jsp:setProperty name="beanInstanceName"
          property="propertyName" param=”paramName" />
```

- We can also set property values from request parameters!

```
<jsp:setProperty name=“pb"
          property=“name" param=”username" />
```

is equivalent to:

```
<% pb.setName(request.getParameter(“username”)); %>
```

Or

```
<jsp:setProperty name=“pb" property=“name"
    value=”<%= request.getParameter(“username”) %>" />
```

jsp:setProperty

- This tag automatically converts the parameter `String` into any of the Java built-in types
  - `byte`, `short`, `int`, `long`, `float`, `double`, `boolean`, `char`
  - And equivalent objects (`Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`, `Boolean`, `Character`)
  - But may throw exceptions

- Bonus!

Using Beans

- We can then use these tags to access and modify our beans
  - E.g. our feedback form

```
<%
  String name = request.getParameter("name");
  String feedback = request.getParameter("feedback");
%>

<html>
<body>
  <p>
    Thank you, <%= name %>, for your feedback!
  </p>
</body>
</html>
```



```
package my.pkg;

public class FormBean {
  private String name;
  private String feedback;

  public void setName(String name) { ... }
  public void setFeedback(String feedback) { ... }

  public String getName() { ... }
  public String getFeedback() { ... }
}
```

```
<jsp:useBean id="feedbackForm" class="my.pkg.FormBean" scope="request" />
<jsp:setProperty name="feedbackForm" property="name" param="name" />
<jsp:setProperty name="feedbackForm" property="feedback" param="feedback" />

<html>
<body>
  <p>
    Thank you, <jsp:getProperty name="feedbackForm" property="name" />,
    for your feedback!
  </p>
</body>
</html>
```

JSP and Java Beans

- Why use `jsp:getProperty` and `jsp:setProperty`?
  - It often requires ‘more code’
  - But, enforces use of the Java Bean interface
    - you can’t bypass it, thus can’t break all the nice “separation” features
  - You can reuse other people’s beans without writing any Java code
  - Can even be used by non-programmers – e.g., Web page designers only need `jsp:useBean`, `jsp:getProperty` and `jsp:setProperty` to have full access to a beans dynamic behaviour
    - But Java programmers may use the getXXX() or setXXX() equivelents

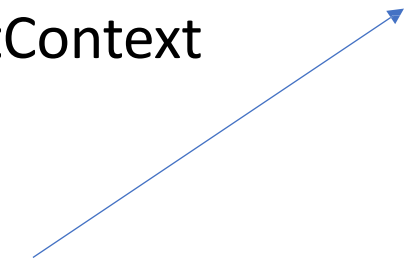
Beans & Servlets

- Common misconception that Servlets cannot access beans
  - Servlets ***can*** access beans

- Beans are stored in our 3 `scoped objects`
  - ‘request’ – HttpServletRequest
  - ‘session’ – Session
  - ‘application’ - ServletContext

- E.g. for session bean

  (Replace with other scoped objects)



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

Bean myBean = (Bean) session.getAttribute("beanName");
if (myBean == null) {
  myBean = new Bean();
  session.setAttribute("beanName", myBean);
}
```

Using Beans

- As a rule of thumb:
  - **JSPs should not contain Java code!**
  - (excluding if/for/while/etc – we replace these later!)
- All business logic should be in the beans
  - We can then access these in Servlets and JSPs (as required)
- This allows a ‘nice’ software design
- Will build upon this next week with MVC designs
  - Needed for Assignment 2

Thinking Java Beans

- Using Java Beans successfully requires a particular way of thinking
  - To implement `int s = sum(21, 12)` as a bean…

```
private int arg1 = 0;
public void setArg1(arg1) { this.arg1 = arg1; }
public int getArg1() { return arg1; }

// similar field, getter, setter for arg2

public int getSum() { return getArg1() + getArg2() };
```

Thinking Java Beans

- Then …
  - Place useBean and setProperty in top of JSP file

```
<jsp:useBean id="sum" class="Sum" />
<jsp:setProperty name="sum" property="arg1" param=”lhs" />
<jsp:setProperty name="sum" property="arg2" param=”rhs" />
```

  - And place getProperty in body of HTML template

```
<jsp:getProperty name="sum" property="sum"/>
```

  - Access with URL params
    - /summing.jsp?lhs=21&rhs=42
    - Will print ‘63’

Thinking Java Beans

- Note that I used `getArg1()` and `getArg2()` inside the `getSum()` method
  - ✓This is more good coding practice
  - ✓If I decide to change the way `arg1` is stored, then I only have to change `getArg1()` and `setArg1()` – other methods which need to access or mutate the value of `arg1` won't need to be changed
  - ✓As a general rule, only the corresponding set and get methods should directly access a private attribute – all other methods (even within the same class) should go through these methods

Thinking Java Beans

The general pattern for using a Java Bean is:

1. Store all “inputs” in the bean using `jsp:setProperty`
2. Get the “results” from the bean using `jsp:getProperty`
   - ✓ The get method will calculate the results from the current set inputs

- ✓ But there are exception to this rule
  - ✓ Sometimes we use beans to only ‘get’ in a JSP page
  - ✓ The instantiation of the bean can be handled by a Servlet
  - ✓ More in later weeks …

Java Bean Resources

- JavaBeans Tutorial
  - ✓http://java.sun.com/docs/books/tutorial/ javabeans/
- Lots of Java Bean Resources, including links to free beans and books
  - ✓http://www.freeprogrammingresources.com/ javabean.html
- JavaBeans Tutorial
  - ✓ http://java.sun.com/docs/books/tutorial/ javabeans/