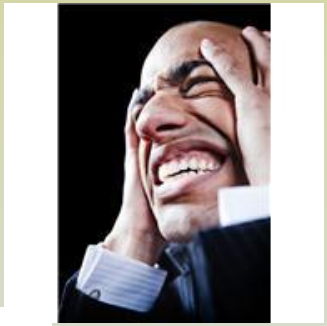
JAVA SERVER PAGES AND MODEL VIEW CONTROLLER ARCHITECTURE

Knower, Known, and Process of Knowing

Main idea of JSP

- separate display from processing, i.e., separate html from java
  - servlet is java with some html mixed in
    - a little plain java code
    - and lots of enclosed HTML
      - out.println(“ … \” … \” … “)
    - writing and maintaining quickly becomes a headache



  - Jsp is html with a little java mixed in
    - a little java code bracketed in
      - <% …. %> , etc
    - and lots of plain HTML

- There are two types of data in a JSP page:
  - Template Data: The static part, copied directly to response
    - static HTML
  - JSP Elements: The dynamic part, translated and executed by the container
    - typically generate additional HTML portions of the page
    - can execute arbitrary Java code

What are Java Server Pages?

JavaServer Pages (JSP) are a technology for developing web pages that support dynamic content, allowing developers to insert java code in HTML pages by making use of special JSP tags, most of which start with <% and end with %>. Eliminates the need to code complex output streams from the response object to produce a view.

```
//Hello.jsp
<form action="HelloName.jsp" method="get"><br/>
Name: <input type = "text" name = "name"/><br/>
<input type = "submit" value = "Submit"/>
</form>
```

```
//HelloName.jsp
<body>
Hello, <%=request.getParameter("name") %>
</body>
```

Four types of JSP elements:

- **Script**: embedded Java statements <% %> <%= %> …
- **Directive**: page level operations for the translation phase.

<%@ page %>

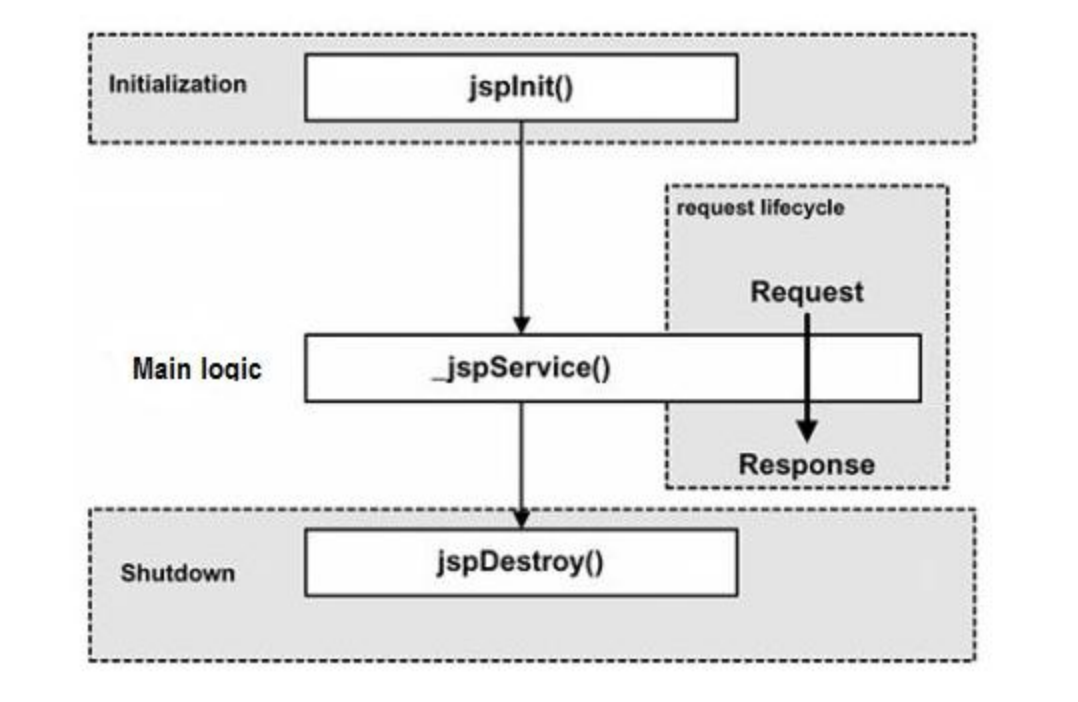
- **EL Expression**: more convenient and powerful than a JSP expression ${ } vs <%= %>
- **Action**: JSP “functions” that encapsulate some commonly used functionality <c:foreach />

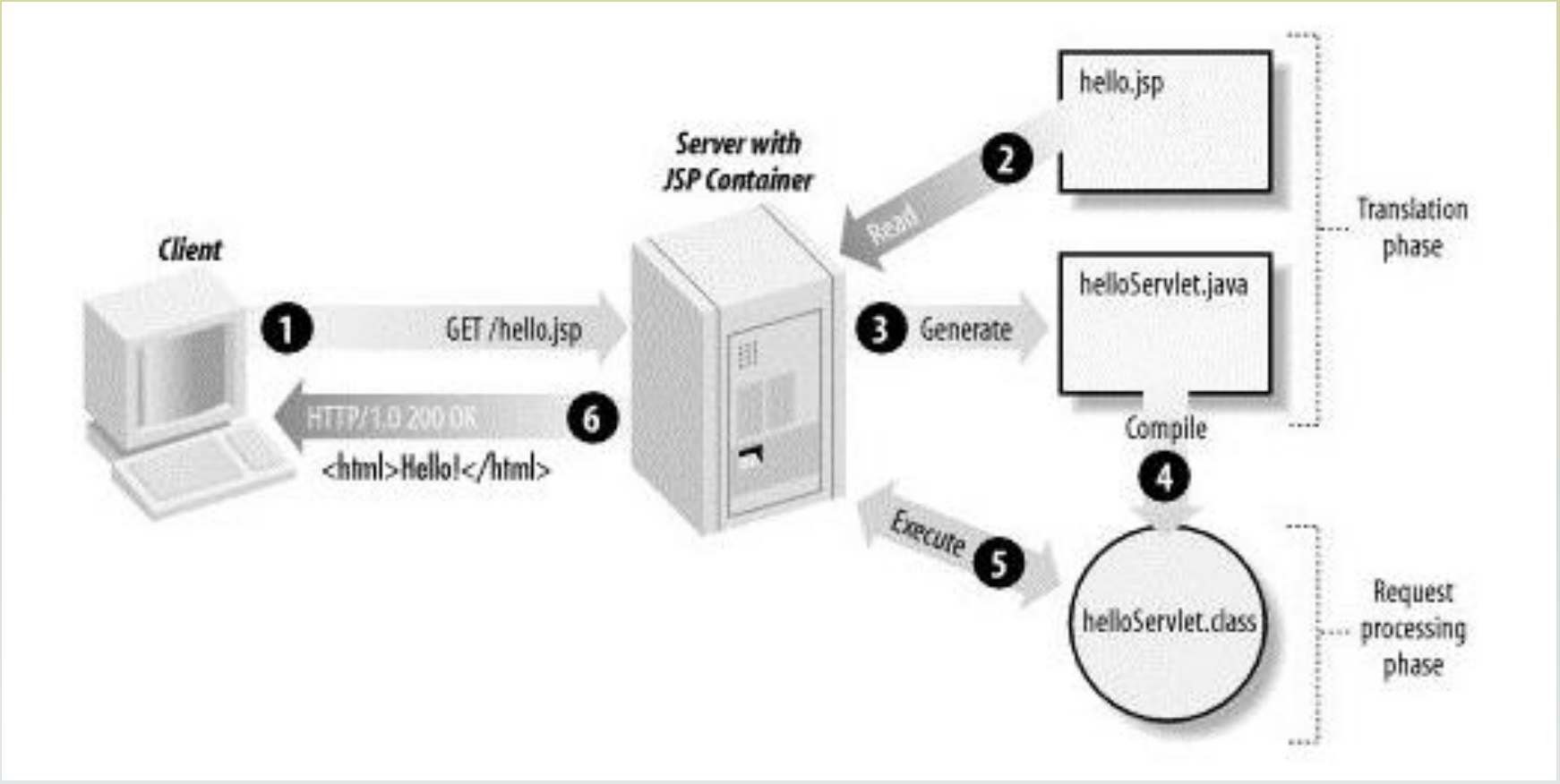
JSP scripting elements:

- **Declaration**:
  - <%! Inserts instance variable and method declarations into servlet
  - <%! Java declaration statements %>
  - <%! int count = 0; %>
- **Scriptlet**:
  - Inserts Java statements inside service method
    - <% Java statements %>
    - <% count = count * 10; %>
    - <% inserts into the service method
- **Expression**:
  - expects a Java expression, which it puts inside 'out.print' in service method
  - <%= Java expression %>
  - wraps it inside a print statement
  - <%= ++count %> becomes … out.print(++count); … in service method
- **Comment**:
  - <%-- jsp comment --%>
  - contrast with HTML comment
    - <!– Comment -->
    - HTML comments get sent to the browser (part of the HTML)
    - JSP elements all processed by container and do not appear in generated HTML

JSP Life Cycle

- A JSP life cycle can be defined as the entire process from its creation till its destruction -- this is similar to the servlet life cycle with an additional step of compiling a JSP into servlet.
- The following are the steps of evolution of a JSP, which are typically initiated by a browser request for a jsp (e.g. index.jsp).
  - Compilation
  - Initialization
  - Execution
  - Cleanup




hello.jsp
Server with
JSP Container
2
Read
Translation
phase
Client
helloServlet.java
1
GET /hello.jsp
3
Generate
HTTP/1.0 200 OK
6
<html>Hello!</html>
Compile
4
Execute
5
helloServlet.class
Request
processing
phase

JSP Compilation

- When a browser asks for a JSP, the JSP engine first checks to see whether it needs to compile the page. If the page has never been compiled, or if the JSP has been modified since it was last compiled, the JSP engine compiles the page.
- The compilation process involves three steps:
  - Parsing the JSP.
  - Turning the JSP into a servlet.
  - Compiling the servlet

  You can see the generated servlet for your own JSPs in the glassfish server directory:

  `….\glassfish\domains\domain1\generated\jsp`

JSP Initialization

- When a container loads a JSP (after translation into a servlet if necessary) it invokes the jspInit() method before servicing any requests. If you need to perform JSP-specific initialization, override the jspInit() method (your code is placed in the method body in a declaration – declarations discussed later).

  <%! **public void** jspInit() { *//your initialization code*} %>
  <html>…</html>

- Initialization is performed only once and as with the servlet init method. You can use jspInit to initialize database connections, open files, and create lookup tables.

JSP Execution

- This phase of the JSP life cycle represents all interactions with requests until the JSP is destroyed.
- Whenever a browser requests a JSP and the page has been loaded and initialized, the JSP engine invokes the **_jspService()** method in the JSP.
- The _jspService() method takes an **HttpServletRequest** and an **HttpServletResponse** as its parameters as follows:

```
void _jspService(HttpServletRequest request,
HttpServletResponse response) {
      // Service handling code...
}
```

JSP Cleanup

- The destruction phase of the JSP life cycle represents when a JSP is being removed from use by a container.
- The **jspDestroy()** method is the JSP equivalent of the destroy method for servlets. Override jspDestroy when you need to perform any cleanup, such as releasing database connections or closing open files.
- The jspDestroy() method has the following form:
  - public void jspDestroy(){

    // Your cleanup code goes here.

    }

JSP Element – Directive

- message from a JSP page to the JSP container that controls processing of entire page.

```
 <%@ page import="java.util.Date" %>
<HTML>
<BODY>
<%    System.out.println( "Evaluating date now" );  Date date = new Date(); %>
Hello!  The time is now <%= date %>
</BODY>
</HTML>
```

```
<HTML>
<BODY>
Going to include hello.jsp...<BR>
<%@ include file="hello.jsp" %>
</BODY>
</HTML>
```

```
<%@ taglib uri="http://www.jspcentral.com/tags" prefix="public" %>
```

JSP Elements – Page Directives

The page Directive:

- The **page** directive is used to provide instructions to the container that pertain to the current JSP page. You may code page directives anywhere in your JSP page. By convention, page directives are coded at the top of the JSP page.
- Following is the basic syntax of page directive:

```
<%@ page attribute="value" %>
```

- Example:

```
<%@ page import="java.util.Date" %>
```

JSP Elements – Include Directives

- The **include** directive is used to includes a file during the translation phase. This directive tells the container to merge the content of other external files with the current JSP during the translation phase. You may code *include* directives anywhere in your JSP page.
- The general usage form of this directive is as follows:

```
<%@ include file="relative url" >
```

- The filename in the include directive is a relative URL. If you just specify a filename with no associated path, the JSP compiler assumes that the file is in the same directory as your JSP.

JSP Elements – Taglib Directives

- The JavaServer Pages API allows you to define custom JSP tags that look like HTML or XML tags and a tag library is a set of user-defined tags that implement custom behavior.
- The **taglib** directive declares that your JSP page uses a set of custom tags, identifies the location of the library, and provides a means for identifying the custom tags in your JSP page.
- The taglib directive follows the following syntax:

```
<%@ taglib uri="uri" prefix="prefixOfTag" >
```

where the **uri** attribute value resolves to a location the container understands and the **prefix** attribute informs the container what bits of markup are custom actions.

Main point 1

The web container generates a servlet from a JSP file the first time the JSP is requested from a web application. Since a JSP is essentially a servlet, one should understand servlets to effectively deal with JSPs. **Science of Consciousness:** Actions in accord with fundamental levels of knowledge promote success in dealing with more expressed values

JSP Predefined Variables = *Implicit Objects*

- request – the HTTPServletRequest object associated with the request
- response – the HTTPServletRespnse object associated with the response
- out – the PrintWriter used to send output to browser
- session – the HttpSession object associated with the request
- application – the ServletContext obtained via getServletConfig()
- config – the ServletConfig object
- pageContext – the PageContext object to get values from and store attributes into any of the other contexts (request, session, servletContext)
- page – a synonym for this – the “this” of the jsp page’s generated servlet; page has its own scope (elements of the page)

The request Object

- Of type javax.servlet.HttpServletRequest
- The request object represents the data submitted by the browser
- Common methods:
  - String getParameter(String) – retrieve form input data
  - Object getAttribute(String) – get an object from the request scope

The response Object

- Of type

  javax.servlet.HttpServletResponse
- Represents the response that will be sent

  back to the browser
- Common methods:
  - void setContentType(String) – sets the content mime format and encoding

The out Object

- Represents the output stream for the page, the contents of which will be sent back to the browser as the body of its response -- of type javax.servlet.jsp.JspWriter.
- Some of the methods of JspWriter interface
  - println() – adds a new line to the stream.
  - close() – closes the output stream
  - getBufferSize() – returns the size of the output buffer (in bytes)
- *Warning:* The JspWriter looks like a PrintWriter, but is not a descendant – only of Writer

PageContext

- Allows easy access to any of the other contexts: page, request, session, application
- A call to pageContext.getAttribute(“key”) will cause the JSP engine to search *page scope* for a value; a call pageContext.getAttribute(“key”, PageContext.SESSION_SCOPE) will search the session context for the key.
- ‘page’ scope means, the JSP object can be accessed only from within the same page where it was created. The default scope for JSP objects created using <jsp:useBean> tag is page.
- Generally: pageContext provides access to all the namespaces associated with a JSP page, provides access to several page attributes

PageContext (continued)

- pageContext.*findAttribute*(“key”) will search all scopes for a matching value starting from page, to request, to session, to application.
- Access to all implicit objects:

  pageContext.getRequest()

  pageContext.getResponse()

  pageContext.getServletContext ()

  pageContext.getServletConfig ()
- Convenience method for forwarding: pageContext.forward(“other.jsp”) is same as

  request.getRequestDispatcher ("other.jsp").forward()
- See http://www.xyzws.com/JSPfaq/what-is-the-pagecontext-implicit-object/36 and http://docs.oracle.com/javaee/1.4/api/javax/servlet/jsp/PageContext.html

EL (Expression Language)

- recall JSP “expression”
  - <%= myMovieList.get(i) %>
  - evaluates the expression and writes it out to the HTML page at its location on the page
  - <%= ((Person) request.getAttribute(“person”)).getDog().getName() %>
- recall that attributes are where web app stores model values
  - values computed in model and then accessed in page for display
    - no model code on the JSP page
- EL simplifies JSP expression syntax
  - ${person.dog.name}

High level description of EL

${something}

- container evaluates this as follows

- checks page scope for an attribute named "something",
  - if found use it.
- otherwise check request scope for an attribute named "something",
  - if found use it.
- otherwise check session scope for an attribute named "something",
  - if found use it
- otherwise check application scope for an attribute named "something",
  - if found use it.
- otherwise ignore the expression.

More detailed description

${firstThing}

- if firstThing is not an implicit EL object, then search page, request, session and application scopes until attribute "firstThing" is found

${firstThing.secondThing}

- if firstThing is a bean then secondThing is a property of the bean
- if firstThing is a map then secondThing is a key of the map

${firstThing[secondThing]}

- if firstThing is a bean then secondThing is a property of the bean
- if firstThing is a map then secondThing is a key of the map
- if firstThing is a List then secondThing is an index into the List

EL is “null friendly”

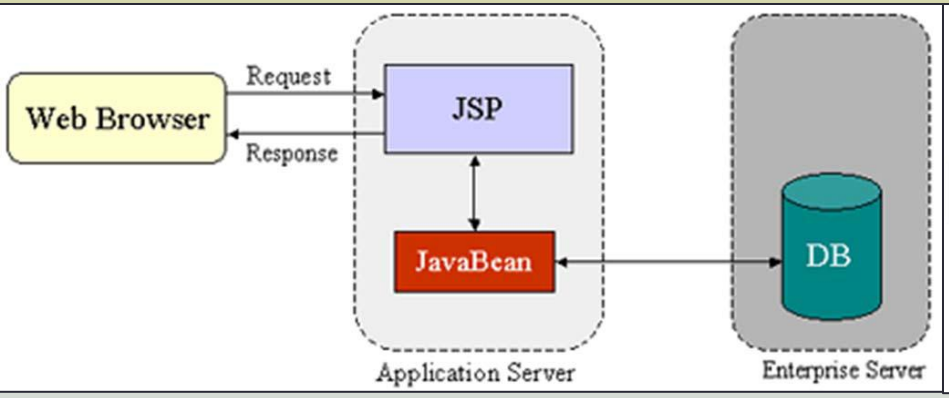
- if EL cannot find a value for the attribute it ignores it
  - caution
  - no warning or error message
- in arithmetic, treats null value as 0
  - could be a surprise
  - ${750 + myBankAccount.balance}
    - “Aaaahhh, where’s all my money!”
- in logical expressions, nulls become “false”
  - more surprises ??

Main point 2

An EL expression is a compact expression of a systematic evaluation of the page, request, session and application scopes. **Science of Consciousness:** The laws of nature are compact expressions that control the infinite diversity of the manifest creation.

JSP Model 1 and Model 2 Architectures

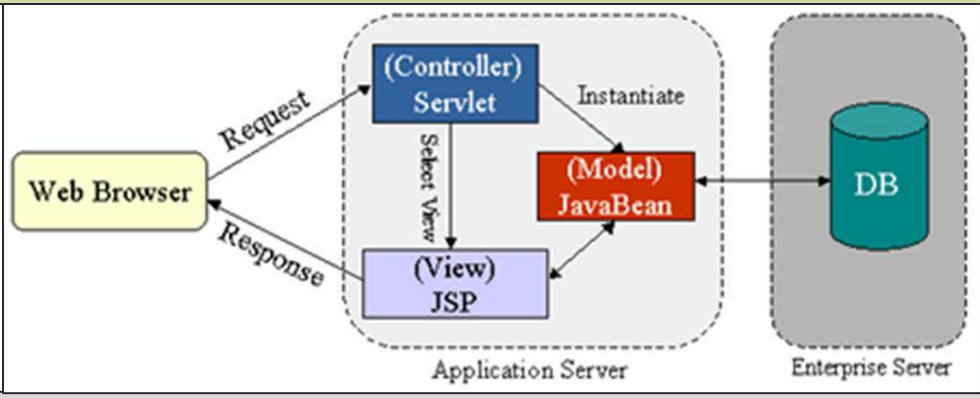
Model 1



- Simple architecture
- For small applications
- Issues

  - Pages are coupled, need to know about each other.
  - Expensive to provide another presentation for same application.
  - Java code in HTML

Model 2



- More complex architecture
- For medium and large applications
- Advantages

  - Each team can work on different pages. Easy to understand each page.
  - Separation of presentation from control, making it easy to change one without affecting the other.
  - no Java code in HTML

Introducing MVC

- Model 1 mixes view and business logic inside each handling servlet (or jsp). This makes it more difficult to change the view independently of that logic, and difficult to change business logic without changing the view.
- Model II cleanly separates business data and logic from the view, and the two are connected by way of a controller. Typical implementation introduces just one servlet which generically dispatches requests to JSP handling code. (This is the approach of the Struts framework.)

MVC

- Separation of M-V-C is important:
- The same data, and the business logic related to it, may need to be presented in several ways: webpage, mobile device, standalone app (Swing, SWT, GWT). If business logic is coupled with servlets, then mobile app and standalone app implementations need to re-implement business logic in another way.
- The mechanism for relating data to presentation (control) also needs to be independent – makes it possible to improve or to introduce a framework solution.

Model 1 vs 2 architecture (cont)

- Model 1:
  - JSP acts as both controller and view
  - JavaBean (POJO) is model
  - Problems:
    - JSPs became very complicated,
    - JSPs contains page navigation logic,
    - JSPs perform validation and conversion of string parameters
- Model 2:
  - Have a single controller servlet that takes requests
  - Gets request parameters
  - Converts and validates them
  - Calls object with business logic to do processing
  - Forwards results to jsp page for display
- http://www.javaworld.com/javaworld/jw-12-1999/jw-12-ssj-jspmvc.html

Example model 1 JSP

```
<H1>Time JSP</H1>

<%
    //the parameter "zone" shall be equal to a number between 0 and 24 (inclusive)
    TimeZone timeZone = TimeZone.getDefault();  //returns the default TimeZone
    if (request.getParameterValues("zone") != null)
        {
           // gets a TimeZone. For this example we're just going to assume
           // its a positive argument, not a negative one.
           String timeZoneArg = request.getParameterValues("zone")[0];
           timeZone = TimeZone.getTimeZone("GMT+" + timeZoneArg + ":00");
        }
    /*
     since we're basing our time from GMT, we'll set our Locale to
     Brittania, and get a Calendar.
    */
    Calendar myCalendar = Calendar.getInstance(timeZone, Locale.UK);
%>

The current time is:
<%= myCalendar.get(Calendar.HOUR_OF_DAY) %>:
<%= myCalendar.get(Calendar.MINUTE) %>:
<%= myCalendar.get(Calendar.SECOND) %>
```

Main point 3

When you use JSP pages according to a Model 2 architecture, there is a servlet that acts as a controller (process of knowing) that sets attribute values based on computations and results from a business model (knower), then dispatches the request to the servlet generated by the JSP page (known). The JSP servlet then retrieves the attribute values and inserts them into the designated places in the HTML being sent to the browser. **Science of Consciousness:** Complete knowledge is the wholeness of knower, known, and process of knowing.